X11/1998-28

MUMPS Development Committee.

Extension to the MDC Standard
Type A Release of the MUMPS Development Committee

Event Processing

June 28, 1998

Produced by the MDC Subcommittee #15
Programming Structures

Art Smith, Chairman
MuMPS Development Committee

Wally Fort, Chairman
Subcommittee #15

The reader is hereby notified that the following MDC specification has been approved by the MUMPS Development Committee but that it may be a partial specification that relies on information appearing in many parts of the MDC Standard. This specification is dynamic in nature, and the changes reflected by this approved change may not correspond to the latest specification available.

Because of the evolutionary nature of MDC specifications, the reader is further reminded that changes are likely to occur in the specification released, herein, prior to a complete republication of the MDC Standard.

Identification of the Proposed Change

1. Title

Event Processing

1.1. MDC Proposer and Sponsor

Arthur B. Smith
Emergent Technologies
4500 County Road 351
Fulton, MO 65251
Voice: (573)882-2666 (weekdays)
Voice: (573)642-8802 (weekends/evenings)
Fax: (573)884-5444
e-mail: SmithAB@missouri.edu
e-mail: Emergent@sockets.net

Subcommittee 15, Task Group 4
Event Processing
Chair: Keith Snell

1.2. Motion

This document (as X11/SC15/1998-6) was approved as MDC Type A at the June 1998 meeting in Boston, MA.

1.3. History

August, 98	X11/1998-28	<This document> Final form (updated history, motion, and discussion, only.
June, 98	X11/SC15/1998-6	History updated. Approved as MDC-A 14:14
March, 98	X11/SC15/TG4/98-1	Incorporated changes approved at the September 97 meeting. Approved with SC-A status 18:0:2.
September, 97	X11/SC15/TG4/97-4	Clarified and extended to meet TG and SC recommendations. Modified in Task Group to change timer behavior and correct typos. Approved as replacement SC-B. 22:1:2
March, 97	X11/SC15/TG4/97-1	Extended to meet TG and SC recommendations, adopted as replacement SC-B 20:0:4
September, 96	X11/SC15/TG4/96-1	Initial proposal, brought forward from task group. Raised to SC-B status: 23:2:5
1991-1995	<various>	A number of proposals discussed event processing. These were examined prior to writing this document but are not direct predecessors. See Section 6 for a list of these documents.

1.4. Dependencies

2. Justification of the Proposed Change

2.1. Needs

Much of modern programming is based on "event driven" programming. Graphical user interfaces require event driven programming. Messaging techniques in object oriented programming require event driven programming. Network protocols requires event driven programming. The ability to handle events in both

synchronous and asynchronous paradigms is now essential.

There are two models for event processing: synchronous and asynchronous. In the synchronous model, often used for graphical user interfaces, the control of the program is turned over entirely to the incoming events . The program runs an "event loop" which processes the incoming events one at a time. When each event is processed, the event loop waits for the next event. No events are processed except when control is in the event loop - events are not interruptable. This is the model used in the MWAPI (ESTART starts the event loop, and ESTOP stops it).

In the asynchronous model, program control follows normal behavior until an event occurs. At that time normal flow is interrupted and the event handler processes the event. When the event has been processed the control returns to the point at which it was interrupted. Because not all sections of processing can be interrupted by all types of events, it is usually necessary to allow selected events to be blocked from interrupting critical sections of programs. When the block is removed, the pending events are processed.

2.2. Existing Practice in the Area of the Proposed Change

Xl 1.6, the M Windowing API introduced the ESTART and ESTOP commands and the ^$EVENT ssvn to handle event processing in a synchronous model. These commands allow the windowing system to respond to a number of special events related to user interaction with the windowing system and other occurrences (e.g., timers).

Error Processing provides a specialized case of asynchronous event processing. Because error conditions can be intentionally generated by setting $ECODE non-empty, this has been used in some cases to provide event processing. This can be ungainly, however, as error processing was not designed to handle generalized event processing.

3. Description of the Proposed Change

3.1. General Description of the Proposed Change

The ESTA[RT] and ESTO[P] commands of the X11.6 MWAPI standard are added to the X11.1 standard with little change. The ETR[IGGER] command is also added, but with some (backwards compatible) modifications. In addition, four additional commands (ASTA[RT], ASTO[P], AB[LOCK] and AUNB[LOCK]) are added to handle asynchronous event processing.

The ASTA[RT] command is used to enable asynchronous event processing. When invoked without arguments, all events are enabled. An argument list can be used to specify that only selected events are enabled, using either a selective or exclusive form. Enabling an event which is already enabled is not an error, and has no effect. Events may be enabled which have not been registered by placing entries in ^$J[OB]. Enabled but unregistered events are ignored until such time as they are registered.

The ASTO[P] command is used to disable asynchronous event processing. When invoked without arguments, all events are disabled. An argument list can be used to specify that only selected events are disabled, using either a selective or exclusive form. Disabling an event which is not enabled is not an error and has no effect. Events may be disabled regardless of whether they are registered in ^$J[OB]. Disabled events, whether registered or not, are ignored.

The AB[LOCK] command is used to temporarily block processing of selected events during critical sections. Events which occur while blocked will be queued and then processed when. the block is removed by the AUNB[LOCK] command. There are up to two queues for pending events for each process - one for synchronous events and a separate one for asynchronous events. The number of events which can be saved in these queue is system dependent. Each system must be able to store at least one pending event in each queue.

Storage of multiple events may be supported by the system in one or both queues. Blocked events which have been stored are to be processed in the order in which they occurred when they are unblocked. Blocked events which are not stored are lost. The events to block or unblock are specified using an argument list, in either an inclusive or exclusive form. Priority levels for events are not defined in this proposal.

ABLOCK works like the incremental form of LOCK - it increments a count of the blocks on the events. A zero value for the count indicates the event is not blocked. A positive value for the count indicates the event is blocked. The count cannot become negative. Calls to AUNBLOCK decrement the count (except that the count cannot be less than zero). Note that a routine which calls AUNBLOCK more times than it called ABLOCK may inadvertently expose an "outer" routine to unexpected interrupts with surprising results.

The ETR[IGGER] command is extended to allow its argument, an especref, to be an expression evaluating to ^$W[INDOW](espec) as currently defined in the MW API, or to ^$J[OB](processid,"EVENT',expr V evclass, evid) where evclass and evid identify a registered synchronous or asynchronous event. In this way, it can be used to generate event conditions for events not associated with windows, elements or menu choices. Use of a processid other than the current job's processid may be restricted in some or all cases by the M vendor. This restricted use does not create an error condition, but also does not generate an event.

An M process registers events by creating an entry in ^$J[OB]. 'These entries are of the form ^$J[OB](processid, "EVENT', evclass, evid), where the value of this node is the entryref of the event handler. As in the MW API, ^$E[VENT] may contain information about the events as they occur.

Asynchronous event handling routines implicitly invoke an argumentless ABLOCK when they are entered as the result of an event. When the event handler is exited, there is an implicit AUNBLOCK of all events.

Event classes which may be registered are COMM, HALT, IPC, INTERRUPT, POWER, TIMER, USER and vendor defined events beginning with Z. All other event classes are reserved for future use. Each of these event classes may be processed using either the synchronous or asynchronous event model.

COMM events use device numbers as the evid values. Not all devices necessarily generate events. What devices generate COMM events, and under what circumstances, is determined by the M vendor.

HALT events are generated when a MUMPS job is halted, either by an explicit HALT command or by a vendor-specified job termination utility. The evid value for a job halted by an explicit HALT command is 1 (the digit "one"). Additional evid values maybe specified by vendors for jobs which are halted by other means.

IPC events use processids as the evid values, and are used for interprocess communication. The ETRIGGER command generates an event coming from the current process (i.e., the one executing the ETRlGGER command). The processid specified in the argument to the ETRlGGER command must be the current processid. If an evid other than the current processid is provided in ETRlGGER, an error occurs. The event may be received by any process which has registered this event (i.e., any process which has created the appropriate node(s) below its ^$JOB(processid, "EVENT", “IPC") node. See 3.2 (Annotated Examples of Use) for a clarifying example.

INTERRUPT events are generated when a user interrupts a running job in some vendor-specified manner (typically by typing Ctrl-C). The range of evid values is specified by the individual vendors.

POWER events, if supported, are generated to indicate imminent power interruption. The range of evid values is specified by the individual vendors.

TIMER events use ^$EVENT("EVENTDEF",”TIMER",TIMERID,"INTERVAL") to identify the running time remaining before or elapsed time after the timer event (in seconds). It also uses ^$EVENT ("EVENTDEF", "TIMER", TIMERID," ACTIVE") and ^$EVENT ("EVENTDEF","TIMER",TIMERID,"AUTO") to control the state of the timer. The timer is started (or restarted) by setting the ACTIVE value non-zero. When the timer runs out (i.e., the "INTERVAL" node goes from a positive to a non-positive value) the timer event occurs

and the INTERVAL value is automatically reset to the AUTO value. Thus if AUTO is true (and positive), this creates a "free-running" timer that fires every AUTO seconds. If AUTO is false (i.e., 0), this is a "single shot" timer that counts down and generates the event as it crosses zero. In the latter case the timer can be queried for the elapsed time since it fired (as INTERVAL will continue to decrement (i.e., become more negative) until it is halted or the INTERVAL value overflows negatively). TIMERID values are names, and the total number of concurrent timers available is determined by the M vendor (this limit may be specified per job, per system or both ways). These TIMERID values are the values for evid in ^$JOB.

USER events are for programmatically generated events. Events are registered using ^$JOB (processid,"EVENT",'LJSER",EVENTID). EVENTID values are names. These events are generated only by calls to ETR[IGGER].

Events occur in the scope of a system (i.e., the set of processes for which $JOB is guaranteed unique). Processing of events is handled by selected processes - those which have registered the event in ^$JOB. If an event is registered by more -than one process, each process' event handler is invoked when the event occurs. The order in which the event handlers are invoked is not specified.

Synchronous and asynchronous event processing can coexist as long as they have non- overlapping event classes. It is an error to start both synchronous and asynchronous event processing on the same event class.

3.2. Annotated Examples Of Use

3.2.1. A POWER event example

This example assumes that the underlying system is capable of receiving a warning of imminent power loss from the UPS which is providing it power. and responding by generating a POWER event with eventid = 1. A given process may want to execute a routine "^SHTDN" to save its state when and if this happens. The following annotated example indicates how it can arrange to do this asynchronously.

```
; Register the POWER event for this process
SET ^$JOB($JOB, "EVENT", "POWER", 1) = "^SHTDN"
; Enable this event for Asynchronous Event Processing
ASTART "POWER"
```

At this point, if the UPS generates the POWER event, this process will transfer control (asynchronously) to ^SHTDN. If this process also has a handler for another event class which should be interruptable by POWER events, that event handler should contain code like:

```
; Unblock POWER events in this handler
AUNBLOCK "POWER"
...the body of the event handler goes here
; Now re-block POWER events so that they are not double-
; unblocked upon exit
ABLOCK "POWER" QUIT
```

3.2.2. IPC Events

Identification of participating processes in IPC events may be confusing. An example will help to clarify the roles of the processes. Suppose the job with processid *X* ("process *X")* wants to register routine ^FROMY to process IPC events from the job with processid *Y* ("process *Y")*. It would need to issue a command like

```
SET ^$JOB (X, " "EVENT" ", " "IPC" ",Y)= "^FROMY"
```

to register this event. When process *Y* is ready to communicate with process *X,* it would issue a command like

```
ETRIGGER "^$JOB (X, " "EVENT" ", " "IPC" ",Y)"
```

This would result in process *X* executing the routine ^FROMY (synchronously or asynchronously, depending on whether an ESTART or ASTART command was issued).

3.3. Formalization

Add to the list in section 2 *(Normative references)*
ANSI Xl 1.6-1995 M Windowing API

Add a new section, 6.3.3
6.3.3 **Event Processing**

Event processing provides a mechanism by which a process can execute specifiable commands In response to some occurrence outside the normal program flow. Event processing can be done using either a synchronous model or an asynchronous model. Synchronous event processing is enabled by issuing the ESTART command, and disabled by issuing the ESTOP command.. Asynchronous event processing is enabled by issuing the ASTART command, and disabled by issuing the ASTOP command. It is possible to temporarily block asynchronous events from being processed using the ABLOCK command. This temporary block is released using the AUNBLOCK command. Events can be generated by running processes using the ETRIGGER command.

Asynchronous event processing and synchronous event processing cannot both be enabled at the same time for any event class.

Events are divided into event classes, and those classes are further divided into event id's. Each event class may be independently enabled, disabled, blocked and unblocked (except that individual event classes may not be disabled in the synchronous model).

The event classes *ire:*

COMM

These are events associated with devices. evid is always a devicexpr for this class of event. Not all devices necessarily generate events. What devices generate COMM events, and under what circumstances is determined by the implementation. It is to be understood that use of COMM events may not be portable.

HALT

HALT events are generated when a process terminates. evid is 1 for processes which halt by an explicit HALT command. Other values may be specified by the implementation to correspond to vendor-specific job termination utilities. It is to be understood that use of these other values may not be portable

IPC

These are events generated by other processes using the ETRIGGER command. The evid values are restricted to valid processids. The evid value will always be the processid of the process which issued the ETRIGGER command.

INTERRUPT

These are events generated by the interruption of a running job in some implementation-specific manner (typically by implementation-specific keyboard commands or job control utilities). Different forms of interrupts may be possible in some implementations, and these may possibly be differentiated by evid values. The valid evid value(s) is determined by the implementor. It is to be understood that use of INTERRUPT events may not be portable

POWER

These are events generated when an imminent loss of power can be anticipated (typically because of a signal from the power source). Different types of warnings may be possible in some implementations, and these may possibly be differentiated by evid values. The evid value(s) is determined by the implementor. It is to be understood that use of POWER events may not be portable.

TIMER

Timer events are generated when a specified interval has elapsed after the timer was set (see ^$EVENT). evid values are names. The implementor may limit the number of concurrent timers available, either by a single process or by the entire M system, or both.

USER

User events are always generated by ETRIGGER commands in the current process. evid values are names.

Z[unspecified]

Z is the initial letter reserved for defining non-standard event classes. The requirement that Z be used permits the unused names to be reserved for future enhancement of the standard without altering the execution of existing routines which observe the rules of the standard.

Only those events which have been registered by creating it node in ^$JOB (processid,"EVENT",evclass,evid) generate action. In those cases the value of the node is an entrvref which specifies the event handler. Asynchronous processing of an event (described below) occurs immediately following the event unless the event is blocked.

Blocked events are saved on one of two per-process event queues (one each for synchronous and asynchronous event classes). Each queue is only guaranteed to hold one event, though they may hold more. Events occurring when the queue is full are lost. Queued events are processed in the order they occurred once they are unblocked. It is possible that blocked events will not execute in the order they occurred if the events are of different event classes, and the event classes are separately unblocked in an order different from the order of occurrence of the events. Disabling an event class via ASTOP or by killing the appropriate node(s) in ^$EVENf or ^$JOB removes all entries of that class from the event queue.

When a registered event is processed in the asynchronous model, the current value of $TEST, the current execution level, and the current execution location are saved in an extrinsic frame on the PROCESS-STACK. The process then increments the block count on all event classes, and implicitly executes the command

DO *handler*

where *handler* is the registered event handler. Note that neither $REFERENCE nor any other shared resource is stacked by this action. If the event handler changes the naked indicator, it may be advisable for it to first NEW $REFERENCE[1]. When the process control returns from the handler, the process decrements the block count on all event classes. The value of $TEST and the execution level are restored, the process returns to the stacked execution location and the extrinsic frame is removed from the PROCESS-STACK.

Synchronous event processing is enabled by the ESTART command, which leaves the process in a wait-for-event state. Events are processed sequentially in the order in which they occur. Each event is added to the per-process synchronous event queue. This queue is only guaranteed to hold one event, though it may hold more. Events occurring when the queue is full are lost. When the process is in the wait-for-event state and there is an event in the queue, the event is processed in the synchronous model.

[1] NEW $REFERENCE passed to MDC-A status (X11/SCI3/1998-4) at the June 1998 meeting. Event Processing references but is not dependent on NEW $REFERENCE.
When a registered event is processed in the synchronous model, the process implicitly executes the command

DO *handler*

where *handler* is the registered event handler. When process control returns from the handler, the process returns to the waiting-for-event state. If the handler executes an ESTOP command, the control implictly performs the number of M QUIT commands necessary to return to the execution level of the most recently executed ESTART command, and then terminates that ESTART command.

When a process is initiated, no event processing is enabled, and no nodes in ^$JOB (processid,"EVENT") are defined. When a process terminates, event processing is implicitly terminated and ^$JOB (processid,"EVENT") is implicitly killed. Any queued events (synchronous or asynchronous event queues) for that process are discarded.

Modify the last sentence of the first paragraph of 7.1.2.3 (Process stack) to be:
Three types of items, or frames, will be placed on the PROCESS-STACK, DO frames (including XECUTEs), extrinsic frames (including exfunc, exvar and asynchronous events) and error frames (for errors that occur during error processing):

Modify the definition of ssvn in 7.1.3 *to include the additional choice*
| syntax of ^$EVENT structured system variable |

Add a new section 7.I.3.x
7.1.3.x ^$EVENT
^$E[VENT] (eventexpr)

eventexpr : : = expr V | einfoattribute |
| EVENTDEF |

note that einfoattribute is defined in XI1.6, the MWAPI standard, along with its semantics.

Nodes under ^$EVENT("EVENTDEF") are used to identify specific behavior of the named events. Node ^$EVENT("EVENTDEF",'TIMER", *timerid*,"INTERVAL"), where *timerid* is a valid evid value for a TIMER event, identifies (if positive) the running time remaining before the timer event (in seconds). This value counts down continuously at the rate of 1/second the corresponding ^$EVENT ("EVENTDEF","TIMER",*timerid*,"ACTIVE") node (see below) evaluates as a tvexpr to 1.

Node ^$EVENT("EVENTDEF","TIMER",*timerid*,"AUTO") , where *timerid* is a valid evid value for a TIMER event, is the value set into ^$EVENT("EVENTDEF","TIMER",*timerid*,"INTERVAL") when it is decremented from a positive value to a non-positive value.

Node ^$EVENT("EVENTDEF","TIMER",*timerid*, "ACTIVE"), where *timerid* is a valid evid value for a TIMER event, identifies the state of the timer. If the node evaluates as a tvexpr to 1, the timer is active (running). If the node evaluates as a tvexpr to 0, the timer is inactive.

All of these nodes must be set to establish the timer. If any of the nodes are killed, no timer event occurs.

Add the following to section 7.1.3.4, ^$JOB:
^$JOB (processid, expr V "EVENT", expr V evc1ass, evid) = entryref

This node identifies the events which are enabled for event processing under either the synchronous or asynchronous event processing models, and specifies the event handler which is invoked to process the event. Setting this node enables the specified events for event processing. Killing this node disables the specified events for event processing, and removes all child nodes, even if KVALUE is used. Implementations are expected to support all of the specified evclass and evid values with the understanding that some events may

never occur on a given implementation. If an evclass or evid not defined in the standard is used an error occurs with an ecode = M38. Attempting to set this node when evid cannot be registered due to resource availability will produce an error with an ecode = "M11O".

^$JOB (processid, expr V "EVENT", expr V evclass, evid, "MODE") = | "DISABLED" | | "SYNCHRONOUS" | | "ASYNCHRONOUS" |

This node identifies the processing mode for the specified event by the specified process. If the specified event class is currently enabled for asynchronous event processing by this process (see *8.2.u,* ASTART), the value will be "ASYNCHRONOUS". If the specified event class is currently enabled for synchronous event processing by this process (see *8.2.x,* EST ART), the value will be "SYNCHRONOUS". If the specified event class is not enabled for either form of processing by this process, the value will be "DISABLED".

^$JOB (processid, expr V "EVENT", expr V evclass, evid, "BLOCKS") = intlit

This node gives the count of blocks (see 8.2.*t*, ABLOCK, and *8.2.w,* AUNBLOCK) on the specified event for the specified process. It only exists if the event class is enabled in either synchronous or asynchronous event processing modes. If the value of the node is zero, the events are not blocked. If the value is greater than zero, the events are blocked.

Add six new commands to section 8.2:

8.2t ABLOCK

AB [LOCK] postcond SP [| L evclass |] | (L evclass |)

evclass : : = expr V | COMM | | IPC | | INTERRUPT | | POWER | | TIMER | | USER | | Z (unspecified) |

Event classes not specified above are reserved for future use.

ABLOCK temporarily blocks events during critical sections of a process. The three forms of ABLOCK are given the following names:

a) L evclass	Selective ABLOCK
b) (L evclass)	Exclusive ABLOCK
c) Empty argument list:	ABLOCK All

In the Selective ABLOCK, the named event classes are blocked as described below. In the Exclusive ABLOCK, all event classes except the named event classes are blocked as described below. In the ABLOCK All, all event classes are blocked as described below.

When an event class is blocked, an internal counter for that event class is incremented. If the counter has a positive value, all events of that class are blocked from interrupting the process executing the ABLOCK command. If a registered event occurs while blocked, the event is queued. Unregistered events are not queued. Additional subsequent events may be queued if space is provided by the implementation (space for only one

event is guaranteed). Events, if queued, will occur in the order in which they occurred when the block is removed (i.e., when the counter becomes zero). All events for a process are stored in one of two queues (one for synchronous events, the other for asynchronous events), rather than a separate queue for each class. Each process, however, must maintain its own queues, as each process blocks and unblocks events independently.

8.2.*u* **ASTA[RT]**

ASTA[RT] postcond SP [| L evclass |]
| (L evclass) |

ASTART enables asynchronous event processing for all or selected event classes. Then three forms of AST ART are given the following names:

a) L evclass	Selective ASTART
b) (L evclass)	Exclusive ASTART
c) Empty argument list:	ASTART All

In the Selective ASTART, the named event classes are enabled for asynchronous event processing as described below. In the Exclusive ASTART, all event classes except the named event classes are enabled for asynchronous event processing as described below. In the ASTART All, all event classes are enabled for asynchronous event processing as described below.

If any of the classes being enabled for asynchronous event processing are currently enabled for synchronous event processing an error occurs with an ecode = "M102".

Event classes are enabled by ASTART only for the process executing the ASTART command. It is not an error to enable an event class which is already enabled for the asynchronous model.

8.2.v **ASTO[P]**

ASTO[P] postcond SP [| L evclass |]
| L evclass |

ASTOP disables asynchronous event processing for all or selected event classes. The three forms of ASTOP are given the following names:

a) L evclass	Selective ASTOP
b) (L evclass)	Exclusive ASTOP
c) Empty argument list:	ASTOP All

In the Selective ASTOP, the named event classes are disabled for asynchronous event processing as described below. In the Exclusive ASTOP, all event classes except the named event classes are disabled for asynchronous event processing as described below. In the ASTOP All, all event classes are disabled for asynchronous event processing as described below.

When asynchronous event processing is disabled for a given event class, events of that class have no effect on the process. Event classes are disabled by ASTOP only for the process executing the ASTOP command. It is not an error to disable an event class which is already disabled.

8.2.*w* **AUNBLOCK**

AUNB[LOCK] postcond SP [| L evclass |]
| (L evclass) |

AUNBLOCK removes a temporary block on events that was imparted by ABLOCK. The three forms of AUNBLOCK are given the following names:

a) L evclass	Selective AUNBLOCK
b) (L evclass)	Exclusive AUNBLOCK
c) Empty argument list:	AUNBLOCK All

In the Selective AUNBLOCK, the named event classes are unblocked as described below. In the Exclusive AUNBLOCK, all event classes except the named event classes are unblocked as described below. In the AUNBLOCK All, all event classes are unblocked as described below.

When an event class is unblocked, the internal counter for the event class (see 8.2.*t* ABLOCK) is decremented, unless it is already zero (the counter may not be negative). If the counter is zero, the temporary block, if any, on the event class is removed. Pending events (see 8.2.*t* ABLOCK), if any, occur in the order in which they arrived. Blocks are removed only for the process executing the AUNBLOCK command. It is not an error to unblock events which are not currently blocked.

8.2.x ESTA[RT]

ESTA[RT] postcond SP [| L wevclass |] | (L wevclass) |

wevclass : : = | evclass | | expr V “WAPI” |

ESTART enables synchronous event processing for the selected event classes. The additional class "WAPI" is provided to enable just the synchronous event processing specified in X 11.6, the MWAPI. If any of the event classes being enabled for synchronous event processing is currently enabled for asynchronous event processing, an error occurs with ecode = "M102". It is not an error to enable an event class which is already enabled for synchronous event processing.

Synchronous event processing remains activated until the termination of execution of the ESTART command, except that synchronous event processing is implicitly deactivated at the initiation of call back processing for each event. At the conclusion of call back processing for each event, synchronous event processing is implicitly reactivated.

The three forms of ESTART are given the following names:

a) L evclass	Selective ESTART
b) (L evclass)	Exclusive ESTART
c) Empty argument list:	ESTART All

In the Selective ESTART, the named event classes are enabled for synchronous event processing as described below. In the Exclusive ESTART, all event classes except the named event classes are enabled for synchronous event processing as described below. In the ESTART All, all event classes are enabled for synchronous event processing as described below.

When synchronous event processing is enabled for a given event class, events of that class will cause the execution of the registered event handler, if any, for that specific event (call back processing). Event classes are enabled by ESTART only for the process executing the ESTART command.

Call back processing can execute an EST ART command. In this case, the effect is to change the event classes which are enabled for subsequent synchronous event processing. ESTART commands are not nested. It is not an error to issue a second ESTART command on the same event classes.

The execution of an ESTART command which starts synchronous event processing is terminated when an ESTOP command is executed during call back processing for that EST ART command. When execution of an ESTART command which starts synchronous event processing is terminated, execution continues with the command following that EST ART command.

8.2.*y* ESTOP
ESTO[P] postcond [SP]

The ESTOP command implicitly performs the number of QUIT commands necessary to return to the execution level of the most recently executed ESTART command that started synchronous event processing, and then terminates that EST ART command. If synchronous event processing is not activated, execution of an ESTOP command has no effect. It is not possible to ESTOP only selected event classes.

8.2.*z* ETRlGGER
ETR[IGGER] postcond SP especref

```
especref : : =   | expr V ^$W[INDOW] (espec) [ :einforef ] |
                 | expr V ^$J[OB] ( erspec )                |
```

Note: espec and einforef should be as defined in X11.6, the MWAPI

erspec : : = processid, “EVENT”, expr V evclass, expr V evid

evid : : = expr

Note that the range of values allowed for evid depends on the value of evc1ass, and may be implementation specific.

ETRlGGER causes an event to occur, though use of a processid other than the current job's own processid may be restricted by the implementation. This restricted use does not generate an error, but will not generate an event. Restrictions (if any) must be specified in the implementation's conformance statement.

If the use is not restricted and the specified event is enabled for either synchronous or asynchronous event processing, the event processing for it will occur subsequently. The event that occurs is specified by evc1ass and evid. If evid does not specify a valid event, an error condition occurs with an ecode = "M103".

If evc1ass evaluates to "IPC" and evid is not the current job's processid an error condition occurs with an ecode ="M104".

Add a new section (##) between section l l and section 12 of the Portability Requirements:

Event processing

##.1 Number of timers
The number of concurrently running timers must not exceed one (1) per process or sixteen (16) per system, whichever is smaller.

##.2 Depth of event queues
The per-process event queues (one each for synchronous and asynchronous events) must not

contain more than one event.

##.3 Resolution of timers
Timers must not use a resolution finer than one second.

Modify section 3.1 *(Conformance/Implementations) by adding 10 the text to be included (two places):*

"The depth of event queues is..."
"The number of timer events is..."
"The resolution of timers is..."

Modify the first paragraph of section 12 *of the Portability Requirements to read:*

Programmers should exercise caution in the use of non integer values for the HANG command and in TIMER events and timeouts. In general, the period of actual time which elapses upon the execution of a HANG command, or which elapses before a TIMER event cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.

4. Implementation Effects

4.1 Effect on Existing User Practices and Investments

No existing code will need to be changed as a result of this proposal. This change is compatible with the ANSI/Xl1.6-1995 MW API standard. Event Processing as defined herein will facilitate networking, communications, and other applications that require event processing capabilities.

4.2 Effect on Existing Vendor Practices and Investments

One vendor representative indicated that much of the underlying effects of this proposal already exist in the ir products to handle the process control inherent in M. They indicated further that they expect that this is the case with all M vendors. In a previous version of this proposal, two vendors were represented in the vote. One voted affirmatively, the other abstained, and no CONs were raised. In the version previous to this, three vendors were present, and all three voted in favor of the proposal. No CONS concerning vendor impact were raised at that time. Vendors are invited to comment further upon this! [In the final vote, only one vendor was present and They voted against it citing implementation difficulties. ABS 8/98]

4.3 Techniques and Costs for Compliance Verification

None

4.4 Legal Considerations

None

5. Closely Related Standards Activities

5.1 Other X11 Proposals Under Consideration

X 11.6, the MWAPI continues to evolve. Modifications to this standard need to be examined to make sure that the two standards remain compatible in their event processing.

X11/SC13/1998-4, NEW $REFERENCE, or similar functionality, will likely be desired by people writing event handlers which modify globals. [As noted earlier this is now at MDC-A. ABS 8/98]

5.2 Other Related Standards Efforts

None

5.3 Recommendations for Coordinating Liaison

None

6. Associated Documents

X11.6-l99S, M Windowing API, defines synchronous event processing used for windowing.

X11/SC13/1998-4, NEW $REFERENCE, is referenced by this document, but is not a dependency.

Historical documents dealing with event processing include:

??? (Oct 22, 1991)	Event Management (Alfredo Garcia)
X11/SC1/91-4	Proposal for Event Processing in MUMPS
X11/SC1/91-81	Error Processing
X11/SC1/91-82	Summary of differences between X11/SC1/91-43 and X11/SC1/91-81
X11/SC1/TG19/91-3	Synchronous Event Processing
X11/SC1/TG19/91-3A	Asynchronous Event Processing
X11/SC15/TG1/91-1	Error Processing
X11/SC15/TG4/92-3	Synchronous Event Processing
X11/SC15/TG4/WG1/92-4	Unified Event Processing Proposal
X11/SC15/TG4/WG1/92-6	Notes on Event Processing for MUMPS
X11/SC15/TG4/WG1/92-7	Interprocess Communication using Event Queue

7. Issues, Pros and Cons, and Discussion

October 1995, New Orleans
Discussion of how to handle event processing in SC15/TG4, Event Processing. The basic ideas of this proposal were laid out and an author was appointed.

March 1996, Boston
No document was available for this meeting, but there was additional informal discussion of the proposal ideas.

September 1996, Toronto
Discussion in Task Group indicated that this proposal was a good start, but needed to be fleshed out, pending SC approval. In particular, several suggestions were made:

1. Additional evclasses should be considered for inter-process communications (may use COM), power fail identification (ditto), notification of error trapping (separate from error handling?) and "control-C" behavior.
2. evclass specific behavior is lacking in the formalism.
3. ABLOCKs should be counted in the manner of incremental LOCKs, with a corresponding number of AUNBLOCKs required to remove the block.
4. There should be an automatic implicit BLOCK of all (or, if prioritization is used all of this or lower priority) events when entering an ASYNC handler, with an implicit UNBLOCK of any events not explicitly UNBLOCKed by the handler when it exits.
5. An asynchronous event handler should stack $TEST, and $REFERENCE, at least. It is noteworthy that no other stack frames save $REFERENCE.
6. Metasymbols timerno and eventno should be changed to timerid and eventid, respectively.
7. Priority levels for interrupts should be considered.
8. An event queue for each evclass should be considered, rather than a single event queue.
9. Timers might be extended to the form in X11/SC15/TG4/93-3, using ACTIVE, INTERVAL and REMAIN.
10. The history and related documents sections are clearly deficient!
11. The formalism might be rewritten to include a discussion of behavior (akin to that used in Transaction Processing), thus eliminating redundancy.
12. It might be nice if all names began with E (EABLOCK, EAUNBLOCK, ...).
13. The status of these things when a process is initiated and halted need to be explicitly addressed.
14. It would be desirable to have the status of block/unblock and event start/stop available somewhere (e.g., ^$JOB ($JOB,"EASTART'».
15. In ABLOCK, evclass needs to reserve all other names.
16. Several areas should be rewritten for clarity. I

In addition, the Subcommittee raised the following instructive PROs and CONs:

PRO	CON
1) Good Start	1) Does not deal with process context
2) Long wanted and requested	2) Should stack blocks and unblocks
	3) Need implicit block when entering event handler and unblock when exiting
	4) History/Associated documents inadequate

These issues are considered in the next version of the document

March 1997, San Diego

The proposal, as published prior to the meeting addresses concerns 1-6, 10-11, 13, and 15-16.
Concern 7 (priority levels) was not addressed because of unanswered questions (see below)
Concern 8 (event queues per event class) was considered but rejected because of loss of sequencing information.
Concern 9 was considered but rejected because it was believed unnecessary.
Concerns 12..and 14 were left to straw polls (see below).
Several straw polls are suggested prior to the meeting. The results of the straw polls taken at the Task Group meeting are shown in italics following the questions.:

- Are event priorities desirable (Y or N) How should they be assigned to standard, vendor- and user-defined events? Do we rank event classes (POWER, INTERRUPT, COMM, TIMER, IPC, USER, with Z... stuck whereever?)? Can different priorities exist within an event class? *No event priorities are required.*
- Is it necessary to stack $REFERENCE? (Y or N) *No, the overhead cost is too high given that many (most?) event handlers will not modify globals, and those that do should be able to use NEW $REFERENCE to protect the naked indicator.*
- Should AB[LOCK], AUNB[LOCK], ASTA[RT] and ASTO[P] be changed to EAB[LOCK],

EAUNB[LOCK], EASTA[RT] and EASTO[P], or something else less ungainly (five letter abbreviations of six letter words seems ridiculous). *Leave them as they are.*

- Should there be a separate queue for each event class? (Is this feasible?) *No (it would require event priorities), but there should be separate queues for synchronous and asynchronous events.*
- Should the status (enabled synch, enabled asynch, blocked asynch, disabled) of each event class be available somehow in ^$JOB? How? *These should be stored in ^$EVENT (but see below) using some appropriate nodes. Separate into status (enabled synch, enabled asynch, disabled) and block count.*

A number of changes were suggested at the Task Group meeting. The minutes of that meeting (X11/SCI5/TG4/97-3) can be consulted for a complete list, but the substantive changes were:

1. Extend the TIMER event class to support free-running timers in addition to one-shot timers.
2. Add a HALT event class to support job termination events.
3. AUNBLOCK should unblock all events, including those which were explicitly unblocked in the event handler. These can be explicitly excluded if necessary, and it simplifies the behavior of AUNBLOCK.
4. For each registered event, the count of BLOCKs and the mode (enabled asynchronous, enabled synchronous or disabled) should be exposed (read-only) in the ^$EVENT ssvn. (but see below)
5. There should be separate event queues for synchronous and asynchronous events.

Subcommittee approved the document as a replacement Subcommittee Type B by a vote of 20:0:4. There were three PROs ("Long wanted and requested", cited 9 times; "Separates event from error processing", cited 4 times; and "Has been mostly implemented", cited 4 times) and no CONs raised.

September 1997, Chicago

The document, as published prior to the meeting includes all the changes recommended by the Task Group and Subcommittee, including the five noted above, though change number 4 was altered. The count of BLOCKs and mode are not per-event states, but are rather per-process states. A given event (say TIMER) may be registered by several processes, blocked or, unblocked independently by those process (with blocked events being stored in the per-process event queue), and may, in fact, be handled synchronously by some processes and asynchronously by others. For this reason, the block count and mode were placed in read-only nodes of ^$JOB, rather than ^$EVENT. This document is the first to cite error codes M102 (Simultaneous synchronous and asynchronous event class), MI03 (Invalid event ID) and M104 (IPC event ID is not $JOB).

When reviewing the document, the Task Group noted several minor errors (e.g., missing quotation marks and inconsistent use of variable names), and clarified the wording regarding unsupported evlcasses and evids. In addition, a series of straw polls conducted in the Task Group indicated that the timer should retain the three control parameters (INTERVAL, AUTO and ACTIVE); rather than taking a four parameter (INTERVAL, RESET, MODE and ACTIVE) approach. It was also decided that active timers never stop (they count down to zero and continue negatively). A set of corrigenda were prepared to reflect these changes and they were presented (on overhead slides) to the Subcommittee when considering the document for replacement Type B status. The motion, including the corrigenda, passed. No further changes will be made prior to the March meeting. Passed 22:1:2, with the one Pro (Needed) cited nine times and the one Con (Big job for some vendors) cited three times.

March 1998, Atlanta

Proposed and passed as Subcommittee Type A without modifications. Passed 18:0:2. Pros and Cons are shown below with the number of citations in parentheses.

PRO	CON
1) Needed functionality (4)	1) Big job for some vendors (3)
2) Compliments Object Usage proposal (2)	2) Not needed for Object Usage (1)

June 1998, Boston

Proposed and passed as MDC Type A with two editorial corrections (one missing word, one extra word) .
Passed 14:1:4 with the following Pros and Cons (citations noted in parentheses):

PRO	CON
1) Needed functionality (4)	1) Expensive to implement (1)
2) Complements Object Usage proposal (4)	
3) Would have been implemented (3)	
4) Good starting list of events (3)	
5) Easy to add more events (3)	