

# Filter Objects

## Introduction

A **Filter** object manages a single filter on a controller. It represents the control algorithm used to control a motor in a closed-loop system. The Filter contains an algorithm, a set of coefficients, inputs, and an output. Its primary responsibility is to take the difference between the command and actual positions and then calculate the output based on the control algorithm and coefficients.

For simple systems, there is a one-to-one relationship between the Axis, Filter, and Motor objects.

## Methods

### Create, Delete, Validate Methods

<a href="#"><u>mpiFilterCreate</u></a>	Create Filter object
<a href="#"><u>mpiFilterDelete</u></a>	Delete Filter object
<a href="#"><u>mpiFilterValidate</u></a>	Validate Filter object

### Configuration and Information Methods

<a href="#"><u>mpiFilterConfigGet</u></a>	Get Filter configuration
<a href="#"><u>mpiFilterConfigSet</u></a>	Set Filter configuration
<a href="#"><u>mpiFilterFlashConfigGet</u></a>	Get flash configuration for Filter
<a href="#"><u>mpiFilterFlashConfigSet</u></a>	Set flash configuration for Filter
<a href="#"><u>mpiFilterGainGet</u></a>	Get gain coefficients
<a href="#"><u>mpiFilterGainSet</u></a>	Set current gain index
<a href="#"><u>mpiFilterGainIndexGet</u></a>	Get current gain index
<a href="#"><u>mpiFilterGainIndexSet</u></a>	Set current gain index

### Memory Methods

<a href="#"><u>mpiFilterMemory</u></a>	Get address to Filter memory
<a href="#"><u>mpiFilterMemoryGet</u></a>	Copy data from Filter memory to application memory
<a href="#"><u>mpiFilterMemorySet</u></a>	Copy data from application memory to Filter memory

### Relational Methods

<a href="#"><u>mpiFilterAxisMapGet</u></a>	Get object map of axes associated with Filter
<a href="#"><u>mpiFilterAxisMapSet</u></a>	Set axes associated with Filter
<a href="#"><u>mpiFilterControl</u></a>	Return handle of Control that is associated with Filter

<a href="#"><u>mpiFilterMotorMapGet</u></a>	Get object map of Motors associated with Filter
<a href="#"><u>mpiFilterMotorMapSet</u></a>	Set Motors to be associated with Filter
<a href="#"><u>mpiFilterNumber</u></a>	Get index of Filter (for Control list)

## Action Methods

<a href="#"><u>mpiFilterIntergratorReset</u></a>	Reset the integrators of filter.
--	----------------------------------

## Postfilter Methods

<a href="#"><u>meiFilterPostfilterGet</u></a>	Reads postfilter information.
<a href="#"><u>meiFilterPostfilterSet</u></a>	Writes postfilter information.
<a href="#"><u>meiFilterPostfilterSectionGet</u></a>	Reads postfilter section information.
<a href="#"><u>meiFilterPostfilterSectionSet</u></a>	Writes postfilter section information.

## Data Types

[MPIFilterCoeff](#)  
[MPIFilterConfig](#) / [MEIFilterConfig](#)  
[MEIFilterForm](#)  
[MPIFilterGain](#)  
[MEIFilterGainIndex](#)  
[MEIFilterGainPID](#)  
[MEIFilterGainPIDCoeff](#)  
[MEIFilterGainPIV](#)  
[MEIFilterGainPIVCoeff](#)  
[MEIFilterGainTypePID](#)  
[MEIFilterGainTypePIV](#)  
[MPIFilterMessage](#)  
[MEIFilterType](#)  
[MEIPostfilterSection](#)

## Constants

[MPIFilterCoeffCOUNT\\_MAX](#)  
[MPIFilterGainCOUNT\\_MAX](#)  
[MEIMaxBiQuadSections](#)

# mpiFilterCreate

## Declaration

```
MPIFilter mpiFilterCreate(MPIControl control,  
                        long number)
```

Required Header: stdmpi.h

## Description

**mpiFilterCreate** creates a Filter object associated with a filter (*number*), that is located on a motion controller (*control*). FilterCreate is the equivalent of a C++ constructor.

### Return Values

<b>handle</b>	to an Filter object
<b>MPIHandleVOID</b>	if the Filter object could not be created

## See Also

[mpiFilterDelete](#) | [mpiFilterValidate](#)

# mpiFilterDelete

## Declaration

```
long mpiFilterDelete(MPIFilter filter)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterDelete** deletes a Filter object and invalidates its handle (*filter*). FilterDelete is the equivalent of a C++ destructor.

### Return Values

**MPIMessageOK**

if *FilterDelete* successfully deletes a Filter object and invalidates its handle

## See Also

[mpiFilterCreate](#) | [mpiFilterValidate](#)

# mpiFilterValidate

## Declaration

```
long mpiFilterValidate(MPIFilter filter)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterValidate** validates the Filter object and its handle (*filter*).

### Return Values

MPIMessageOK	if Filter is a handle to a valid object.
--------------	--

## See Also

[mpiFilterCreate](#) | [mpiFilterDelete](#)

# mpiFilterConfigGet

## Declaration

```
long mpiFilterConfigGet(MPIFilter filter,
                       MPIFilterConfig *config,
                       void *external)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterConfigGet** gets a Filter's (*filter*) configuration and writes it into the structure pointed to by *config*, and also writes it into the implementation-specific structure pointed to by *external* (if *external* is not NULL).

The Filter's configuration information in *external* is in addition to the Filter's configuration information in *config*, i.e, the Filter's configuration information in *config* and in *external* is not the same information. Note that *config* or *external* can be NULL (but not both NULL).

## Remarks

*external* either points to a structure of type **MEIFilterConfig{}** or is NULL.

### Return Values

**MPIMessageOK**

if *FilterConfigGet* successfully writes the Filter's configuration to the structure(s)

## See Also

[mpiFilterConfigSet](#) | [MEIFilterConfig](#)

# mpiFilterConfigSet

## Declaration

```
long mpiFilterConfigSet(MPIFilter filter,
                       MPIFilterConfig *config,
                       void *external)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterConfigSet** sets a Filter's (*filter*) configuration using data from the structure pointed to by *config*, and from the implementation-specific structure pointed to by *external* (if *external* is not NULL).

The Filter's configuration information in *external* is in addition to the Filter's configuration information in *config*, i.e, the Filter's configuration information in *config* and in *external* is not the same information. Note that *config* or *external* can be NULL (but not both NULL).

## Remarks

*external* either points to a structure of type `MEIFilterConfig{}` or is NULL.

### Return Values

**MPIMessageOK**

if *FilterConfigSet* successfully sets the Filter's configuration using data from the structure(s)

## See Also

[mpiFilterConfigGet](#) | [MEIFilterConfig](#)

# mpiFilterFlashConfigGet

## Declaration

```
long mpiFilterFlashConfigGet(MPIFilter filter,
                             void *flash,
                             MPIFilterConfig *config,
                             void *external)
```

Required Header: stdmpi.h

## Description

**mpiFilterFlashConfigGet** gets a Filter's (*filter*) flash configuration and writes it into the structure pointed to by *config*, and also writes it into the implementation-specific structure pointed to by *external* (if *external* is not NULL).

The Filter's flash configuration information in *external* is in addition to the Filter's flash configuration information in *config*, i.e., the flash configuration information in *config* and in *external* is not the same information. Note that *config* or *external* can be NULL (but not both NULL).

## Remarks

*external* either points to a structure of type **MEIFilterConfig{}** or is NULL.

### Return Values

#### MPIMessageOK

if *FilterFlashConfigGet* successfully writes the Filter's flash configuration to the structure(s)  
*flash* is either an MEIFlash handle or MPIHandleVOID. If *flash* is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

## See Also

[MEIFlash](#) | [mpiFilterFlashConfigSet](#) | [MEIFilterConfig](#)

# mpiFilterFlashConfigSet

## Declaration

```
long mpiFilterFlashConfigSet(MPIFilter filter,
                             void *flash,
                             MPIFilterConfig *config,
                             void *external)
```

Required Header: stdmpi.h

## Description

**mpiFilterFlashConfigSet** sets a Filter's (*filter*) flash configuration using data from the structure pointed to by *config*, and also using data from the implementation-specific structure pointed to by *external* (if *external* is not NULL).

The Filter's flash configuration information in *external* is in addition to the Filter's flash configuration information in *config*, i.e., the flash configuration information in *config* and in *external* is not the same information. Note that *config* or *external* can be NULL (but not both NULL).

## Remarks

*external* either points to a structure of type **MEIFilterConfig{}** or is NULL.

### Return Values

#### MPIMessageOK

if *FilterFlashConfigSet* successfully sets the Filter's flash configuration using data from the structure(s)  
*flash* is either an MEIFlash handle or MPIHandleVOID. If *flash* is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

## See Also

[MEIFlash](#) | [mpiFilterFlashConfigGet](#) | [MEIFilterConfig](#)

# mpiFilterGainGet

## Declaration

```
long mpiFilterGainGet(MPIFilter    filter,
                     long         gainIndex,
                     MPIFilterGain *gain)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterGainGet** gets the gain coefficients of a Filter (*filter*, for the gain index specified by *gainIndex*) and writes them into the structure pointed to by *gain*.

### Return Values

<b>MPIMessageOK</b>	if <i>FilterGainGet</i> successfully writes the gain coefficients to the structure
---------------------	--

## Sample Code

```
/* Sets reasonable tuning parameters for a Trust TA9000 test stand */
void setPIDs(MPIFilter filter)
{
    MPIFilterGain gain;
    long returnValue;

    returnValue = mpiFilterGainGet(filter, 0, &gain);
    msgCHECK(returnValue);

    gain.coeff[MEIFilterGainPIDCoeffGAIN_PROPORTIONAL].f = (float)100;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_INTEGRAL].f = (float)0.2;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_DERIVATIVE].f = (float)1000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_POSITION].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY].f = (float)45;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION].f = (float)101000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION].f = (float)450;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_REST].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffDRATE].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMIT].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITLOW].f = (float)-32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_OFFSET].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_POSITIONFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_FILTERFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_VELOCITYFFT].f = (float)0;

    returnValue = mpiFilterGainSet(filter, 0, &gain);
}
```

```
    msgCHECK(returnValue);  
}
```

## See Also

[mpiFilterGainSet](#)

# mpiFilterGainSet

## Declaration

```
long mpiFilterGainSet(MPIFilter    filter,
                    long          gainIndex,
                    MPIFilterGain *gain)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterGainSet** sets the gain coefficients of a Filter (*filter*, for the gain index specified by *gainIndex*) using data from the structure pointed to by *gain*.

### Return Values

**MPIMessageOK**

if *FilterGainSet* successfully sets the gain coefficients of a Filter using data from the structure

## Sample Code

```
/* Sets reasonable tuning parameters for a Trust TA9000 test stand */
void setPIDs(MPIFilter filter)
{
    MPIFilterGain gain;
    long returnValue;

    returnValue = mpiFilterGainGet(filter, 0, &gain);
    msgCHECK(returnValue);

    gain.coeff[MEIFilterGainPIDCoeffGAIN_PROPORTIONAL].f = (float)100;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_INTEGRAL].f = (float)0.2;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_DERIVATIVE].f = (float)1000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_POSITION].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY].f = (float)45;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION].f = (float)101000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION].f = (float)450;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_REST].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffDRATE].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMIT].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITLOW].f = (float)-32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_OFFSET].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_POSITIONFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_FILTERFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_VELOCITYFFT].f = (float)0;
```

```
    returnValue = mpiFilterGainSet(filter, 0, &gain);  
    msgCHECK(returnValue);  
}
```

## See Also

[mpiFilterGainGet](#)

# mpiFilterGainIndexGet

## Declaration

```
long mpiFilterGainIndexGet(MPIFilter filter,
                           long      *gainIndex)
```

Required Header: stdmpi.h

## Description

**mpiFilterGainIndexGet** gets the current gain index of a Filter (*filter*) and writes it to the location pointed to by *gainIndex*. Reading the gain index tells you what gain table is being used currently.

If the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeMOTION_ONLY`, the gain index is automatically changed by the firmware as described at [MEIXmpSwitchType](#). When the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeNONE`, the gain index is controlled by the user.

Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV).

### Return Values

<b>MPIMessageOK</b>	if <i>FilterGainIndexGet</i> successfully writes the gain index to the location
---------------------	---

## See Also

[MPIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIFilterGainIndex](#) | [MEIXmpSwitchType](#) | [mpiFilterGainIndexSet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)

# mpiFilterGainIndexSet

## Declaration

```
long mpiFilterGainIndexSet(MPIFilter filter,
                           long gainIndex)
```

Required Header: stdmpi.h

## Description

**mpiFilterGainIndexSet** sets the current gain index of a Filter (*filter*) to *gainIndex*. Writing the gain index controls what gain table is currently being used.

If the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeMOTION_ONLY`, the gain index is changed automatically by the firmware as described at [MEIXmpSwitchType](#). Be aware that the filter can change the gain index in real-time, thereby overwriting your changes in this mode.

When the filter is in state `MEIXmpSwitchType MEIXmpSwitchTypeNONE`, the gain index is controlled by the user. This is the normal state when using `FilterGainIndexSet(...)`. Gain Scheduling is a feature that switches filter gains for the acceleration, deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV).

### Return Values

**MPIMessageOK**

if *FilterGainIndexSet* successfully sets the current gain index to *gainIndex*

## See Also

[MPIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIFilterGainIndex](#) | [MEIXmpSwitchType](#) | [mpiFilterGainIndexGet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)

# mpiFilterMemory

## Declaration

```
long mpiFilterMemory(MPIFilter filter,  
                    void **memory)
```

Required Header: stdmpi.h

## Description

**mpiFilterMemory** writes an address, which is used to access a Filter's (*filter*) memory to the contents of *memory*. This address, or an address calculated from it, can be passed as the src parameter to **MPIFilterMemoryGet(...)** and as the *dst* parameter to **MPIFilterMemorySet(...)**.

### Return Values

**MPIMessageOK**

if *FilterMemory* successfully writes the Filter's memory address to the contents of *memory*

## See Also

[mpiFilterMemoryGet](#) | [mpiFilterMemorySet](#)

# mpiFilterMemoryGet

## Declaration

```
long mpiFilterMemoryGet (MPIFilter filter,
                        void *dst,
                        void *src,
                        long count)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterMemoryGet** copies *count* bytes of a Filter's (*filter*) memory (starting at address *src*) and writes them into application memory (starting at address *dst*).

### Return Values

<b>MPIMessageOK</b>	if <i>FilterMemoryGet</i> successfully copies data from Filter memory to application memory
---------------------	---

## See Also

[mpiFilterMemorySet](#) | [mpiFilterMemory](#)

# mpiFilterMemorySet

## Declaration

```
long mpiFilterMemorySet (MPIFilter filter ,  
                        void      *dst ,  
                        void      *src ,  
                        long      count )
```

**Required Header:** stdmpi.h

## Description

**mpiFilterMemorySet** copies *count* bytes of application memory (starting at address *src*) and writes them into a Filter's (*filter*) memory (starting at address *dst*).

### Return Values

<b>MPIMessageOK</b>	if <i>FilterMemorySet</i> successfully copies data from application memory to Filter memory
---------------------	---

## See Also

[mpiFilterMemorySet](#) | [mpiFilterMemory](#)

# mpiFilterAxisMapGet

## Declaration

```
long mpiFilterAxisMapGet(MPIFilter filter,  
                        MPIObjectMap *axisMap)
```

Required Header: stdmpi.h

## Description

**mpiFilterAxisMapGet** gets the object map of the Axes that are associated with a Filter (*filter*), and writes it into the structure pointed to by *axisMap*.

### Return Values

**MPIMessageOK**

if *FilterAxisMapGet* successfully writes the object map of Axes to the structure

## See Also

[mpiFilterAxisMapSet](#)

# mpiFilterAxisMapSet

## Declaration

```
long mpiFilterAxisMapSet(MPIFilter filter,  
                        MPIObjectMap axisMap)
```

Required Header: stdmpi.h

## Description

**mpiFilterAxisMapSet** sets the Axes associated with a Filter (*filter*), using data from the object map specified by *axisMap*.

### Return Values

<b>MPIMessageOK</b>	if <i>FilterAxisMapSet</i> successfully sets the Axes using the object map
---------------------	--

## See Also

[mpiFilterAxisMapGet](#)

# mpiFilterControl

## Declaration

```
MPIControl mpiFilterControl(MPIFilter filter)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterControl** returns a handle to the motion controller (Control object) associated with the specified Filter object (*filter*).

### Return Values

<b>handle</b>	to a Control object that a Filter object is associated with
<b>MPIHandleVOID</b>	if the Filter object is invalid

## See Also

[mpiFilterConfigGet](#) | [MEIFilterConfig](#)

# mpiFilterMotorMapGet

## Declaration

```
long mpiFilterMotorMapGet(MPIFilter filter,  
                          MPIObjectMap *motorMap)
```

Required Header: stdmpi.h

## Description

**mpiFilterMotorMapGet** gets the object map of the Motors associated with the Filter (*filter*), and writes it into the structure pointed to by *motorMap*.

### Return Values

**MPIMessageOK**

if *FilterMotorMapGet* successfully writes the object map of the Motors to the structure

## See Also

[mpiFilterMotorMapSet](#)

# mpiFilterMotorMapSet

## Declaration

```
long mpiFilterMotorMapSet(MPIFilter filter,  
                          MPIObjectMap motorMap)
```

Required Header: stdmpi.h

## Description

**mpiFilterMotorMapSet** sets the Motors associated with the Filter (*filter*) using data from the object map specified by *motorMap*.

### Return Values

**MPIMessageOK**

if *FilterMotorMapGet* successfully sets the Motors using data from the object map

## See Also

[mpiFilterMotorMapGet](#)

# mpiFilterNumber

## Declaration

```
long mpiFilterNumber(MPIFilter filter,  
                    long *number)
```

Required Header: stdmpi.h

## Description

For a motion controller that *filter* is associated with, **mpiFilterNumber** writes the index of *filter* to the contents of *number*.

### Return Values

**MPIMessageOK**

if *FilterNumber* successfully writes the index of a Filter to the contents of **number**

## See Also

# mpiFilterIntegratorReset

## Declaration

```
long mpiFilterIntegratorReset(MPIFilter filter)
```

**Required Header:** stdmpi.h

## Description

**mpiFilterIntegratorReset** resets the integrators of filter.

### Return Values

<b>MPIMessageOK</b>	if <i>mpiFilterIntegratorReset</i> successfully clears the integrators of <i>filter</i> .
<b>MPIFilterMessageINVALID_ALGORITHM</b>	if the <i>filter's</i> current algorithm does not use integrators.

## Sample Code

```
/* Enable the amplifier for every motor attached to a motion supervisor */
void motionAmpEnable(MPIMotion motion)
{
    MPIControl          control;
    MPIAxis             axis;
    MPIMotor            motor;
    MPIFilter           filter;
    MPIObjectMap        map;
    MPIObjectMap        motionMotorMap;
    long                motorIndex;
    long                filterIndex;
    long                returnValue;
    double              position;
    long                enableState;

    /* Get the controller handle */
    control = mpiMotionControl(motion);

    for (axis = mpiMotionAxisFirst(motion);
         axis != MPIHandleVOID;
         axis = mpiMotionAxisNext(motion, axis)) {

        /* Get the object map for the motors */
        returnValue = mpiAxisMotorMapGet(axis, &map);
        msgCHECK(returnValue);

        /* Add map to motionMotorMap */
        motionMotorMap |= map;
    }

    /* For every motor ... */
    for (motorIndex = 0; motorIndex < MEIXmpMAX_Motors; motorIndex++) {

        if (mpiObjectMapBitGET(motionMotorMap, motorIndex)) {

            /* Create motor handle */
            motor = mpiMotorCreate(control, motorIndex);
        }
    }
}
```

```

msgCHECK(mpiMotorValidate(motor));

/* Get the state of the amplifier */
returnValue = mpiMotorAmpEnableGet(motor, &enableState);
msgCHECK(returnValue);

/* If the amplifier is disabled ... */
if (enableState == FALSE) {

    /* For every axis */
    for (axis = mpiMotionAxisFirst(motion);
        axis != MPIHandleVOID;
        axis = mpiMotionAxisNext(motion, axis)) {

        /* Get the object map for the motors */
        returnValue = mpiAxisMotorMapGet(axis, &map);
        msgCHECK(returnValue);

        /* If axis is attached to motor ... */
        if (mpiObjectMapBitGET(map, motorIndex)) {

            /* Get the actual position of the axis */
            returnValue = mpiAxisActualPositionGet(axis,
&position);

            msgCHECK(returnValue);

            /* Set command position equal to actual position */
            returnValue = mpiAxisCommandPositionSet(axis,
position);

            msgCHECK(returnValue);

        }

    }

    /* Get the object map for the filters */
    returnValue = mpiMotorFilterMapGet(motor, &map);
    msgCHECK(returnValue);

    /* For every filter ... */
    for (filterIndex = 0;
        filterIndex < MEIXmpMAX_Filters;
        filterIndex++) {

        if (mpiObjectMapBitGET(map, filterIndex)) {

            /* Create filter handle */
            filter = mpiFilterCreate(control, filterIndex);
            msgCHECK(mpiFilterValidate(filter));

            /* Reset integrator */
            returnValue = mpiFilterIntegratorReset(filter);
            msgCHECK(returnValue);

            /* Delete filter handle */
            returnValue = mpiFilterDelete(filter);
            msgCHECK(returnValue);

        }

    }

    /* Enable the amplifier */
    returnValue = mpiMotorAmpEnableSet(motor, TRUE);
    msgCHECK(returnValue);

}

/* Delete motor handle */
returnValue = mpiMotorDelete(motor);
msgCHECK(returnValue);

}
}

```

}

## Troubleshooting

If an axis is not in an error state and the filter associated with that axis' motor has a non-zero integration term, then it is very likely that the integrator has built up a substantial integral term. Enabling the motor's amplifier when this has happened could cause the motor to jump with enormous force. Use **mpiFilterIntegratorReset** to reset the integrator before enabling the motor's amplifier to prevent this kind of jump.

Another condition that can cause the motor to jump upon enabling its amplifier is that the command position of the axis is not equal to the actual position of the axis. To prevent this situation, one should use **mpiAxisActualPositionGet** and **mpiAxisCommandPositionSet**. Please refer to this functions for a more in depth discussion.

## See Also

[MPIFilter](#) | [MEIFilterConfig](#) | [MEIFilterGainPID](#) | [MEIFilterGainPIV](#)  
[mpiAxisActualPositionGet](#) | [mpiAxisCommandPositionSet](#)

# mpiFilterPosfilterGet

## Declaration

```
long meiFilterPostfilterGet(MPIFilter          filter,
                           long                *sectionCount,
                           MEIPostfilterSection *sections);
```

**Required Header:** stdmei.h

## Description

**meiFilterPostfilterGet** reads an MPIFilter object's postfilter configuration. It writes to **sectionCount** the number of sections within a postfilter if **sectionCount** is not NULL. It also writes to **sections** the current array of **filter**'s postfilter sections if sections is not NULL.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

<b>filter</b>	the handle of the MPIFilter object whose postfilter configuration is to be read.
<b>*sectionCount</b>	the data location where the postfilter's current section count will be written.
<b>*sections</b>	the data location where the postfilter's current section configuration data will be written.

## Return Values

<b>MPIMessageOK</b>	if <i>PostfilterGet</i> successfully retrieves postfilter information.
<b>MPIFilterMessageCONVERSION_DIV_BY_0</b>	Returned when meiFilterPostfilterGet(...) cannot convert digital coefficients to analog coefficients. When this error occurs, the offending section(s) will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.
<b>MPIFilterMessageINVALID_FILTER_FORM</b>	Returned when meiFilterPostfilterGet(...) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

## Sample Code

```

/* Count the number of resonator sections in a MPIFilter object's postfilter.
Sample usage:

returnValue =
    filterResonatorCount(filter, &resonatorCount);
*/

long filterResonatorCount(MPIFilter filter, long* count)
{
    MPIFilterConfig config;
    MEIPostfilterSection sections[MEIMaxBiQuadSections];
    long sectionCount, index;
    long returnValue = (count==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            meiFilterPostfilterGet(filter, &sectionCount, sections);

        if (returnValue == MPIMessageOK)
        {
            for (*count=0, index=0; index < sectionCount; ++index)
            {
                if (sections[index].type == MEIFilterTypeRESONATOR) ++(*count);
            }
        }
    }
    return returnValue;
}

```

## See Also

[MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meFilterPostfilterSectionGet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#)

# meiFilterPostfilterSet

## Declaration

```
long meiFilterPostfilterSet(MPIFilter          filter,
                             long             *sectionsCount,
                             MEIPostfilterSection *sections);
```

**Required Header:** stdmei.h

## Description

**meiFilterPostfilterSet** sets the number of postfilter sections within an MPIFilter object and configures each postfilter section as well. If **numberOfSections** equals zero, then **sections** can be NULL and the postfilter will be disabled.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

<b>filter</b>	the handle of the MPIFilter object whose postfilter sections will be configured.
<b>*sectionsCount</b>	the number of postfilter sections to set in the <i>filter</i> object.
<b>*sections</b>	a pointer to an array of MEIPostfilterSection data structures to be set in <i>filter</i> .

### Return Values

<b>MPIMessageOK</b>	if <i>PostfilterSet</i> successfully writes postfilter information.
---------------------	---

## Sample Code

```
/* Set a 4th order low-pass post-filter by using two 2nd order low-pass sections.
Sample usage:

returnValue =
    fourthOrderLowPass(filter, 300 /* Hz */);
*/
long filterFouthOrderLowpass(MPIFilter filter, long breakPointFrequency)
{
    MPIFilterConfig config;
    MEIPostfilterSection section[MEIMaxBiQuadSections];
    long returnValue;

    section[0].type = MEIFilterTypeLOW_PASS;
    section[0].form = MEIFilterFormINT_BIQUAD;
```

```
    section[0].data.lowPass.breakpoint = breakPointFrequency;
    section[1] = section[0]; /* copy first section */

    returnValue =
        meiFilterPostfilterSet(filter, 2, section);

    return returnValue;
}
```

## See Also

[MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meFilterPostfilterSectionSet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#)

# meiFilterPostfilterSectionGet

## Declaration

```
long meiFilterPostfilterSectionGet(MPIFilter filter,
                                   long      sectionNumber,
                                   MEIPostfilterSection *section);
```

**Required Header:** stdmei.h

## Description

**meiFilterPostfilterSectionGet** reads the configuration of a single section of an MPIFilter object's postfilter. It writes to ***\*section*** the configuration of ***filter's*** postfilter ***sectionNumber***<sup>th</sup> section.

The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

<b>filter</b>	the handle of the MPIFilter object whose postfilter section configuration is to be read.
<b>sectionNumber</b>	the index of the postfilter section whose configuration is to be read.
<b>section</b>	the data location where the postfilter's current section configuration data will be written.

### Return Values

<b>MPIMessageOK</b>	if <i>PostfilterSectionGet</i> successfully reads postfilter section information.
<b>MPIFilterMessageCONVERSION_DIV_BY_0</b>	Returned when meiFilterPostfilterSectionGet(...) cannot convert digital coefficients to analog coefficients. When this error occurs, the section will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.
<b>MPIFilterMessageSECTION_NOT_ENABLED</b>	Returned when no postfilter sections are enabled.
<b>MPIFilterMessageINVALID_FILTER_FORM</b>	Returned when meiFilterPostfilterSectionGet(...) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

## Sample Code

```
/* Test a section of a MPIFilter object's postfilter to see if it is a notch type.
   Sample usage:

   returnValue =
       isSectionTypeNotch(filter, 0, &isNotch);
*/
long isSectionTypeNotch(MPIFilter filter, long sectionIndex, long* isNotch)
{
    MPIFilterConfig config;
    MEIPostfilterSection section;
    long returnValue = (isNotch==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            meiFilterPostfilterSectionGet(filter, sectionIndex, &section);
        if (returnValue == MPIMessageOK)
        {
            *isNotch = (section.type == MEIFilterTypeNOTCH) ? TRUE : FALSE;
        }
    }

    return returnValue;
}
```

## See Also

[MEIPostfilterSection](#) | [meiFilterPostfilterGet](#) | [meFilterPostfilterSectionSet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#)

# meiFilterPostfilterSectionSet

## Declaration

```
long meiFilterPostfilterSectionSet(MPIFilter filter,
                                   long      sectionNumber,
                                   MEIPostfilterSection *section);
```

**Required Header:** stdmei.h

## Description

**meiFilterPostfilterSectionSet** sets the configuration of a single section of an MPIFilter object's postfilter. It sets *filter*'s postfilter *sectionNumber*<sup>th</sup> section to the configuration specified in *\*section*. If the postfilter type is IIR, then this method is essentially equivalent to meiFilterPostfilterSet().

The MPI calculates the post filter coefficients taking into consideration the sample rate of the controller at that time. If you change the change the sample rate of the controller, you will need to recalculate your post filters. This can be done for all filters specified in Hertz by setting the filters again using the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

<b>filter</b>	the handle of the MPIFilter object whose postfilter section configuration is to be set.
<b>sectionNumber</b>	the index of the postfilter section whose configuration is to be set.
<b>*section</b>	the data location of the section configuration to copy to the controller.

### Return Values

<b>MPIMessageOK</b>	if <i>PostfilterSectionSet</i> successfully writes postfilter section information.
---------------------	--

## Sample Code

```
/* Set a section of a MPIFilter object's postfilter to a unity gain filter type.
Sample usage:

returnValue =
    setSectionTypeUnityGain(filter, 3);
*/
long setSectionTypeUnityGain(MPIFilter filter, long sectionIndex)
{
    MPIFilterConfig config;
    MEIPostfilterSection section;
    long returnValue;
```

```
    section.type = MEIFilterTypeUNITY_GAIN;
    section.form = MEIFilterFormBIQUAD;

    returnValue =
        meiFilterPostfilterSectionSet(filter, sectionIndex, sion);

    return returnValue;
}
```

## See Also

[MEIPostfilterSection](#) | [meiFilterPostfilterSet](#) | [meFilterPostfilterSectionGet](#) | [MEIMaxBiQuadSections](#) | [Post Filter Theory](#)

# MPIFilterCoeff

## Definition

```
typedef union {  
    float    f;  
    long     l;  
} MPIFilterCoeff;
```

## Description

**MPIEventStatus** holds information about a particular event that was generated by the XMP.

<b>f</b>	float coefficient
<b>l</b>	long coefficient

## See Also

[MPIFilterCoeffCOUNT\\_MAX](#) | [MEIFilterGainPIDCoeff](#) | [MEIFilterGainPIVCoeff](#)

# MPIFilterConfig / MEIFilterConfig

## Definition: MPIFilterConfig

```
typedef struct MPIFilterConfig {
    long          gainIndex;
    MPIFilterGain gain[MPIFilterGainCOUNT_MAX];

    MPIObjectMap axisMap;
    MPIObjectMap motorMap;
} MPIFilterConfig;
```

## Description

<b>gainIndex</b>	Gain table index. Gain tables number 0 to MPIFilterGainCOUNT_MAX -1 (MPIFilterGainCOUNT_MAX = 5).
<b>gain</b>	See <a href="#">MPIObjectMap</a>
<b>axisMap</b>	See <a href="#">MPIObjectMap</a>
<b>motorMap</b>	See <a href="#">MPIObjectMap</a>

## Definition: MEIFilterConfig

```
typedef struct MEIFilterConfig {
    MEIXmpAlgorithm      Algorithm;

    MEIXmpAxisInput      Axis[MEIXmpFilterAxisInputs];

    long                 *VelPositionPtr;

    MEIXmpSwitchType     GainSwitchType;
    float                GainDelay;
    long                 GainWindow;
    MEIXmpSwitchType     PPISwitchType;
    MEIXmpPPIMode        PPIMode;
    float                PPIDelay;
    long                 PPIWindow;
    MEIXmpIntResetConfig ResetIntegratorConfig;
    float                ResetIntegratorDelay;

    MEIXmpFilterForm     PostFilterForm;
    MEIXmpPostFilter     PostFilter;
} MEIFilterConfig;
```

## Description

**MEIFilterConfig** contains configuration information specific to a controller. With the exception of the Algorithm element, MEIFilterConfig contains configuration information that are more intuitively accessed by other means (Postfilter parameter) or information for advanced setups and custom controller firmware.

<b>Algorithm</b>	This value defines the algorithm that the filter is executing every servo cycle. The most common values are:  MEIXmpAlgorithmPID      PID algorithm MEIXmpAlgorithmPIV      PIV algorithm MEIXmpAlgorithmNONE    No control algorithm
<b>Axis</b> [MEIXmpFilterAxisInputs]	This array defines the axis (pointer to the axis) and coefficient for the position input into the filter. The input to the filter is the position error of the axis, which is multiplied by the coefficient defined by the Axis array.
<b>VelPositionPtr</b>	Velocity position pointer to an encoder input for algorithms that require a velocity encoder position input (such as the PIV algorithm).
<b>AuxInput</b> [MEIXmpFilterAuxInputs]	This array is a place holder for additional filter inputs from analog sources. <b>This is currently not supported and is reserved for future use.</b>
<b>GainSwitchType</b>	Value to define the gain table switch type. <b>Not implemented in standard firmware.</b>
<b>GainDelay</b>	Custom Delay <b>Not implemented in standard firmware.</b>
<b>GainWindow</b>	Custom Delay <b>Not implemented in standard firmware.</b>
<b>PPISwitchType</b>	Value to define the gain switch type for PPI mode. <b>Not implemented in standard firmware.</b>
<b>PPIMode</b>	Value to define the PPI switch mode. <b>Not implemented in standard firmware.</b>
<b>PPIDelay</b>	Custom Delay <b>Not implemented in standard firmware.</b>
<b>PPIWindow</b>	Custom Window <b>Not implemented in standard firmware.</b>
<b>ResetIntegratorConfig</b>	Value to define the integrator's reset configuration. <b>Not supported in standard firmware.</b>
<b>ResetIntegratorDelay</b>	Value to define the integrator's reset delay. <b>Not supported in standard firmware.</b>

<b>PostFilterForm</b>	<p>This value defines the form for postfilters when they are configured using <code>mpiFilterConfigGet/Set()</code>.</p> <p>Supported values are:</p> <ul style="list-style-type: none"> <li>• <b>MEIXmpFilterFormIIR</b>, IIR Filter</li> <li>• <b>MEIXmpFilterFormBIQ</b>, Bi-Quad Filter</li> <li>• <b>MEIXmpFilterFormSS_BIQ</b>, State Space form of Bi-Quad Filter</li> <li>• <b>MEIXmpFilterFormINT_BIQ</b>, Integer (64-bit) Bi-Quad Filter</li> <li>• <b>MEIXmpFilterFormINT_SS_BIQ</b>, Integer State Space form of Bi-Quad Filter</li> </ul> <p>Though the postfilter may be configured through this parameter, it is strongly recommended that users use the <code>meiFilterPostfilter.()</code> methods instead for a more intuitive and user-friendly interface.</p>
<b>PostFilter</b>	<p>This array defines the configuration for the filter's postfilter (the type, the length and values for the post filter coefficients). Though the postfilter may be configured though this parameter, it is strongly recommended that users use the <code>meiFilterPostfilter.()</code> methods instead for a more intuitive interface.</p> <p>Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.</p>

## Sample Code

```

/* Test whether an MPIFilter object's control loop algorithm is PID.
Sample usage:

returnValue =
    isAlgorithmPid(filter, &isPid);
*/

long isAlgorithmPid(MPIFilter filter, long* isPid)
{
    MEIFilterConfig xmpConfig;
    long returnValue = (isPid==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue =
            mpiFilterConfigGet(filter, NULL, &xmpConfig);
        if (returnValue == MPIMessageOK)
        {
            *isPid = (xmpConfig.Algorithm == MEIXmpAlgorithmPID) ? TRUE : FALSE;
        }
    }
}

```

```
        }  
    }  
  
    return returnValue;  
}
```

## See Also

[mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [meiFilterPostfilterGet](#) |  
[meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#)

# MEIFilterForm

## Definition

```
typedef enum{
    MEIFilterFormIIR,
    MEIFilterFormBIQUAD,
    MEIFilterFormSS_BIQUAD,
    MEIFilterFormINT_BIQUAD,
    MEIFilterFormINT_SS_BIQUAD,
} MEIFilterForm;
```

## Description

**MEIFilterForm** describes the form that a digital filter takes on the controller. Please note that the equations listed below use the coefficients loaded onto the controller, not necessarily the coefficients used by the MPI. A user may specify a low pass filter with only a single parameter (the breakpoint) and request that the form of the filter be a space-state biquad form on the controller.

Digital filtering on the XMP is accomplished through 32-bit words. This equates to the use of single precision floating point numbers - a 24-bit mantissa or about 7 decimal places of accuracy. This lack of precision can cause errors in the filtering process normally appearing as DC gain shifts or limit cycling, this especially true when the filter requires more than one section, a 6th order low pass filter would be one example. Filter forms using integer math can provide more internal precision for coefficients and internal registers but at the cost of less dynamic range. Filter forms using integer math take more processing time for the controller and can potentially limit the maximum sample rate of the controller.

The state-space (SS) filter forms allow the scaling of the input and the output, whereas the non-state-space forms only allow output scaling. This helps to prevent the loss of precision of the internal registers while still maintaining a very large dynamic range. Filter forms using state-space forms take more processing time for the controller and can potentially limit the maximum sample rate of the controller. However, a non-integer state-space filter form takes less processing power than an integer non-state-space filter form.

<b>MEIFilterFormIIR</b>	<i>Deprecated</i> . Cascaded biquad sections offer better precision and better calculation performance.
<b>MEIFilterFormBIQUAD</b>	<p>Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a single precision floating point canonical form. The biquad filter is defined by the following discrete transfer function:</p> <p>The XMP's representation of this filter is:</p> <p>w0: Intermediate result  u(k): filter input  a1, a2, b0, b1, and b2: discrete biquad coefficients  y(k):filter output  x1k and x2k: filter states</p>
<b>MEIFilterFormSS_BIQUAD</b>	<p>Second order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a single precision, floating point state space implementation. This filter applies input and output scaling to the canonical form. The XMP's state space representation of this filter is:</p> <p>u(k): filter input  d1, c1, c2, a2, a1,b1: discrete biquad coefficients  y(k):filter output  p1k and p2k: filter states</p>
<b>MEIFilterFormINT_BIQUAD</b>	<p>Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a fixed point canonical form state space implementation. This form is a fixed point implementation of the floating point form <b>MEIFilterFormBIQUAD</b>. See the definition of <b>MEIFilterFormBIQUAD</b> above for the defining equations for this filter.</p> <p>The input coefficients for this filter (b0, b1, b2, a1 and a2) should all be greater than -2, and less than 2. The coefficients are represented as 32 bit 2's complement, with <math>1=2^{30}</math>. The coefficient's numerical format is 1.29 (1 bit whole, 29 bits fractional), and the controller uses an 80 bit accumulator. Only the 32 bit result of the multiplication is output from each section.</p>

**MEIFilterFormINT\_SS\_BIQUAD**

Second Order digital filter form, for implementing low/high pass, notch, lead/lag and custom filters. The filter is a fixed point canonical form state space implementation. This form is a fixed point implementation of the floating point form `MEIFilterFormSS_BIQUAD`. See the definition of `MEIFilterFormSS_BIQUAD` above for the defining equations for this filter.

The input coefficients for this filter (d1, c1, c2, a2, a1 and b1) should all be greater than -2, and less than 2. The coefficients are represented as 32 bit 2's complement, with  $1=2^{30}$ . The coefficient's numerical format is 1.29 (1 bit whole, 29 bits fractional), and the controller uses an 80 bit accumulator. Only the 32 bit result of the multiplication is output from each section.

**See Also**

[MEIPostfilterSection](#)

# MPIFilterGain

## Definition

```
typedef struct MPIFilterGain {
    MPIFilterCoeff  coeff[MPIFilterCoeffCOUNT_MAX];
} MPIFilterGain;
```

## Description

<b>coeff</b>	see <a href="#">MPIFilterCoeff</a>
--------------	------------------------------------

## Sample Code

```
/* Sets reasonable tuning parameters for a Trust TA9000 test stand */
void setPIDs(MPIFilter filter)
{
    MPIFilterGain gain;
    long returnValue;

    returnValue = mpiFilterGainGet(filter, 0, &gain);
    msgCHECK(returnValue);

    gain.coeff[MEIFilterGainPIDCoeffGAIN_PROPORTIONAL].f = (float)100;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_INTEGRAL].f = (float)0.2;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_DERIVATIVE].f = (float)1000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_POSITION].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY].f = (float)45;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION].f = (float)101000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION].f = (float)450;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_REST].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffDRATE].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMIT].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITLOW].f = (float)-32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_OFFSET].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_POSITIONFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_FILTERFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_VELOCITYFFT].f = (float)0;

    returnValue = mpiFilterGainSet(filter, 0, &gain);
    msgCHECK(returnValue);
}
```

## See Also

[MPIFilterGainCOUNT\\_MAX](#) | [MEIFilterGainPIDCoeff](#) | [MEIFilterGainPIVCoeff](#)

# MEIFilterGainIndex

## Definition

```
typedef enum {

    /* Gain table index for normal firmware. */
    MEIFilterGainIndexNO_MOTION = MEIXmpGainNOT_MOVING,
    MEIFilterGainIndexACCEL      = MEIXmpGainACCEL,
    MEIFilterGainIndexDECEL     = MEIXmpGainDECEL,
    MEIFilterGainIndexVELOCITY  = MEIXmpGainCONSTANT_VEL,

    /* Gain table index for Custom 1 firmware. */
    MEIFilterGainIndexSTOPPING2 = MEIXmpGainSTOPPED2,
    MEIFilterGainIndexSTOPPING1 = MEIXmpGainSTOPPED1,
    MEIFilterGainIndexSETTLING  = MEIXmpGainSETTLING,
    MEIFilterGainIndexMOVING    = MEIXmpGainMOVING,
    MEIFilterGainIndexSTOPPING3 = MEIXmpGainSTOPPED3,

    /* Gain table index for Custom 5 firmware. */
    MEIFilterGainIndexMIN       = MEIXmpGainMIN,
    MEIFilterGainIndexMAX       = MEIXmpGainMAX,
    MEIFilterGainIndexNONE     = MEIXmpGainNONE,
    MEIFilterGainIndexSLOPE    = MEIXmpGainSLOPE,

    MEIFilterGainIndexLAST     = MEIXmpGainLAST,
    MEIFilterGainIndexALL      = MEIFilterGainIndexLAST,
                                /* used for gain get/set() */
    MEIFilterGainIndexFIRST    = MEIFilterGainIndexINVALID + 1,

    MEIFilterGainIndexDEFAULT  = MEIFilterGainIndexNO_MOTION,
} MEIFilterGainIndex;
```

## Description

**MEIFilterGainIndex** is an enumeration for the gain index used in gain scheduling.

In standard firmware, only

MEIFilterGainIndexNO\_MOTION,

MEIFilterGainIndexACCEL,

MEIFilterGainIndexDECEL, and

MEIFilterGainIndexVELOCITY

are used. The gain index that is currently used can be found with

[mpiFilterGainIndexGet\(...\)](#).

Gain Scheduling is a feature that switches filter gains for the acceleration,

deceleration, constant velocity, and idle states of motion. The post filters are not affected by gain scheduling. Standard algorithms are used with gain scheduling (PID, PIV). To change the gain scheduling type from NONE (uses only the gains in gain table index 0), use [MEIFilterConfig](#). GainSwitchType is set with [mpiFilterConfigSet\(...\)](#).

When setting filter gain parameters using [mpiFilterGainGet\(...\)](#) and [mpiFilterGainSet\(...\)](#), use the gain index value to write to a gain index of your choosing.

<b>MEIFilterGainIndexNO_MOTION</b>	No commanded motion. Trajectory parameters Velocity, Acceleration, and Jerk equal zero.
<b>MEIFilterGainIndexACCEL</b>	Acceleration portion of the commanded move.
<b>MEIFilterGainIndexDECEL</b>	Deceleration portion of the commanded move.
<b>MEIFilterGainIndexVELOCITY</b>	Constant velocity portion of the commanded move. Gain switching is configured by setting the GainSwitchType, GainDelay, and GainWindow in the MEIFilterConfig{...} structure and calling mpiFilterConfigGet/Set(...). The GainSwitchType has the following options:

## See Also

[MEIFilterConfig](#) | [mpiFilterConfigGet](#) | [mpiFilterConfigSet](#) | [MEIXmpSwitchType](#) | [mpiFilterGainIndexSet](#) | [mpiFilterGainIndexGet](#) | [mpiFilterGainGet](#) | [mpiFilterGainSet](#)

# MEIFilterGainPID

## Definition

```
typedef struct MEIFilterGainPID {
    struct {
        float    proportional;    /* Kp */
        float    integral;        /* Ki */
        float    derivative;      /* Kd */
    } gain;
    struct {
        float    position;        /* Kpff */
        float    velocity;        /* Kvff */
        float    acceleration;    /* Kaff */
        float    friction;        /* Kfff */
    } feedForward;
    struct {
        float    moving;          /* MovingIMax */
        float    rest;            /* RestIMax */
    } integrationMax;
    long    dRate;                /* DRate */
    struct {
        float    limit;           /* OutputLimit */
        float    limitHigh;       /* OutputLimitHigh */
        float    limitLow;        /* OutputLimitLow */
        float    offset;          /* OutputOffset */
    } output;
    struct {
        float    positionFFT;     /* Ka0 */
        float    filterFFT;       /* Ka1 */
        float    velocityFFT;     /* Ka2 */
    } noise;
} MEIFilterGainPID;
```

## Description

**MEIFilterGainPID** is a structure that defines the filter coefficients for the PID filter algorithm.

## See Also

[High/Low Output Limits](#) section for special instructions regarding MEIFilterGainPID.  
[MEIFilterGainPIDCoeff](#)

# MEIFilterGainPIDCoeff

## Definition

```
typedef          enum {
    MEIFilterGainPIDCoeffINVALID = -1,

    MEIFilterGainPIDCoeffGAIN_PROPORTIONAL, /* Kp */
    MEIFilterGainPIDCoeffGAIN_INTEGRAL,     /* Ki */
    MEIFilterGainPIDCoeffGAIN_DERIVATIVE,   /* Kd */

    MEIFilterGainPIDCoeffFEEDFORWARD_POSITION, /* Kpff */
    MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY, /* Kvff */
    MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION, /* Kaff */
    MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION, /* Kfff */

    MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING, /* MovingIMax */
    MEIFilterGainPIDCoeffINTEGRATIONMAX_REST, /* RestIMax */

    MEIFilterGainPIDCoeffDRATE, /* DRate */

    MEIFilterGainPIDCoeffOUTPUT_LIMIT, /* OutputLimit */
    MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH, /* OutputLimitHigh */
    MEIFilterGainPIDCoeffOUTPUT_LIMITLOW, /* OutputLimitLow */
    MEIFilterGainPIDCoeffOUTPUT_OFFSET, /* OutputOffset */

    MEIFilterGainPIDCoeffNOISE_POSITIONFFT, /* Ka0 */
    MEIFilterGainPIDCoeffNOISE_FILTERFFT, /* Ka1 */
    MEIFilterGainPIDCoeffNOISE_VELOCITYFFT, /* Ka2 */
} MEIFilterGainPIDCoeff;
```

## Description

**MEIFilterGainPIDCoeff** is a structure of enums that defines the filter coefficients for the PID filter algorithm.

## Sample Code

```
/* Sets reasonable tuning parameters for a Trust TA9000 test stand */
void setPIDs(MPIFilter filter)
{
    MPIFilterGain gain;
    long returnValue;

    returnValue = mpiFilterGainGet(filter, 0, &gain);
    msgCHECK(returnValue);

    gain.coeff[MEIFilterGainPIDCoeffGAIN_PROPORTIONAL].f = (float)100;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_INTEGRAL].f = (float)0.2;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_DERIVATIVE].f = (float)1000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_POSITION].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY].f = (float)45;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION].f = (float)101000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION].f = (float)450;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_REST].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffDRATE].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMIT].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITLOW].f = (float)-32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_OFFSET].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_POSITIONFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_FILTERFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_VELOCITYFFT].f = (float)0;

    returnValue = mpiFilterGainSet(filter, 0, &gain);
    msgCHECK(returnValue);
}
```

## See Also

[MEIFilterGainPID](#)

# MEIFilterGainPIV

## Definition

```

typedef      struct MEIFilterGainPIV {
    struct {
        float    proportional;    /* Kpp */
        float    integral;        /* Kip */
    } gainPosition;
    struct {
        float    proportional;    /* Kpv */
    } gainVelocity1;
    struct {
        float    position;        /* Kpff */
        float    velocity;        /* Kvff */
        float    acceleration;    /* Kaff */
        float    friction;        /* Kfff */
    } feedForward;
    struct {
        float    moving;          /* MovingIMax */
        float    rest;            /* RestIMax */
    } integrationMax;
    struct {
        float    feedback;        /* Kdv */
    } gainVelocity2;
    struct {
        float    limit;           /* OutputLimit */
        float    limitHigh;       /* OutputLimitHigh */
        float    limitLow;        /* OutputLimitLow */
        float    offset;          /* OutputOffset */
    } output;
    struct {
        float    integral;        /* Kiv */
        float    integrationMax;   /* VintMax */
    } gainVelocity3;
    struct {
        float    positionFFT;     /* Ka0 */
        float    smoothing;       /* Ka1 */
        float    filterFFT;       /* Ka2 */
    } noise;
} MEIFilterGainPIV;

```

**Change History:** Modified in the 03.02.00

## Description

**MEIFilterGainPIV** is a structure that defines the filter coefficients for the PIV filter algorithm.

## See Also

[High/Low Output Limits](#) section for special instructions regarding MEIFilterGainPIV.  
[MEIFilterGainPIVCoeff](#)

# MEIFilterGainPIVCoeff

## Definition

```

typedef          enum {
    MEIFilterGainPIVCoeffINVALID = -1,

    MEIFilterGainPIVCoeffGAINPOSITION_PROPORTIONAL,      /* Kpp */
    MEIFilterGainPIVCoeffGAINPOSITION_INTEGRAL,          /* Kip */

    MEIFilterGainPIVCoeffGAINVELOCITY_PROPORTIONAL,      /* Kpv */

    MEIFilterGainPIVCoeffFEEDFORWARD_POSITION,           /* Kpff */
    MEIFilterGainPIVCoeffFEEDFORWARD_VELOCITY,           /* Kvff */
    MEIFilterGainPIVCoeffFEEDFORWARD_ACCELERATION,       /* Kaff */
    MEIFilterGainPIVCoeffFEEDFORWARD_FRICTION,           /* Kfff */

    MEIFilterGainPIVCoeffINTEGRATIONMAX_MOVING,           /* MovingIMax */
    MEIFilterGainPIVCoeffINTEGRATIONMAX_REST,             /* RestIMax */

    MEIFilterGainPIVCoeffGAINVELOCITY_FEEDBACK,          /* Kdv */

    MEIFilterGainPIVCoeffOUTPUT_LIMIT,                   /* OutputLimit */
    MEIFilterGainPIVCoeffOUTPUT_LIMITHIGH,               /* OutputLimitHigh */
    MEIFilterGainPIVCoeffOUTPUT_LIMITLOW,                /* OutputLimitLow */
    MEIFilterGainPIVCoeffOUTPUT_OFFSET,                  /* OutputOffset */

    MEIFilterGainPIVCoeffGAINVELOCITY_INTEGRAL,          /* Kiv */
    MEIFilterGainPIVCoeffGAINVELOCITY_INTEGRATIONMAX,    /* Vintmax */

    MEIFilterGainPIVCoeffNOISE_POSITIONFFT,              /* Ka0 */
    MEIFilterGainPIVCoeffSMOOTHINGFILTER_GAIN,           /* Ka1 */
    MEIFilterGainPIVCoeffNOISE_FILTERFFT,                /* Ka2 */
} MEIFilterGainPIVCoeff;

```

**Change History:** Modified in the 03.02.00

## Description

**MEIFilterGainPIVCoeff** is a structure of enums that defines the filter coefficients for the PIV filter algorithm.

## Sample Code

```
/* Sets reasonable tuning parameters for a Trust TA9000 test stand */
void setPIDs(MPIFilter filter)
{
    MPIFilterGain gain;
    long returnValue;

    returnValue = mpiFilterGainGet(filter, 0, &gain);
    msgCHECK(returnValue);

    gain.coeff[MEIFilterGainPIDCoeffGAIN_PROPORTIONAL].f = (float)100;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_INTEGRAL].f = (float)0.2;
    gain.coeff[MEIFilterGainPIDCoeffGAIN_DERIVATIVE].f = (float)1000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_POSITION].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_VELOCITY].f = (float)45;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_ACCELERATION].f = (float)101000;
    gain.coeff[MEIFilterGainPIDCoeffFEEDFORWARD_FRICTION].f = (float)450;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_MOVING].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffINTEGRATIONMAX_REST].f = (float)15000;
    gain.coeff[MEIFilterGainPIDCoeffDRATE].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMIT].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITHIGH].f = (float)32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_LIMITLOW].f = (float)-32767;
    gain.coeff[MEIFilterGainPIDCoeffOUTPUT_OFFSET].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_POSITIONFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_FILTERFFT].f = (float)0;
    gain.coeff[MEIFilterGainPIDCoeffNOISE_VELOCITYFFT].f = (float)0;

    returnValue = mpiFilterGainSet(filter, 0, &gain);
    msgCHECK(returnValue);
}
```

## See Also

[High/Low Output Limits](#) section for special instructions regarding MEIFilterGainPIV.  
[MEIFilterGainPIV](#)

# MEIFilterGainTypePID

## Definition

```
static MEIDataType MEIFilterGainTypePID[MPIFilterCoeffCOUNT_MAX] =
{
    MEIDataTypeFLOAT, /* Kp          */ /* /
    MEIDataTypeFLOAT, /* Ki          */ /* /
    MEIDataTypeFLOAT, /* Kd          */ /* /

    MEIDataTypeFLOAT, /* Kpff       */ /* /
    MEIDataTypeFLOAT, /* Kvff       */ /* /
    MEIDataTypeFLOAT, /* Kaff       */ /* /
    MEIDataTypeFLOAT, /* Kfff       */ /* /

    MEIDataTypeFLOAT, /* MovingIMax */ /* /
    MEIDataTypeFLOAT, /* RestIMax   */ /* /

    MEIDataTypeLONG,  /* DRate      */ /* /

    MEIDataTypeFLOAT, /* OutputLimit */ /* /
    MEIDataTypeFLOAT, /* OutputLimitHigh */ /* /
    MEIDataTypeFLOAT, /* OutputLimitLow */ /* /
    MEIDataTypeFLOAT, /* OutputOffset */ /* /
    MEIDataTypeFLOAT, /* Ka0        */ /* /
    MEIDataTypeFLOAT, /* Ka1        */ /* /
    MEIDataTypeFLOAT, /* Ka2        */ /* /
};
```

## Description

**MEIFilterGainTypePID** is a static array that describes the data type of the coefficients for the PID algorithm. Specifically, an element of **MEIFilterGainTypePID** describes which member of the union **MPIFilterCoeff** to access when using the data structure **MPIFilterCoeff**.

**MEIFilterGainTypePID** allows for a more simple design of general case utilities and configuration routines. If it is known that only the PID parameters will be used, then the data structure **MEIFilterGainPID** can be used directly without having to manipulate **MPIFilterCoeff**, **MPIFilterCoeff**, and **MEIFilterGainTypePID**.

## Sample Code

```

/* Read the current value of a filter's PID coefficient. Sample usage:

returnValue =
    getPidFilterCoeff(filter, MEIFilterGainPIDCoeffGAIN_PROPORTIONAL, &kp);
*/
long getPidFilterCoeff(MPIFilter filter, long index, double* value)
{
    MPIFilterConfig config;
    long returnValue = (value==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue = mpiFilterConfigGet(filter, &config, NULL);

        if (returnValue == MPIMessageOK)
        {
            switch(MEIFilterGainTypePID[index])
            {
                case MEIDataTypeLONG:
                    *value = config.gain[config.gainIndex].coeff[index].l;
                    break;
                case MEIDataTypeFLOAT:
                    *value = config.gain[config.gainIndex].coeff[index].f;
                    break;
                default:
                    returnValue = MPIMessageARG_INVALID;
            }
        }
    }
    return returnValue;
}

```

## See Also

[MPIFilterCoeff](#) | [MEIFilterGainTypePIV](#) | [MEIFilterGainPID](#) | [MEIDataType](#) | [MPIFilterGain](#)

# MEIFilterGainTypePIV

## Definition

```

static MEIDataType MEIFilterGainTypePIV[MPIFilterCoeffCOUNT_MAX] =
{
    MEIDataTypeFLOAT, /* Kpp          */
    MEIDataTypeFLOAT, /* Kip          */

    MEIDataTypeFLOAT, /* Kpv          */

    MEIDataTypeFLOAT, /* Kpff         */
    MEIDataTypeFLOAT, /* Kvff         */
    MEIDataTypeFLOAT, /* Kaff         */
    MEIDataTypeFLOAT, /* Kfff         */

    MEIDataTypeFLOAT, /* MovingIMax   */
    MEIDataTypeFLOAT, /* RestIMax     */

    MEIDataTypeFLOAT, /* Kdv          */

    MEIDataTypeFLOAT, /* OutputLimit  */
    MEIDataTypeFLOAT, /* OutputLimitHigh */
    MEIDataTypeFLOAT, /* OutputLimitLow */
    MEIDataTypeFLOAT, /* OutputOffset */

    MEIDataTypeFLOAT, /* Kiv          */
    MEIDataTypeFLOAT, /* Vintmax     */
    MEIDataTypeFLOAT, /* Ka0         */
    MEIDataTypeFLOAT, /* Ka1         */
    MEIDataTypeFLOAT, /* Ka2         */
};

```

## Description

**MEIFilterGainTypePIV** is a static array that describes the data type of the coefficients for the PIV algorithm. Specifically, an element of **MEIFilterGainTypePIV** describes which member of the union **MPIFilterCoeff** to access when using the data structure **MPIFilterCoeff**.

**MEIFilterGainTypePIV** allows for a more simple design of general case utilities and configuration routines. If it is known that only the PIV parameters will be used, then the data structure **MEIFilterGainPIV** can be used directly without having to manipulate **MPIFilterCoeff**, **MPIFilterCoeff**, and **MEIFilterGainTypePIV**.

## Sample Code

```

/*  Read the current value of a filter's PIV coefficient.  Sample usage:

    returnValue =
        getPivFilterCoeff(filter, MEIFilterGainPIVCoeffGAINVELOCITY_PROPORTIONAL,
&kpv);
*/
long getPivFilterCoeff(MPIFilter filter, long index, double* value)
{
    MPIFilterConfig config;
    long returnValue = (value==NULL) ? MPIMessageARG_INVALID : MPIMessageOK;

    if (returnValue == MPIMessageOK)
    {
        returnValue = mpiFilterConfigGet(filter, &config, NULL);

        if (returnValue == MPIMessageOK)
        {
            switch(MEIFilterGainTypePIV[index])
            {
                case MEIDataTypeLONG:
                    *value = config.gain[config.gainIndex].coeff[index].l;
                    break;
                case MEIDataTypeFLOAT:
                    *value = config.gain[config.gainIndex].coeff[index].f;
                    break;
                default:
                    returnValue = MPIMessageARG_INVALID;
            }
        }
    }

    return returnValue;
}

```

## See Also

[MPIFilterCoeff](#) | [MEIFilterGainTypePID](#) | [MEIFilterGainPIV](#) | [MEIDataType](#) | [MPIFilterGain](#)

# MPIFilterMessage

## Definition

```
typedef enum {
    MPIFilterMessageFILTER_INVALID,
    MPIFilterMessageINVALID_ALGORITHM,
    MPIFilterMessageINVALID_DRATE,
    MPIFilterMessageCONVERSION_DIV_BY_0,
    MPIFilterMessageSECTION_NOT_ENABLED,
    MPIFilterMessageINVALID_FILTER_FORM,
} MPIFilterMessage;
```

## Description

**MPIFilterMessage** is an enumeration of Filter error messages that can be returned by the MPI library.

### MPIFilterMessageFILTER\_INVALID

The filter number is out of range. This message code is returned by [mpiFilterCreate\(.\)](#) if the filter number is less than zero or greater than or equal to MEIXmpMAX\_Filters.

### MPIFilterMessageINVALID\_ALGORITHM

The filter algorithm is not valid. This message code is returned by [mpiFilterIntegratorReset\(.\)](#) if the filter algorithm is not a member of the MEIXmpAlgorithm enumeration (does not support integrators). This problem occurs if the filter type is set to user or an unknown type with [mpiFilterConfigSet\(.\)](#).

### MPIFilterMessageINVALID\_DRATE

The filter derivative rate is not valid. This message code is returned by [mpiFilterConfigSet\(.\)](#) if the filter derivative rate is less than 0 or greater than 7.

**NOTE:** The derivative rate for all gain tables must be in the range [0,7], not just the derivative rate for the current gain table.

### MPIFilterMessageCONVERSION\_DIV\_BY\_0

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) cannot convert digital coefficients to analog coefficients. When this error occurs, the offending section(s) will report its type as MEIFilterTypeUNKNOWN and will not contain any analog data.

### MPIFilterMessageSECTION\_NOT\_ENABLED

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) attempt to read postfilter data when no postfilter sections are enabled.

## **MPIFilterMessageINVALID\_FILTER\_FORM**

Returned when [meiFilterPostfilterGet\(...\)](#) or [meiFilterPostfilterSectionGet\(...\)](#) cannot interpret the current postfilter's form (when the form is something other than NONE, IIR, or BIQUAD).

### **See Also**

[mpiFilterCreate](#)

# MEIFilterType

## Definition

```
typedef enum {
    MEIFilterTypeUNITY_GAIN,
        /* B0 = 1    B1=B2=A1=A2 = 0
           (effectively acting as no filter) */
    MEIFilterTypeSINGLE_ORDER,
    MEIFilterTypeLOW_PASS,
    MEIFilterTypeHIGH_PASS,
    MEIFilterTypeNOTCH,
    MEIFilterTypeRESONATOR,
    MEIFilterTypeLEAD_LAG,
    MEIFilterTypeZERO_GAIN,
        /* b0=b1=b2=a1=a2 = 0
           (this does act as a filter.... zeroing the output) */
    MEIFilterTypeBIQUAD,
        /* Only valid for setting.
           Reading will not return these types */
    MEIFilterTypeDIGITAL_BIQUAD,
    MEIFilterTypePOLES_ZEROS,
    MEIFilterTypeDIGITAL_POLES_ZEROS,
    MEIFilterTypeUNKNOWN,
        /* algorithm couldn't figure out what
           this filter was from the coeffs! */
} MEIFilterType;
```

## Description

**NOTE:** The MPI will attempt to return analog & digital biquad and pole/zero information from [meiFilterPostfilterGet\(...\)](#) and [meiFilterPostfilterSectionGet\(...\)](#). However, the filter types MEIFilterTypeDIGITAL\_BIQUAD, MEIFilterTypePOLES\_ZEROS, and MEIFilterTypeDIGITAL\_POLES\_ZEROS are never returned by get() calls -- they are used only for setting postfilters. MEIFilterTypeBIQUAD will only be returned by meiFilterPostfilterGet(...) and meiFilterPostfilterSectionGet(...) if the analog coefficients can be calculated (there is no division by 0) and the section cannot be identified as one of the other analog filter types.

<b>MEIFilterTypeUNITY_GAIN</b>	A unity gain filter. This effectively performs no filtering.
<b>MEIFilterTypeSINGLE_ORDER</b>	A single order filter
<b>MEIFilterTypeLOW_PASS</b>	A low pass filter
<b>MEIFilterType_HIGH_PASS</b>	A high pass filter.
<b>MEIFilterTypeNOTCH</b>	A notch filter
<b>MEIFilterTypeRESONATOR</b>	A resonator filter.
<b>MEIFilterTypeLEAD_LAG</b>	A lead or lag filter.
<b>MEIFilterTypeZERO_GAIN</b>	Zeros the output of a filter.
<b>MEIFilterTypeBIQUAD</b>	An analog biquad filter. When reading postfilter data, this type means that the postfilter section could not be identified as a standard filter type.
<b>MEIFilterTypeDIGITAL_BIQUAD</b>	A digital biquad filter. This is only used for setting postfilter sections.
<b>MEIFilterTypePOLES_ZERO</b>	Analog poles and zeros filter (maximum of two poles and zeros) with unity zero-frequency amplitude. This is only used for setting postfilter sections.
<b>MEIFilterTypeDIGITAL_POLES_ZEROS</b>	Digital poles and zeros filter (maximum of two poles and zeros) with unity zero-frequency amplitude. This is only used for setting postfilter sections.
<b>MEIFilterTypeUNKNOWN</b>	Returned by <code>meiFilterPostfilterGet(...)</code> and <code>meiFilterPostfilterSectionGet(...)</code> if analog coefficients cannot be found. only digital data will be available.

## See Also

[MEIPostfilterSection](#) | [meiFilterPosterfilterGet](#) | [meiFilterPosterfilterSet](#) | [meiFilterPosterfilterSectionGet](#) | [meiFilterPosterfilterSectionSet](#)

# MEIPostfilterSection

## Definition

```

typedef struct MEIPostfilterSection {
    MEIFilterType    type;
    MEIFilterForm   form;
    struct {
        struct {
            double breakPoint;      /* Hz */
        } lowPass;

        struct {
            double breakPoint;      /* Hz */
        } highPass;

        struct {
            double centerFrequency; /* Hz */
            double bandwidth;      /* Hz */
        } notch;

        struct {
            double centerFrequency; /* Hz */
            double bandwidth;      /* Hz */
            double gain;           /* dB */
        } resonator;

        struct {
            double lowFrequencyGain; /* dB */
            double highFrequencyGain; /* dB */
            double centerFrequency; /* Hz */
        } leadLag;

        struct {
            double a1;
            double a2;
            double b0;
            double b1;
            double b2;
        } biquad;

        struct {
            double a1;
            double a2;
            double b0;
            double b1;
            double b2;
        } digitalBiquad;
    }

```

```

    struct {
        long poleCount;
        long zeroCount;
        struct {
            double real;
            double imag;
        } pole[2];
        struct {
            double real;
            double imag;
        } zero[2];
    } polesZeros;

    struct {
        long poleCount;
        long zeroCount;
        struct {
            double real;
            double imag;
        } pole[2];
        struct {
            double real;
            double imag;
        } zero[2];
    } digitalPolesZeros;

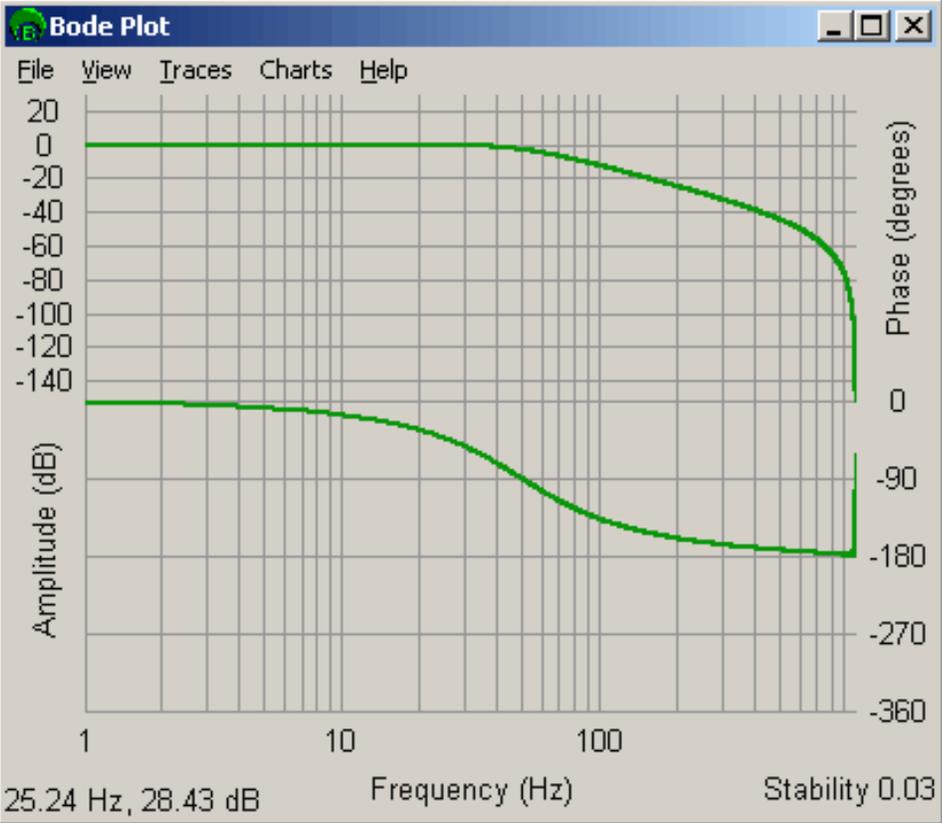
    struct {
        double d1;
        double c1;
        double c2;
        double a2;
        double a1;
        double b1;
    } stateSpaceBiquad;
} data;
} MEIPostfilterSection;

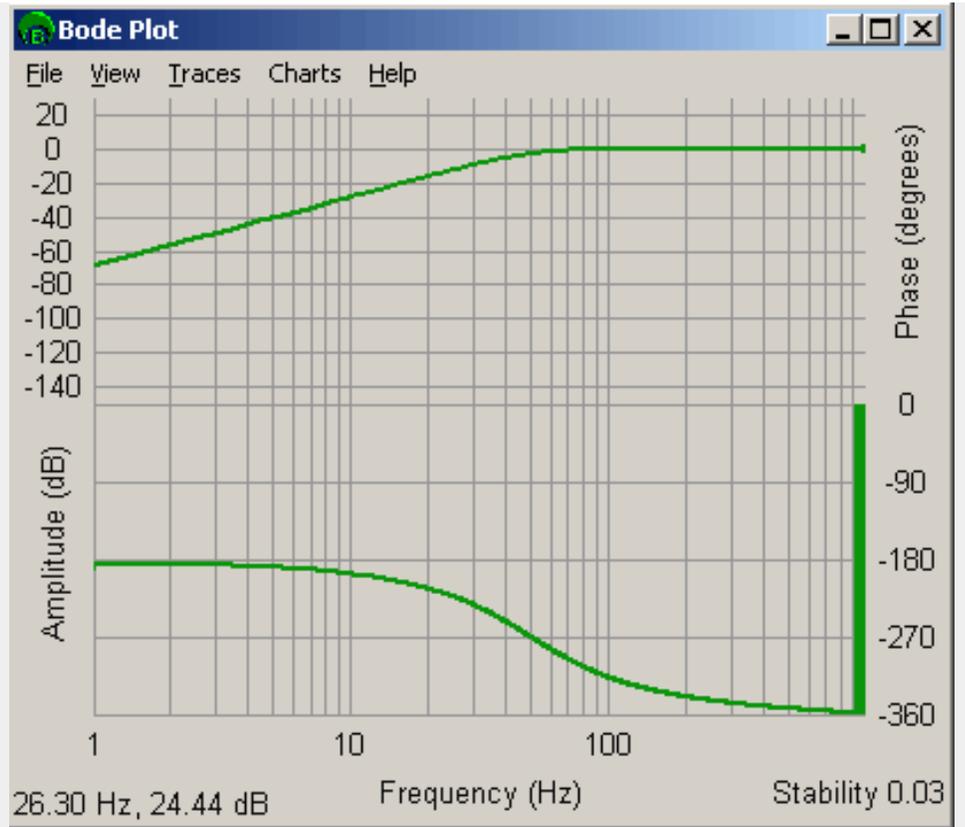
```

## Description

**MEIPostfilterSection** holds the configuration data for a single section of an MPIFilter object's postfilter. The MPI calculates the post filter coefficients and takes into consideration the sample rate of the controller at that time. If you change the sample rate of the controller, you will need to recalculate the post filters. This can be done for all filters specified in Hertz by setting the filters again with the MPI. The MPI will calculate the filters using the current servo sample rate.

Postfilters are used to digitally filter the output of a control loop. One common use for postfilters is the compensation of system resonances.

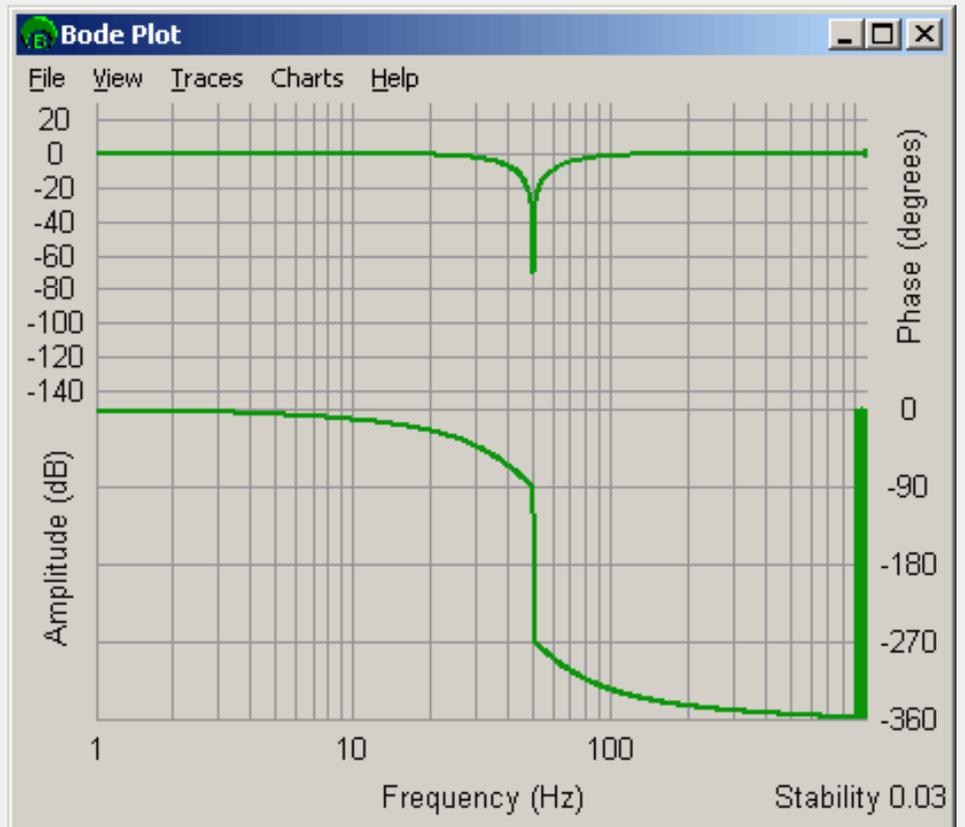
<b>type</b>	The postfilter section type. This field determines which field of the MEIPostfilterSection.data union is used by meiFilterPostfilter.() methods. More information about particular filter types can be found below and in the <a href="#">MEIFilterType</a> documentation.
<b>form</b>	The form of a postfilter section. The form determines how a particular postfilter section is calculated on the controller. All forms have certain limitations and tradeoffs. Please refer to <a href="#">MEIFilterForm</a> for more information.
<b>lowPass.breakpoint</b>	<p>The break point (measured in Hertz) of a low pass postfilter section.</p>  <p style="text-align: center;">Example of a 50 Hz low pass filter.</p>
<b>highPass.breakpoint</b>	The break point (measured in Hertz) of a high pass postfilter section.



Example of a 50 Hz High pass filter

### notch.centerFrequency

The bandwidth (measured in Hertz) of a notch postfilter section.

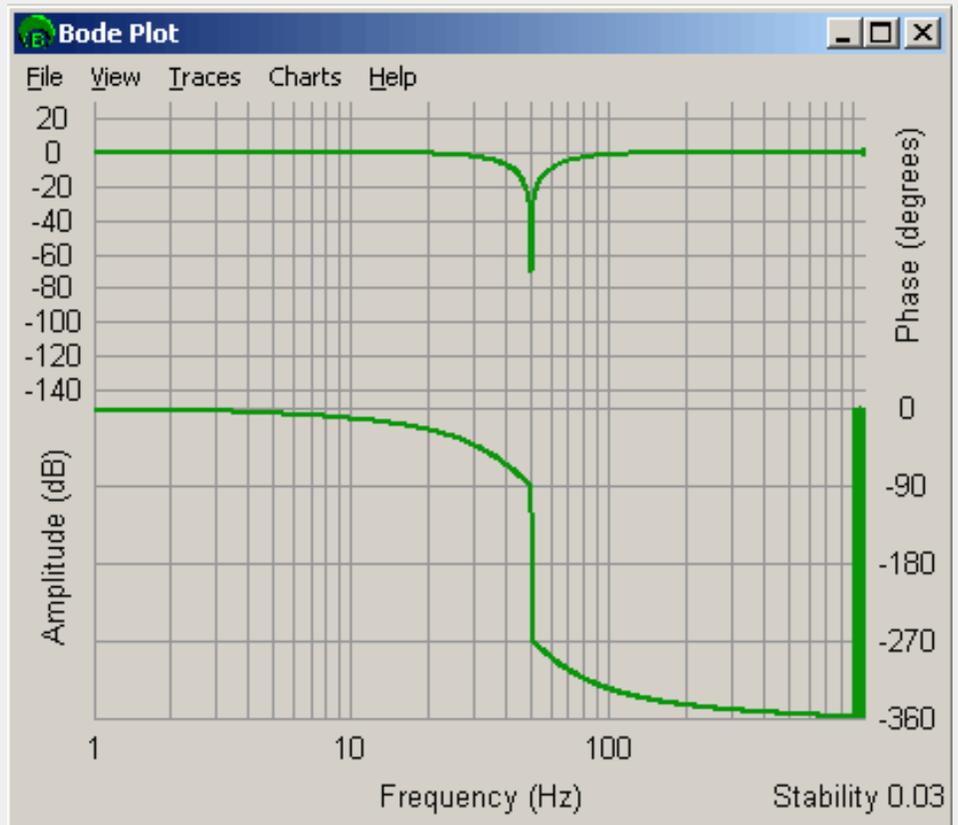


Example of a 50 Hz Center / 50 Hz Bandwidth Notch filter. Note that phase wrapping gives the illusion that phase drops 180 degrees after the

center frequency. The phase raises by 180 degrees.

**notch.bandwidth**

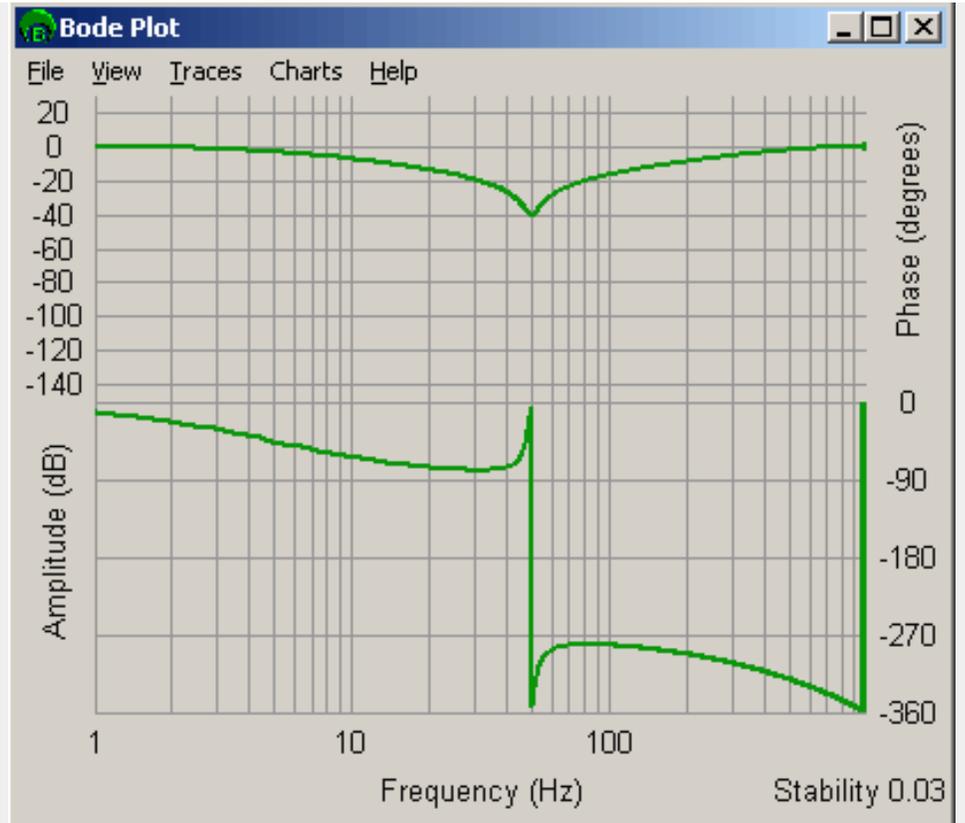
The center frequency (measured in Hertz) of a notch postfilter section.



Example of a 50 Hz Center / 50 Hz Bandwidth Notch filter. Note that phase wrapping gives the illusion that phase drops 180 degrees after the center frequency. The phase raises by 180 degrees.

**resonator.centerFrequency**

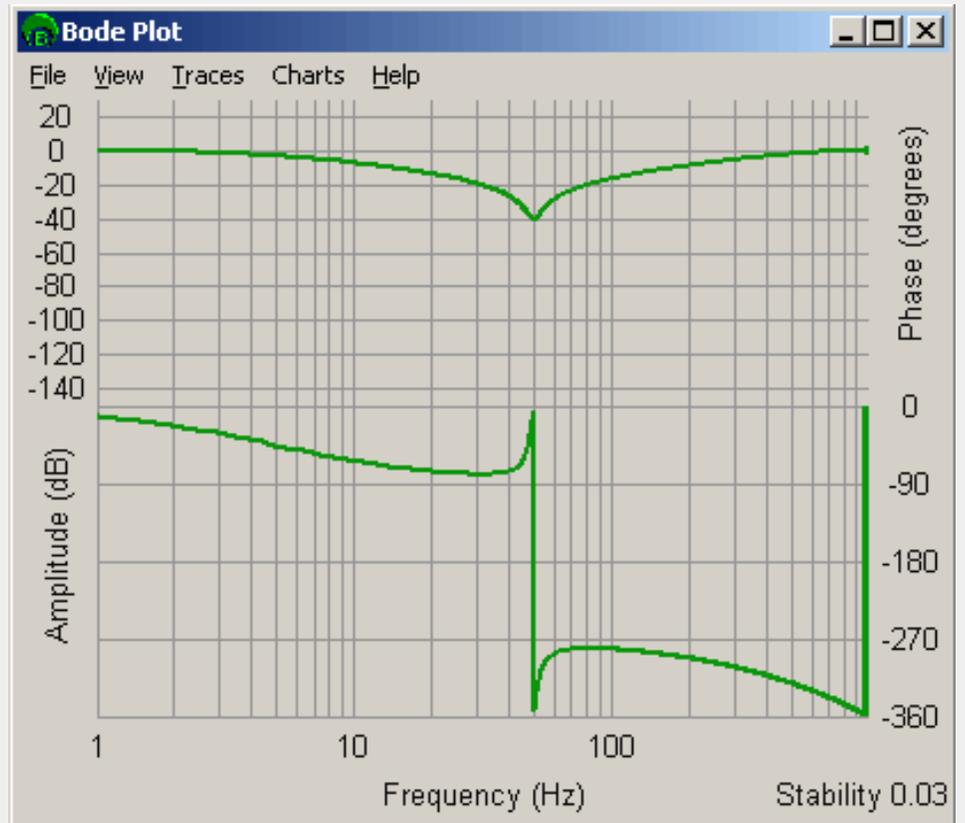
The center frequency (measured in Hertz) of a resonator postfilter section.



Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

### resonator.bandwidth

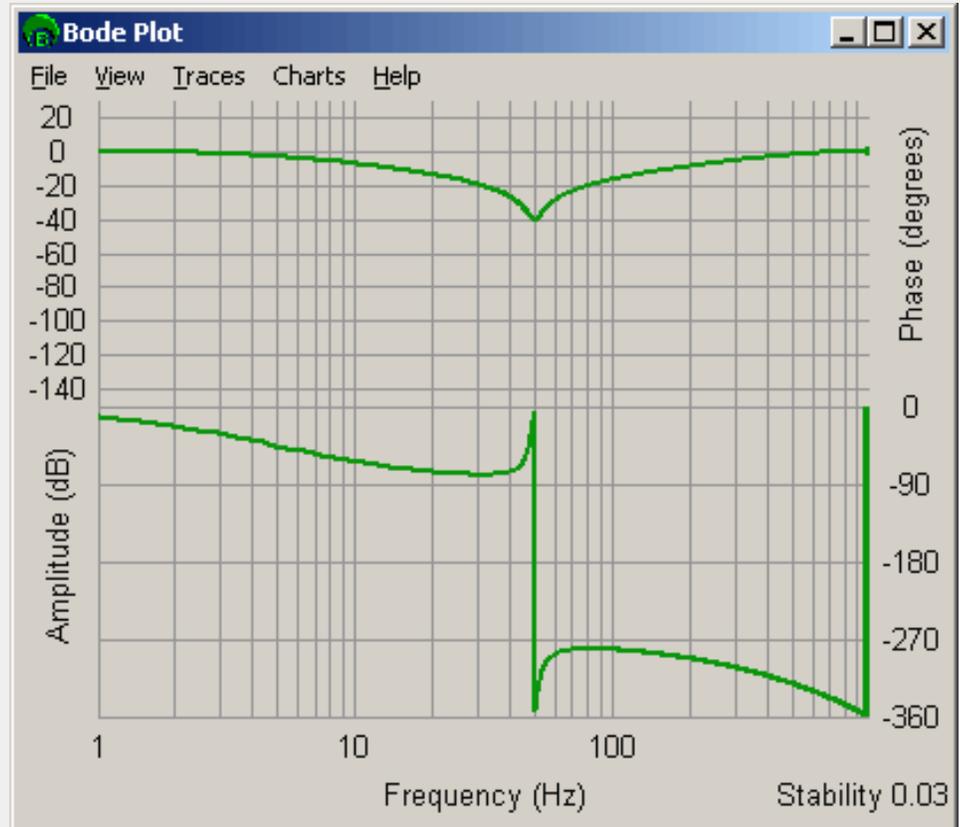
The bandwidth (measured in Hertz) of a resonator postfilter section.



Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

### resonator.gain

The center frequency gain (measured in dB) of a resonator postfilter section.



Example of a 50 Hz center / 50 Hz Bandwidth / -40 dB Gain Resonator filter. Note that phase wrapping gives the illusion that the phase drops 360 degrees after the center frequency.

### leadLad.centerFrequency

The center frequency (measured in Hertz) of a lead or lag postfilter section. The amplitude at this frequency is the average amplitude of the low and high frequency amplitudes. The gain (measured in dB) at this point is given by:

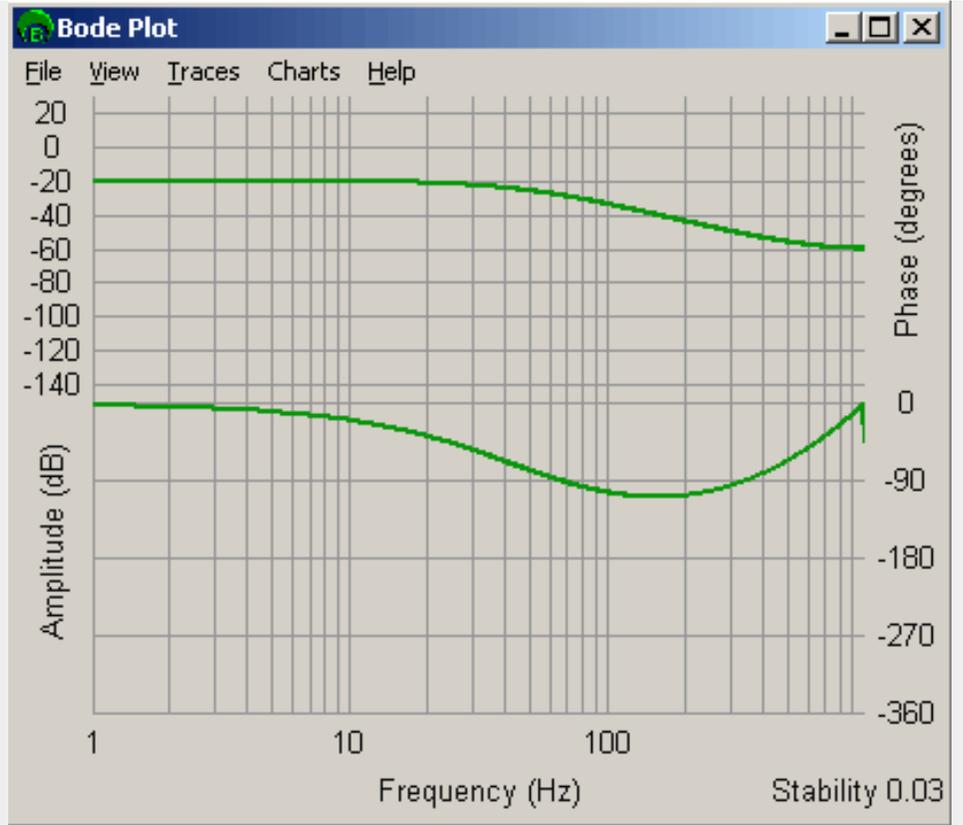
$$centerFrequencyGain = 20 \cdot \log_{10} \left( \frac{\frac{lowFrequencyGain}{20} + \frac{highFrequencyGain}{20}}{2} \right)$$



Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.

### **leadLag.lowFrequencyGain**

The low frequency gain (measured in dB) of a lead or lag postfilter section.



Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.

### leadLag.highFrequencyGain

The high frequency gain (measured in dB) of a lead or lag postfilter section.



	Example of a -20 dB low frequency gain / -60 dB high frequency gain / 50 Hz center lead lag filter.
<b>biquad.a1</b>	The analog coefficients of a single order or bi-quad postfilter section.
<b>biquad.a2</b>	Analog values of the postfilter coefficients are produced as parts of a Laplace Transform:
<b>biquad.b0</b>	$H(s) = \frac{b_0 + b_1 \cdot s + b_2 \cdot s^2}{1 + a_1 \cdot s + a_2 \cdot s^2} \text{ where } s = j \cdot \omega_{warped}$
<b>biquad.b1</b>	and
<b>biquad.b2</b>	$\omega_{warped} = 2 \cdot sampleRate \cdot \tan\left(\frac{\pi \cdot f}{sampleRate}\right)$
<b>digitalBiquad.a1</b>	
<b>digitalBiquad.a2</b>	
<b>digitalBiquad.b0</b>	The digital coefficients of a single order or bi-quad postfilter section.
<b>digitalBiquad.b1</b>	
<b>digitalBiquad.b2</b>	
<b>digitalBiquad.d1</b>	
<b>digitalBiquad.c1</b>	
<b>digitalBiquad.c2</b>	
<b>digitalBiquad.a2</b>	The digital coefficients of a state-space bi-quad postfilter section.
<b>digitalBiquad.a1</b>	
<b>digitalBiquad.b1</b>	
<b>polesZeros.poleCount</b>	
<b>polesZeros.zeroCount</b>	
<b>polesZeros.pole[].real</b>	Analog poles and zeros.
<b>polesZeros.pole[].imag</b>	
<b>digitalPolesZeros.poleCount</b>	
<b>digitalPolesZeros.zeroCount</b>	Digital poles and zeros.

<b>digitalPolesZeros.pole[].real</b>	
<b>digitalpolesZeros.pole[].imag</b>	
<b>stateSpaceBiquad.d1</b>	State space coefficients.
<b>stateSpaceBiquad.c1</b>	
<b>stateSpaceBiquad.c2</b>	
<b>stateSpaceBiquad.a2</b>	
<b>stateSpaceBiquad.a1</b>	
<b>stateSpaceBiquad.b1</b>	

## Sample Code

```

/* Set a 4th order low-pass post-filter by using two
   2nd order low-pass sections.
   Sample usage:

   returnValue =
       fourthOrderLowPass(filter, 300 /* Hz */);
*/
long filterFouthOrderLowpass(MPIFilter filter, long breakPointFrequency)
{
    MPIFilterConfig config;
    MEIPostfilterSection sections[2];
    long returnValue;

    section[0].type = MEIFilterTypeLOW_PASS;
    section[0].form = MEIFilterFormINT_BIQUAD;
    section[0].lowPass.breakpoint = breakPointFrequency;
    section[1] = section[0]; /* copy first section */

    returnValue =
        meiFilterPostfilterSet(filter, 2, sections);

    return returnValue;
}

```

## See Also

[MEIFilterType](#) | [MEIFilterForm](#) | [MEIMaxIIRCoefficients](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#) | [Post Filter Theory](#)

# MPIFilterCoeffCOUNT\_MAX

## Definition

```
#define MPIFilterCoeffCOUNT_MAX (20)
```

## Description

**MPIFilterCoeffCOUNT\_MAX** is a constant that defines the maximum number of filter coefficients contained in a gain table.

## See Also

[MPIFilterCoeff](#)

# MPIFilterGainCOUNT\_MAX

## Definition

```
#define MPIFilterGainCOUNT_MAX (5)
```

## Description

**MPIFilterGainCOUNT\_MAX** is a constant that defines the maximum number of filter gain tables. The first gain table is used by the standard filter types (all filter types except for the user filter type as defined by the structure MEIXmpAlgorithm). Additional gain tables can be used for manual or automatic gain switching. For firmware that implements automatic gain switching, please [contact](#) Motion Engineering. Manual gain switching can be accomplished by specifying the gainIndex of the mpiFilterConfig structure using the mpiFilterConfigSet method. Valid gainIndex values range from 0 to MPIFilterGainCOUNT\_MAX.

## See Also

[MPIFilterGain](#)

# MEIMaxBiQuadSections

## Definition

```
#define MEIMaxBiQuadSections (6)
```

## Description

**MEIMaxBiQuadSections** is the maximum number of Bi-Quad sections a postfilter can use.

**NOTE:** The PIV algorithm uses the last Bi-Quad section internally. Thus a user can only use (MEIMaxBiQuadSections - 1) Bi-quad sections with the PIV algorithm.

## See Also

[MEIPostFilterSection](#) | [meiFilterPostfilterGet](#) | [meiFilterPostfilterSet](#) | [meiFilterPostfilterSectionGet](#) | [meiFilterPostfilterSectionSet](#)