

NAME

mac_hdw – macro-hardware facility

DESCRIPTION

The macro-hardware facility links user-defined macro functions with `spec`'s built-in C code for motor, counter and MCA control, providing a simple method to control hardware for which `spec` doesn't have built-in support. (MCA support added in release 5.09.03-1.)

The facility also provides a simplified method for implementing calculational pseudomotors. These pseudomotors correspond to motions formed by a combination of real motors. For example, the motions of a slit's gap and offset are derived from the real motions of the slit's blades, or a table's height, pitch and roll are derived from the real motions of the table legs.

The macro counter implementation also includes an optional polling feature, to accommodate hardware devices that have a run cycle that may take longer than the timing cycle controlled by `spec`'s configured master timer (as of `spec` release 5.08.03-4).

For regular macro-hardware motors, counters and MCA devices, there are three possible macro functions that can be user defined. During start up and on the `reconfig` command, for each macro-hardware "controller" unit and for each macro-hardware motor and counter channel assigned in the hardware `config` file, `spec` will call the "configuration" user-defined macro function. For each low-level command needed to implement a motor, counter or MCA control function, `spec` will make calls to the "command" user-defined macro function. Finally, `spec` will call the "parameter" user-defined macro function to set or get values for associated motor, counter and MCA parameters.

Calculational pseudomotors are pseudomotors whose positions depend on the positions of real motors. For such a macro-hardware pseudomotor, `spec` will call a "calculation" user-defined macro function that will calculate the position of each pseudomotor based on the real motor positions or will calculate the real motor positions based on a target pseudomotor position. There is no "command" macro function for calculational pseudomotors.

To enable a macro motor or counter, configure a macro controller on the **Devices** screen of configuration editor, as follows:

MOTORS	DEVICE	ADDR	<>MODE	NUM	<>TYPE
YES	motxx	-		5	Macro Motor

SCALERS	DEVICE	ADDR	<>MODE	NUM	<>TYPE
YES	cntxx	-		5	Macro Counter
YES	cntzz	-		5	Macro Counter/Timer

Here, the `motxx`, `cntxx` or `cntzz` entries for `DEVICE` are arbitrary names, but will be used by `spec` as a prefix for the macros associated with the particular macro-hardware controller.

To enable a macro-hardware MCA, configure the controller on the MCA-like configure screen as follows:

MCA-like	DEVICE	ADDR	<>MODE	<>TYPE
0 YES	mac_mca	-		Macro MCA

The `ADDR` field is optional in all cases, but may contain a string value. For motors and counters, the value can be retrieved within `spec` using

```
motor_par(mne, "address")
```

or

```
counter_par(mne, "address")
```

where `mne` is the mnemonic of any motor or counter channel associated with the controller (as of `spec` release 5.06.04-4). For MCA devices, use

```
mca_par("address")
```

or

```
mca_spar(num, "address")
```

where *num* is the MCA number. In addition, within the macro functions described below, the value will assigned to a local variable named *prefix_ADDR* (as of spec release 5.06.04-8).

Nonstandard optional controller parameters, entered and modified using the *p* command on the **Devices** screen and stored in the *config* file prefixed by the string *CONPAR:*, are accessible within the macro hardware functions as elements of the associative array *prefix_CONPAR[]*, where the array elements are indexed by the parameter name. In addition, the parameters for motors and counters are also available using

```
motor_par(mne, par)
```

or

```
counter_par(mne, par)
```

where *mne* is the mnemonic of any motor or counter channel associated with the controller and *par* is the parameter name (as of spec release 5.08.05-1). For MCA devices, use

```
mca_par(par)
```

or

```
mca_spar(num, par)
```

where *num* is the MCA number.

Individual motor and counters are assigned to macro-hardware controllers on the **Motor** and **Scaler** screens, respectively. On the **Motor** screen, select *MAC_MOT* as the controller type. For ordinary counters, choose *MAC_CNT* as the controller type on the **Scaler** screen. For polled counters (introduced in spec release 5.08.03-3), choose the *MAC_CNTP* controller type.

The Macro Functions

For standard macro motors and macro counters, the three possible user-defined macro functions have names formed by prepending the prefix defined in the *config* file to *_config()*, *_cmd()* and *_par()*. For calculational pseudomotors, a *_calc()* function is needed rather than the *_cmd()* function.

For all the macro functions, the first argument is the motor or counter number if the call applies to a single channel or the string ". ." if the call applies to all channels associated with the motor or counter controller. For MCA devices, the first argument is the MCA number. The second argument is a string that contains a key specific to the command. Remaining arguments, if any, contain parameters.

As explained in the previous section, if an optional *ADDR* string is included in the *config* file for the associated controller, the string will be assigned to a variable named *prefix_ADDR*. Likewise, nonstandard optional parameters configured for the associated controller are assigned to an associative array variable named *prefix_CONPAR*. The variables are only visible within the macro functions.

To send an error back to spec from the macro functions, return the special string ".error.".

Set the spec *DEBUG* level to 128 to display the macro function calls, which may help make clear when and in what order the various macro functions are called.

The *_config()* Function

The *_config()* function is called after reading the *config* file. On startup, the function is called after the initial command files have been read, so that macros defined in the initial command files can be used to set up the macro hardware. For regular macro motors and counters, this function is optional and need not exist. The function is required for calculation pseudomotors to configure the dependencies between real and pseudomotors. The function is also required for MCA devices to configure the maximum number of channels and the native data type.

For motor and counter controllers, the `_config` macro function is called as follows:

`prefix_config(".", "ctrl", p1, p2)` – Called for each macro-hardware controller unit. The parameter `p1` is the unit number of the controller, while `p2` is the number of channels set as NUM on the **Devices** screen of the configuration editor. If the function returns the string ".error.", `spec` will consider the controller unresponsive and won't call the macro functions for the associated channels.

`prefix_config(mne, "mot", unit, module, chan)` – Called for each macro motor channel, where `mne` is the motor mnemonic, `unit` is the unit number of the associated controller, `module` is the optional module number and `chan` is the channel number. For calculational pseudomotors, this call must return a string containing a list of mnemonics for the real motors on which the pseudomotor depends. If the function returns the string ".error.", `spec` will consider the channel unusable.

`prefix_config(mne, "cnt", unit, 0, chan)` – Called for each macro counter channel, where `mne` is the counter mnemonic, `unit` is the unit number of the associated controller, and `chan` is the channel number. The fourth argument is currently always zero. If the function returns the string ".error.", `spec` will consider the channel unusable.

`prefix_config(num, "mca")` – Called for each macro MCA, where `num` is the MCA number. This function must return a string with two arguments in any order: the maximum number of channels for the MCA and the native data type. The data type is one of the words `byte`, `ubyte`, `short`, `ushort`, `long`, `ulong`, `float` or `double`. An example return string is "4096 ulong". If the function returns the string ".error.", `spec` will consider the MCA unit unusable.

For motors and counters, the unit and channel numbers are assigned in the configuration editor just as with other motors and counters. The optional module number became available in `spec` release 5.04.03-2. Unit numbers are assigned consecutively to each controller type. That is, the first macro motor controller listed is unit zero, as is the first macro counter controller, both independent of other controller types. Counter unit and channel numbers are assigned explicitly on the **Scaler** screen of the configuration editor. Motor unit, module and channel numbers can be explicitly assigned in the unit/channel field of the configuration editor. If left blank, the channels numbers are assigned automatically in consecutive order.

For macro hardware designed for general purpose applications, these calls to the `_config()` macro function can be used to set up the rest of the macro interface. For simple applications, this function won't be required for motors and counters.

MOTORS

The `_cmd()` Function For Motors

The `_cmd()` function is called to control regular macro motors. There are many more command keys in the list below than any particular macro motor would need. Only the command keys that are relevant to the particular application should be included in the user-defined macro function.

The syntax of the function call is:

`prefix_cmd(mne, key [, p1 [, p2]] [,unit])` – Called by the C code for all operations related to motor control. `mne` is the string "." for keys that apply to all motors. `mne` is the motor number for keys that apply to individual motors. Each key is only called one way or the other. `key` is a string containing the particular command. `p1` and `p2`, if present, are parameters related to the command. If the second argument is the string "." the `unit` parameter will be included and specifies the unit number for which the command applies (as of `spec` release 5.07.03-3).

In the following, the phrase “sent when changed” means the `_cmd()` macro function is only called with the given key before the first applicable move command or home command after `spec` reads the `config` file, either on startup or after a `reconfig` command (included in the `config` macro) or after the associated parameter has been changed using the `motor_par()` function.

- "base_rate" – Sent when changed. The `p1` parameter contains the base rate in units of Hz.
- "slew_rate" – Sent when changed. The `p1` parameter contains the slew rate in units of Hz.
- "acceleration" – Sent when changed. Also called if the base rate or slew rate have changed, since some motor controllers need to be told to recalculate acceleration ramps if the velocity parameters change. The `p1` parameter contains the acceleration time in units of milliseconds. The `p2` parameter contains the acceleration in units of steps per second per second.
- "home_base_rate" – Sent when changed. The `p1` parameter contains the home base rate in units of Hz.
- "home_slew_rate" – Sent when changed. The `p1` parameter contains the home slew rate in units of Hz.
- "home_acceleration" – Sent when changed. The `p1` parameter contains the home acceleration time in units of milliseconds. The `p2` parameter contains the home acceleration in units of steps per second per second.
- "preread_all" – Sent prior to a possible read of all the motors. Note, depending on the configured hardware read modes for the motors, there may be no subsequent commands to read a motor associated with this controller. Either "preread_all" or "preread_one" (below), but not both, will be called prior to the "position" call below.
- "preread_one" – Sent prior to reading an individual motor. Either "preread_all" (above) or "preread_one", but not both, will be called prior to the "position" call below.
- "position" – For this key, the macro function must return the current motor position in dial units. This call will be preceded by a call with a key of either "preread_one" or "preread_all".

The macro-hardware motors use natural units for positions. For such motor controllers, `spec` only uses the steps-per-degree parameter in the `config` file to determine the precision of the position values. The precision is determined by the magnitude of the parameter. For example, a value of 1000 will cause `spec` to round to the nearest 0.001, while a value of 5000 will cause `spec` to round to the nearest 0.0002. The rounding is performed on the value returned by the "position" key.

- "set_position" – Sent to set the current dial position of the macro motor. The parameter `p1` contains the position in dial units.
- "prestart_all" – For regular moves, sent if any motors associated with the macro motor controller need to be moved.
- "prestart_one" – For regular moves and homing moves, sent for each motor that needs to be moved. For regular moves, a call of "prestart_all" comes first.
- "magnitude" – For regular moves, sent with the magnitude of the move in dial units. The parameter `p1` contains the position in dial units and includes the sign of the move. The magnitude is also included with the "start_one" key (below).
- "start_one" – Sent to start a regular move for one motor. The parameter `p1` contains the target position in dial units to accommodate a controller that requires absolute positions. The parameter `p2` contains the magnitude of the move in dial units to

accommodate a controller that requires relative positions.

"start_all" – Sent after all "start_one" commands to accommodate controllers that use a simultaneous start.

"get_status" – Sent to get the move and limit status of individual motors. If the motor is moving, the macro function must return a value with bit 0x02 set. If the low limit is active, the return value must have bit 0x04 set. If the high limit is active, the return value must have bit 0x08 set. Setting bit 0x10 indicates an “emergency stop” and setting bit 0x20 indicates a motor fault (as of spec release 5.07.04-4), both which currently result in similar behavior to when a limit is hit. Otherwise, the macro function must return a value of zero.

"flush_all" – Sent before the “hard” position synchronization that occurs on startup and before and after reading the *config* and *settings* files on a reconfig command.

"flush_one" – Sent for each motor after the "flush_all" key above and before a "get_status" during the position synchronization.

"abort_one" – Sent for each active motor when motors are halted, normally either by a ^C from the keyboard or by a stop() command.

"abort_all" – Sent to each macro motor controller that has busy motors when motor are halted. The key is sent after the "abort_one" keys are sent. This command accommodates controllers that allow a single command to halt all its associated motors.

"search" – Sent to initiate a home or limit search. A "prestart_one" call will precede this call. The parameter *p1* indicates the type of search as follows.

"home" – Find the home position as appropriate.

"home+" – Find the home position by moving in the positive direction.

"home-" – Find the home position by moving in the negative direction.

"lim+" – Find the positive limit.

"lim-" – Find the negative limit.

The parameter *p2* contains the position in dial units that corresponds to the home or limit switch.

"diff_position" – Sent if the motor is configured for a settle time. To configure a motor for settle time, both the DC dead-band and the DC settle-time parameters have to be set (usually from the first optional motor parameter screen of the configuration editor – get there by typing an m from the primary motor screen). The settle-time parameter is in seconds. spec will wait for at least as long as the settle time before treating a move as complete. In addition, spec will wait until the value returned by this call is less than the dead band, but for no longer than five seconds. The preferred units for dead band are steps, but it is only necessary that the units in the *config* file agree with the units returned by this call.

Backlash is handled as two separate move commands. If the macro function will take care of backlash, set the backlash parameter to zero in the *config* file.

The default behavior with respect to reading the motor position is to only ask for the motor position from the controller during position synchronization or at the end of a move. The motor parameter “hardware read mode” (on the second optional motor parameter screen of the configuration editor) can be set to require spec to ask for the position before each move and/or for every read_motors() call from user level. The hardware read mode can also be set so that position discrepancies are always silently resolved in favor of the value returned by the controller (or macro function).

A minimal implementation would likely recognize the keys "start_one", "get_status", "position" and "set_position". An example that does nothing useful follows:

```
def motxx_cmd(mne, key, p1, p2) '{
    global demo_pos[]

    if (key == "set_position") {
        demo_pos[mne] = p1
        return
    }
    if (key == "position") {
        return(demo_pos[mne])
    }
    if (key == "start_one") {
        demo_pos[mne] = p1
        return
    }
    if (key == "get_status") {
        return(0)
    }
}'
```

The `_par()` Function For Motors

The `_par()` macro function is called when various motor parameters are set, and when the `motor_par()` function is used to retrieve a user-defined parameter. The *mne* argument will always be a motor number and never the string "." that is used with the other user-defined macro-hardware functions.

The function will be called as:

`prefix_par(mne, key, "get")` – The function should return a value for the parameter named as *key* for motor number *mne*. The macro function will never be called to get a parameter value when *key* is a parameter name that is built into the spec C code.

`prefix_par(mne, key, "set", p1 [, p2])` – Called when various parameters change their value, as described below.

The built-in parameter names are as follows. The `_par()` function will never be called with "get" for these parameters, as their values are maintained internally.

The first set below are called only when `motor_par()` is executed from user-level.

"acceleration" – *p1* contains the acceleration time in milliseconds. *p2* contains the acceleration value in steps per second per second.

"backlash" – *p1* contains the new backlash value in steps.

"backlash_rate" – *p1* contains the new backlash rate in Hz.

"base_rate" – *p1* contains the new base rate in Hz.

"disable" – *p1* contains 1 or 0, depending on whether the motor was disabled or undisable (available as of spec release 5.06.03-8).

"slew_rate" or "velocity" – *p1* contains the new slew rate in Hz.

"step_size" – *p1* contains the new step-size parameter.

The following two keys are called when an associated function is executed from user level.

"limits" - *p1* contains the low limit in dial units. *p2* contains the high limit in dial units.

Called when the user-level `set_lim()` function is executed.

"offset" - Called when the user-level `chg_offset()` function is executed. *p1* contains the offset in user units.

The following optional motor parameters generate a call to the user-defined macro-hardware function when the values are read from the *config* file and when the values are set with the `motor_par()` function. See the *motors* help file for additional information on the parameters.

```
"home_slew_rate"
"home_base_rate"
"home_acceleration"
"dc_dead_band"
"dc_settle_time"
"dc_proportional_gain"
"dc_derivative_gain"
"dc_integral_gain"
"dc_integration_limit"
"dc_following_error"
"dc_sampling_interval"
"encoder_step_size"
"step_mode"
"slop"
"read_mode"
"deceleration"
"torque"
"misc_1"
"misc_2"
"misc_3"
"misc_4"
"misc_5"
"misc_6"
```

There are a number of valid arguments to `motor_par()` which will not generate a call to the `_par()` macro function at all. These include "disable", "unit", "channel", "responsive", "controller", "device_id", "active", "status", "config_step_size", "config_acceleration", "config_velocity", "config_base_rate", "config_backlash", "low_limit" and "high_limit".

Arguments to `motor_par()` that are not recognized by the built-in C code will be passed on, as is, to the `_par()` user-defined macro function.

Calculational Pseudo Motors

For calculational pseudomotors, two macro functions must be provided with names formed by prepending the prefix from the *config* file to `_config()` and `_calc()`.

The `_config()` function, when called with the key equal to "mot", must return a string that contains a space-delimited list of mnemonics for the real motors on which motor *mne* depends.

The `_calc()` macro function will be called as follows:

prefix_calc(mne, mode) - When called with *mode* equal to zero, the function should assign a value to `A[mne]` corresponding to the current position of the pseudomotor *mne*. When called with *mode* equal to one, the function should assign a value to `A[mne]` corresponding to the target position of the real motor *mne*.

When called to calculate the real motor positions for a move (with *mode* equal 1), the function will first be called with *mne* set to the string ". . .", then called with each of the real motor mnemonics as arguments, in turn. One can use the initial call to calculate quantities that depend on the current positions of the real motors before new values are assigned in subsequent calls.

The `_calc()` function should only include commands to calculate pseudomotor positions from real motor positions or vice versa. The function should not contain calls to do hardware access. In fact, calls to the built-in functions `wait()` or `read_motors()` will return immediately if called from the `_calc()` macro function, to avoid possible recursion as those built-in functions can subsequently call the invoking `_calc()` macro function.

Note, if the pseudomotor is disabled via `motor_par(mne, "disable", 1)` the calls to `_calc()` with *mode* equal to 1 will be skipped. That will prevent new positions from being calculated for the associated real motors.

With spec release 5.06.04-4, a special `motor_par()` option called "chan0" is available for macro motors. The command

```
motor_par(mne, "chan0")
```

will return the motor number of the macro motor in the first channel of the same controller unit and module number of the macro motor with mnemonic *mne*. This feature allows for simplified implementation of general-purpose calculational pseudomotors. For example, if there are parameters associated with a group of motors, one copy of the parameters can be assigned to the "chan0" motor and then accessed by the related motors.

The following examples implements calculational pseudomotors for a slit. The slit has two blades whose real motors have mnemonics `s12t` and `s12b` (slit 2 top and bottom). The pseudomotors are the slit gap and the slit offset position with mnemonics `s12g` and `s12o`, respectively.

The **Devices** screen of the configuration editor should look as follows for the controller:

MOTORS	DEVICE	ADDR	<>MODE	NUM	<>TYPE
YES	slit2			2	Macro Motor

The macros would be as follows:

```
def slit2_config(mne, type, unit, module, chan) '{
    if (type == "mot")
        return "s12t s12b"
}'
def slit2_calc(mne, mode) '{
    if (mode == 0) {
        if (mne == s12g)
            A[mne] = A[s12t] + A[s12b]
        if (mne == s12o)
            A[mne] = (A[s12t] - A[s12b])/2
    } else {
        if (mne == s12b)
            A[mne] = -A[s12o] + A[s12g]/2
        if (mne == s12t)
            A[mne] = A[s12o] + A[s12g]/2
    }
}'
```

The following `make_slit_macros` macro can be used to generate macros such as the above.

```
def make_slit_macros '{
    local name, file
    local l, r, g, o

    file = getval("File for macros", "tty")
    name = getval("Name of slit", "Slit1")
    r = getval("Mnemonic for right/top slit", "slr")
    l = getval("Mnemonic for left/bottom slit", "sll")
    g = getval("Mnemonic for gap", "slvg")
    o = getval("Mnemonic for offset", "slvo")

    fprintf(file, "\n\
def %s_config(mne, type, unit, module, chan) \'{\n\
    if (type == \"mot\")\n\
        return \"%s %s\"\n\
}\n\
def %s_calc(mne, mode) \'{\n\
    if (mode == 0) {\n\
        if (mne == %s)\n\
            A[mne] = A[%s] + A[%s]\n\
        if (mne == %s)\n\
            A[mne] = (A[%s] - A[%s])/2\n\
    } else {\n\
        if (mne == %s)\n\
            A[mne] = -A[%s] + A[%s]/2\n\
        if (mne == %s)\n\
            A[mne] = A[%s] + A[%s]/2\n\
    }
}\n\
}\n\
\", name,l,r, name, g,r,l, o,r,l, l,o,g, r,o,g)
}'
```

The following example implements a table-height pseudomotor with mnemonic `t1z` that is the average height of the three real motors `t1f`, `t1b1` and `t1b2` that correspond to the table legs. When the height is moved, each leg is moved by an amount equal to the difference of the current height and the target height. The current average height needs to be calculated from the current real-motor positions before the new positions are assigned. The feature where the `_calc()` function is called with `mne` set to the string `..` before being called with the real motor mnemonics is used to save the average position in a global variable to be used in the subsequent calls.

```
def tabl_config(mne, type, unit, module, chan) '{
    global tabl_ave
    if (type == "mot")
        return "t1f t1b1 t1b2"
}'
def TE1_vert_jack_calc(mne, mode) '{
    if (mode == 0) {
        if (mne == t1z)
            A[mne] = (A[t1f] + A[t1b1] + A[t1b2])/3
    } else {
        if (mne == "..")
            tabl_ave = (A[t1f]+A[t1b1]+A[t1b2])/3
    }
}'
```

```

        else if (mne == t1f)
            A[mne] += A[t1z] - tab1_ave
        else if (mne == t1b1)
            A[mne] += A[t1z] - tab1_ave
        else if (mne == t1b2)
            A[mne] += A[t1z] - tab1_ave
    }
}'

```

This last example shows how an *energy* pseudomotor can be readily created that ties in with the standard *spec* energy macros from the *energy.mac* distribution file. Such a pseudomotor can then be scanned using the standard motor scans. Note, the existing energy macros, such as *EScan*, *moveE*, etc., will still work.

The `_config()` function below makes use of the monochromator mnemonic conventions set in *energy.mac*.

```

def Energy_config(mne, type, unit, module, chan) '{
    if (type == "mot") {
        if (mono_type == 1)
            return(motor_mne(Mono))
        if (mono_type == 2)
            return("mono mon_y mon_z")
        if (mono_type == 3)
            return("monu mond montrav")
        if (mono_type == 4)
            return("monu mond montrav monoff")
    }
}'
def Energy_calc(mne, mode) '{
    if (mode == 0) {
        calcE
        A[energy] = hc_over_e / LAMBDA
    } else {
        calcM A[energy]
        calcE
    }
}'

```

The prefix *Energy* and motor mnemonic *energy* are, as always, arbitrary, but must match the configured device name on the **Devices** screen and motor mnemonic on the **Motor** screen.

COUNTER/TIMERS

The `_cmd()` Function For Counters

The `_cmd()` macro function for counters is called in response to *spec*'s counter-related built-in functions and commands: `tcount()`, `mcount()`, `wait()`, `stop()`, `move_cnt`, `getcounts` and `sync`.

There are three possible counter channel configurations: a master timer, a regular counter channel or a polled counter channel. The function calls differ depending on the configuration.

A master timer belongs to controller type "Macro-Hardware Counter/Timer". The counter type is `MAC_CNT` and the individual channel is identified as either `timer` or `monitor`. For such a channel, the "start_one" call will include a nonzero parameter indicating the count mode as described below. Also, "get_status" calls will be made to determine when the

count time has completed.

A regular counter channel belongs to controller type "Macro-Hardware Counter" with counter type MAC_CNT and function type counter.

The polled counter type also has controller type "Macro-Hardware Counter" and function type counter, but the counter type is MAC_CNTP. Polled counters will receive "get_status" calls to see when the counter is no longer busy.

The syntax of the function call is:

prefix_cmd(mne, key [, p1 [, p2]] [,unit]) – Called by the C code for all operations related to counter/timer control. *mne* is the string "." for keys that apply to all counters. *mne* is the counter number for keys that apply to individual counters. Each key is only called one way or the other. *key* is a string containing the particular command. *p1* and *p2*, if present, are parameters related to the command. If the *mne* argument is the string "." the *unit* parameter will be included and specifies the unit number for which the command applies (as of spec release 5.07.03-3).

When the "start_all" and "start_one" keys are sent as described below, the count-mode parameter will be set as follows:

- 0 – this channel is not the master timer
- 1 – count to a monitor preset – *mcount()*
- 2 – count to a time preset – *tcount()*
- 3 – just count until the counters are halted – *move_cnt*

Possible keys are as follows:

"prestart_all" – Sent to the controller (*mne* set to ".") before any counters are started. *p1* will contain the count preset, either in seconds or monitor counts. *p2* contains the count mode (1, 2 or 3), as in the above list. The fifth argument will contain the unit number of the controller.

"start_one" – Sent to start each counter at the start of the counting period. The function will be called for the regular counting channels before being called for the master timer channel. *p1* is the count time in seconds if counting to time or in counts if counting to monitor. If *mne* is the master timer, *p2* will contain 1, 2 or 3 to indicate the counting mode. If *mne* is not the master, *p2* will be zero.

"get_status" – Sent after the counters have been started, but only to the master timer/counter, if there is one, or to any counters configured as polled (with type MAC_CNTP). Must return nonzero if the channel is busy or zero if the counting is finished.

"counts" – Called when spec's built-in *getcounts* function is executed. The macro function must return the current counts for scaler channel number *mne*. Note, all the scalers channels associated with real counters will have been read before the macro function call, so the values in the *S[]* counter array will be current for the non-macro counters.

"halt_all" – Sent to each macro counter controller (*mne* set to ".") when counting is halted. The key is sent before the "halt_one" keys are sent. The third argument will contain the unit number of the controller.

"halt_one" – Sent to each active counter when counting is halted. The *p1* argument will be zero if called at the end of normal counting and one if called when ^C is typed at the keyboard or when a *stop()* or *sync* command is entered or after motors have stopped with a *move_cnt*.

Commands to halt counters are sent when a preset count time elapses, if a $\wedge C$ is typed from the keyboard, when a `stop()` or `sync` command is encountered from user-level or after motors have stopped when the `move_cnt` command is used.

The `_par()` Function For Counters

There are currently no built-in keywords that produce calls to the `_par()` macro function for counters. The macro function will only be called when `spec`'s user-level `counter_par()` function is called to set or retrieve an otherwise unrecognized parameter.

The function will be called as:

`prefix_par(mne, key, "get")` – The function should return a value for the user-defined parameter named as `key` for counter number `mne`.

`prefix_par(mne, key, "set", p1)` – Called to set the user-defined parameter `key` to `p1` for counter number `mne`.

The `mne` argument will always be a counter number and never the string `..`.

Built-in arguments to `counter_par()` that can't be passed to a user-defined macro function include `"disable"`, `"unit"`, `"channel"`, `"responsive"`, `"controller"`, `"device_id"`, `"scale"`, `"monitor"`, and `"timer"`.

MCA's

The `_cmd()` Function For MCA Devices

The `_cmd()` macro function for MCA devices is called in response to `spec`'s MCA and counting-related built-in functions and commands: `mca_get()`, `mca_put()`, `tcount()`, `mcount()`, `move_cnt`, `wait()`, `stop()` and `sync`. The `_cmd()` macro function will also be called in response to the standard `mca_par()` commands `"clear"`, `"run"`, and `"halt"`.

The syntax of the function call is:

`prefix_cmd(num, key [, p1 [, p2 [, data]])` – Called by the C code for operations related to MCA control. `num` is the MCA number. `key` is a string containing the particular command. `p1` and `p2`, if present, are parameters related to the command. For the `mca_get()` and `mca_put()` functions, `data` will be a data array for sending or receiving the MCA data.

None of the keys are mandatory. Possible keys are as follows:

`"clear"` – Sent in response to `mca_par("clear")` and before `"run"` when `"auto_clear"` mode is enabled. There are no additional parameters.

`"run"` – Sent in response to `mca_par("run")` and at the start of counting associated with the standard `tcount()`, `mcount()` and `move_cnt` commands when `"auto_run"` mode is enabled. If `"soft_preset"` mode is enabled, `p1` will be the counting preset (seconds or monitor counts), otherwise `p1` will be the value set with the `mca_par()` `"preset"` option. `p2` will be the count mode, as follows:

- 1 - called by `tcount()`
- 2 - called by `mcount()`
- 3 - called via `move_cnt`
- 4 - called by `mca_par("run")`

`"halt"` – Sent in response to `mca_par("halt")` and at the end of counting associated with the standard `tcount()`, `mcount()` and `move_cnt` commands when `"auto_run"` mode is enabled. Not called at the normal end of counting if `"soft_preset"` is enabled or if the `mca_par()` `"preset"` value is nonzero. The `p1` argument will be zero if called at the end of normal counting and one if called when $\wedge C$ is typed at the keyboard or when a `stop()` or `sync` command is entered or after motors have stopped with a `move_cnt`.

"get_status" – Called during counting. A return value of one will indicate the MCA is still busy.

"read" – Sent to read data. *p1* and *p2* are the first and last channels to be read, respectively, reflecting the optional first and last channel arguments to `mca_get()`. *data* is a data array of the native type (configured with the `_config()` call) where the data should be written. The optional return value is the number points actually read. The value will be used by `mca_get()` to determine how many points to copy to the returned array. If no value is returned, `mca_get()` will use the requested number of points.

"write" – Sent to write data. *p1* and *p2* are the first and last channels to be write, respectively, reflecting the optional first and last channel arguments to `mca_write()`. *data* is a data array of the native type (configured with the `_config()` call) from where the data can be read. The optional return value is the number points actually written.

The `_par()` Function For MCA Devices

The `_par()` macro function is called when various MCA parameters are set, and when the `mca_par()` or `mca_spar()` functions are used to retrieve a user-defined parameter. The *num* argument is the MCA number.

The standard `mca_par()` commands "info", "chans", "max_chans", "auto_run", "soft_preset", "auto_clear", "native_type", "preset", and "chan#", are handled by `spec`'s built-in code and will not generate calls to the `_par()` macro function. The standard `mca_par()` "disable" call will be passed to the `_par()` function when setting or clearing the disabled state.

The function will be called as:

`prefix_par(num, key, "get")` – The function should return a value for the user-defined parameter named as *key* for MCA number *num*.

`prefix_par(num, key, "set", p1)` – Called to set the user-defined parameter *key* to *p1* for MCA number *num*.

MCA Counting Mode Summary

The `mca_par()` "auto_run" mode must be enabled for the standard `spec` counting commands to control the MCA. If the `mca_par()` "soft_preset" mode is enabled, when the `_cmd()` function is called with the "run" argument, *p1* will be the count time, seconds when counting to time and counts when counting to monitor. When in this mode, the macro hardware MCA will be polled to determine when it is finished. If the MCA takes longer to count to the preset than the master timer, `spec` will wait until the MCA has finished. Unless the counting is aborted, The "halt" command will not be called, as the macro hardware MCA is expected to count until it reaches its programmed preset value. The *p1* argument will be zero for powder-mode counting with the `move_count` command, and the MCA will be sent a "halt" command when `spec` senses counting has finished due to the designated motor having completed its trajectory.

If "soft_preset" is disabled, when the `_cmd()` function is called with the "run" argument *p1* will be the value set using `mca_par()` "preset", or zero if the value has not been set. If the value is nonzero, `spec` will poll the MCA until it returns a non-busy status. Thus one can have the macro hardware MCA count to a different preset than the master timer. If the preset value is zero, the MCA will be sent a halt command when the master timer reaches the end of the count interval.

In all cases, a count abort will generate a "halt" command.