



33 South La Patera Lane  
Santa Barbara, CA 93117-3214  
ph (805) 681-3300  
fax (805) 681-3311  
tech@motioneng.com  
www.motioneng.com

# APPLICATION NOTE 215, REV. C

## User Limits

### INTRODUCTION

This application note describes User Limits for post-20000913xx versions and releases of MPI/XMP software. User limits are a means by which a user can configure and generate custom XMP events via the mpiMotorEventConfigGet/Set(...) functions. These events will be treated like other events (such as MPIEventTypeMOTION\_DONE) and can be passed to event managers and used by notify objects. A User Limit will evaluate conditional statements on memory registers in the XMP and generate an Event when those registers meet the specified conditions. Also, when the event is generated, the Limit can write an output word (or bit) to a user-defined register in the XMP memory.

Initialization of user limits will require the configuration of three structures. The first, MEIXmpLimitData, enables the user limit and includes the general configuration data. The second, MEIXmpLimitCondition, includes parameters of the actual conditions that are evaluated to generate the event. In addition to generating an Event that is returned to the host, the User Limits can be configured to write an Output value to an XMP memory register. This is configured with the last user limit data structure, MEIXmpLimitOutput. Descriptions of the data structures are below.

Sample applications demonstrating the use of User Limits can be found in the mei\xmp\app directory.

Document Revision History: Application Note 215				
Rev.	Date	Description	DCR	ECO
C	22APR2002	Edit to Introduction section	611	1598
B	13FEB2001	For general releases after 20000913xx.	530	
A	18JAN2001	For general releases 20000913xx and earlier.	524	

## MEIXmpLimitData

```
typedef struct {
    MEIXmpLimitCondition  Condition[MEIXmpLimitConditions];
    MEIXmpStatus          Status;
    MEIXmpLogic           Logic;
    MEIXmpLimitOutput     Output;
    long                  Count;
    long                  State;
} MEIXmpLimitData;
```

MEIXmpLimitConditions is currently defined to have the value of two. This allows the user the option of evaluating two conditions and then either logically ANDing or logically ORing them together.

Status defines what actions the XMP will take when a user limit evaluates TRUE. Always set Status to at least MEIXmpStatusLIMIT to notify the motor object that a limit has occurred.

Value of Status †‡	Action to be taken
MEIXmpStatusLIMIT	None
MEIXmpStatusLIMIT   MEIXmpStatusPAUSE	Axes attached to the motor will be Paused
MEIXmpStatusLIMIT   MEIXmpStatusSTOP	Axes attached to the motor will be Stopped
MEIXmpStatusLIMIT   MEIXmpStatusABORT	Axes attached to the motor will be Aborted
MEIXmpStatusLIMIT   MEIXmpStatusESTOP	Axes attached to the motor will be E-Stopped
MEIXmpStatusLIMIT   MEIXmpStatusESTOP_ABORT	Axes attached to the motor will be E-Stopped and Aborted

† If MEIXmpStatusLIMIT is not included in the value of Status, then if another action is taken (i.e. an abort event if Status = MEIXmpStatusABORT) an application will not be able to identify why the action was taken.

‡ Do not OR more than one MEIXmpStatus bit to MEIXmpStatusLIMIT. For example, do not set Status = MEIXmpStatusLIMIT | MEIXmpStatusABORT | MEIXmpStatusESTOP.

MEIXmpLogic is the logic applied between the two condition block outputs, Condition[0] and Condition[1].

Value of MEIXmpLogic	Evaluates	Motor object notified that a limit has occurred if...
MEIXmpLogicNEVER	Nothing	No event is generated
MEIXmpLogicSINGLE	Condition[0]	<i>Condition[0] == TRUE</i>
MEIXmpLogicOR	Condition[0], Condition[1]	<i>(Condition[0]    Condition[1]) == TRUE</i>
MEIXmpLogicAND	Condition[0], Condition[1]	<i>(Condition[0] &amp;&amp; Condition[1]) == TRUE</i>
other MEIXmpLogic enums	For internal use only.	

When finished with user limits, it's a good idea to set the Logic to MEIXmpLogicNEVER so that the XMP will no longer use background time to process these "dead" events.

*Count and State are for internal use only!* The MPI method, mpiMotorEventConfig-Set(...) will not write these values.

## MEIXmpLimitCondition


```
typedef struct {
    MEIXmpLimitType      Type;
    void                 *SourceAddress;
    long                 Mask;
    MEIXmpGenericValue   LimitValue;
} MEIXmpLimitCondition;
```

SourceAddress is a pointer to an XMP memory location. Use the mpiMemory...() functions to obtain pointers to the structures of interest. Please see the sample applications for specific examples.

A condition will evaluate TRUE if:

```
( (*SourceAddress & Mask)  Type  LimitValue.l )   for long comparison Type's
( *SourceAddress  Type  LimitValue.f )             for float comparison Type's
```

evaluates TRUE, where *Type* represents some comparison type such as "<".

MEIXmpLimitType	Meaning	Mask ANDed to (*SourceAddress) <sup>†</sup>
MEIXmpLimitTypeFALSE	Condition evaluates FALSE	No
MEIXmpLimitTypeTRUE	Condition evaluates TRUE	No
MEIXmpLimitTypeGT	> (long data types)	Yes
MEIXmpLimitTypeGE	>= (long data types)	Yes
MEIXmpLimitTypeLT	< (long data types)	Yes
MEIXmpLimitTypeLE	<= (long data types)	Yes
MEIXmpLimitTypeEQ	== (long data types)	Yes
MEIXmpLimitTypeBIT_CMP	== (bit masks)	Yes
MEIXmpLimitTypeNE	!=	Yes
MEIXmpLimitTypeABS_GT	*SourceAddress & mask  > (long data types)	Yes
MEIXmpLimitTypeABS_LE	*SourceAddress & mask  <= (long data types)	Yes
MEIXmpLimitTypeFGT	> (float data types)	No
MEIXmpLimitTypeFGE	>= (float data types)	No
MEIXmpLimitTypeFLT	< (float data types)	No
MEIXmpLimitTypeFLE	<= (float data types)	No
MEIXmpLimitTypeFEQ	== (float data types)	 YES
MEIXmpLimitTypeFNE	!= (float data types)	
MEIXmpLimitTypeFABS_GT	*SourceAddress  > (float data types)	No
MEIXmpLimitTypeFABS_LE	*SourceAddress  <= (float data types)	No

<sup>†</sup>To be safe set Mask=0xFFFFFFFF whenever a long or float comparison is desired.

## MEIXmpLimitOutput

```
typedef struct {
    long    AndMask;
    long    OrMask;
    long    *OutputPtr;
    long    Enabled;
} MEIXmpLimitOutput;
```

\*OutputPtr is a pointer to an XMP memory location. Use the mpiMemory...() functions to obtain the pointers of interest.

AndMask is a bit mask that will be bit-wise ANDed with the data pointed to by OutputPtr.

OrMask is a bit mask that will be bit-wise ORed with the result of (AndMask & \*OutputPtr).

Enabled tells the XMP whether or not to use the MEIXmpLimitOutput structure. It takes either TRUE or FALSE values.

Effectively, if a user limit evaluates TRUE, then the MEIXmpLimitOutput structure is used as follows:

```
if (Enabled) {
    *OutputPtr = OrMask | ( AndMask & (*OutputPtr) );
}
```

Here is an example showing what happens for each combination of bits for \*OutputPtr, AndMask, and OrMask:

Bit Representations								Data
0	0	0	0	1	1	1	1	*OutputPtr
AND								
0	0	1	1	0	0	1	1	AndMask
↓								
0	0	0	0	0	0	1	1	Result
OR								
1	0	1	0	1	0	1	0	OrMask
↓								
1	0	1	0	1	0	1	1	*OutputPtr
								(result is written back to *OutputPtr)

The setup of the five most common output operations are outlined below:

To set bit(s), set:

```
Enabled      =    TRUE;
AndMask      =    0xFFFFFFFF;
OrMask       =    ( bit(s) to set );
```

To clear bit(s), set:

```
Enabled      =    TRUE;
AndMask      =    ~( bit(s) to clear );
OrMask       =    0;
```

To set a long value, set:

```
Enabled      =    TRUE;
AndMask      =    0;
OrMask       =    ( value );
```

To set a float value, set:

```
Enabled      =    TRUE;
AndMask      =    0;
OrMask       =    *(long*)( & ( float variable ) );
```

or

```
MEIXmpGenericValue generic;

Enabled      =    TRUE;
AndMask      =    0;
generic.f    =    ( value );
OrMask       =    generic.l;
```

To not set any output:

```
Enabled      =    FALSE;

/* To be safe, these values won't change value of (*OutputPtr) if Enabled=TRUE */
AndMask      =    0xFFFFFFFF;
OrMask       =    0;
```

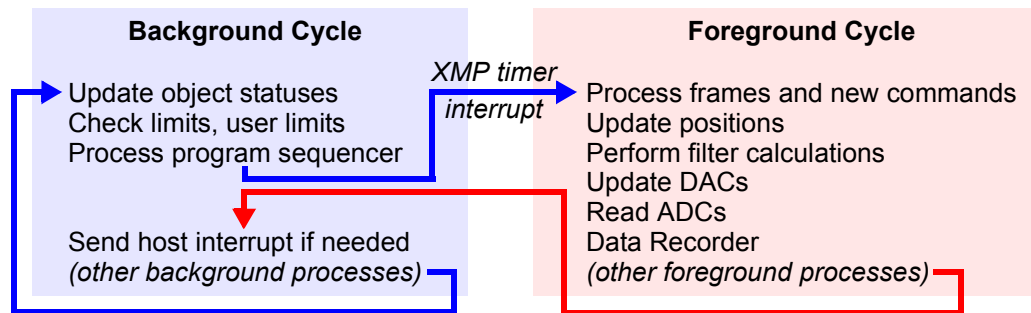
Unlike user limit events, the output structure is used every time the user limit evaluates TRUE, not just when the user limit changes from a FALSE state to a TRUE state.

## Performance Characteristics

It is important to understand that user limits:

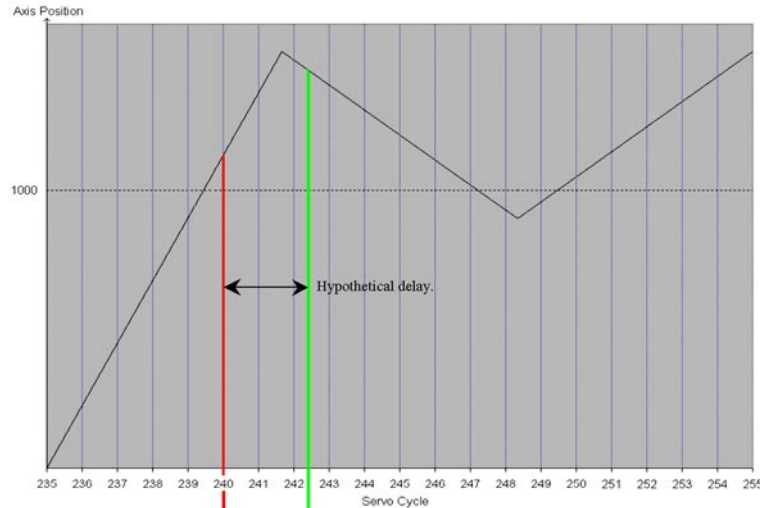
- are evaluated in the background cycle.
- trigger events only when the limit changes state from FALSE to TRUE. In other words, a user limit **must** be reset by changing its state to FALSE before it can trigger another event.

The foreground cycle processes crucial information that needs to be updated every servo cycle. The background cycle processes all other information. The background cycle runs continuously on the XMP as often as it can. A foreground cycle starts when a timer interrupt on the XMP puts the background cycle on hold. After the foreground cycle is done, the background cycle continues running. The foreground cycle will be started at regular intervals at the rate of once per servo cycle. The background cycle runs as quickly as possible with whatever spare time the foreground cycle does not use.



Usually the background cycle will complete many cycles in the time it takes to complete a servo cycle. However, the time to take to complete a single background cycle can end up spanning many servo cycles if the sample rate is raised or if the foreground cycle takes more time to process. If an application requires a high number of XMP objects and features or requires a fast sample rate, it is possible that background cycles could take multiple servo cycles to process, even dozens if the XMP is pushed to its limits.

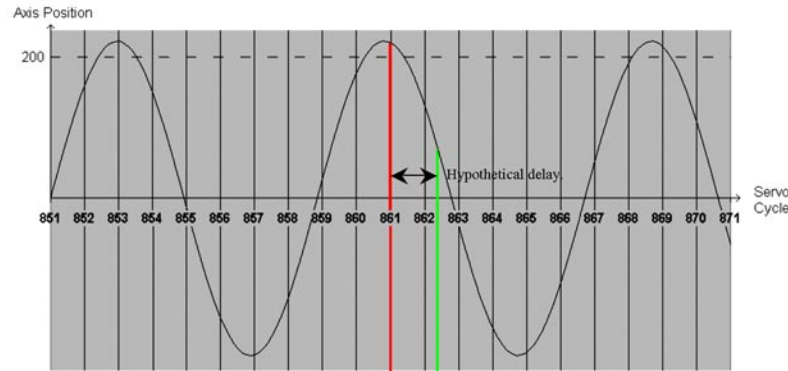
Since user limits are evaluated in the background cycle, events might not be generated or output might not be written in the same servo cycle as when the conditions for the user limit would otherwise first be evaluated as TRUE. In the example below, a user limit is set up to evaluate TRUE when the axis position is greater than 1000. We would expect the conditions of the user limit to evaluate TRUE for sample 240. The XMP's background cycle, however, does not evaluate the user limit until sample 242. Therefore, an event is triggered two samples later than one would hope. The delay of two samples shown here is not indicative of typical XMP setups. Background cycles are commonly evaluated more quickly than foreground cycles, but as explained above, it is possible for the foreground cycle to process more frequently.



XMP updates the position. This is the first point in time that the limit's conditions can be interpreted as TRUE.

This is the next point in time where the background cycle actually evaluates the limit's conditions.

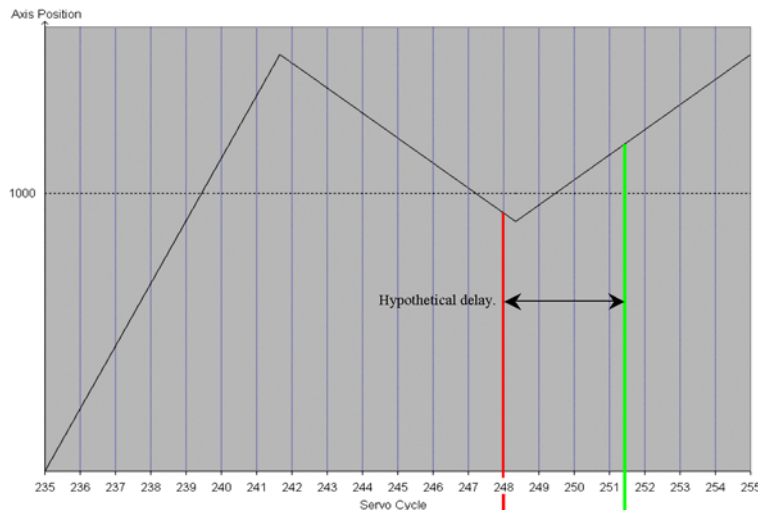
The following example shows how it is possible to even miss a user limit when the conditions that would cause the user limit to evaluate TRUE change too quickly back to FALSE. In this example, a user limit has been set up to evaluate TRUE when the axis position is greater than 200. The axis is performing sinusoidal motion of amplitude 220 encoder counts with an approximate period of eight samples. If the background cycle delays evaluating the user limit by even one servo cycle, it will miss triggering an event.



XMP updates the position. This is the first point in time that the limit's conditions can be interpreted as TRUE.

This is the next point in time where the background cycle actually evaluates the limit's conditions. Now, the limit's conditions are FALSE and the limit missed triggering an event.

We now return to our first example to show the corollary to the previous example. It is equally possible to miss resetting the state of the user limit to FALSE after the position drops below 1000, so that it will not trigger an event when the conditions for the user limit to evaluate TRUE occur again.



XMP updates the position. This is the first point in time that the limit's conditions can be interpreted as FALSE.

This is the next point in time where the background cycle actually evaluates the limit's conditions. Now, the limit's conditions are TRUE and the limit missed being reset.