

**NAME**

mca – Multichannel Analyzer Support

**INTRODUCTION**

In `spec`, an MCA-type device is something that returns a one-dimensional array of data. Such devices include multichannel analyzers, multichannel scalers, and digital correlators. Operation of most of these devices is through either manual commands to start and stop acquisition when appropriate to the experiment or automatic operation with the standard counting commands and macros.

Currently supported MCA-type devices include:

- FAST ComTec MCD/PC (ISA Board)
- FAST ComTec MCDLAP (ISA Board)
- Hecus ASA-32 MCA (ISA Board)
- Oxford/Tennelec/Nucleus MicroFast MCA (ISA Board)
- Oxford/Tennelec/Nucleus PCA-3 MCA (ISA Board)
- Oxford/Tennelec/Nucleus PCA II MCA (ISA Board)
- Ortec TRUMP 8K/2K Multichannel Buffer (ISA Board)

- Ortec TRUMP PCI (PCI Board)
- Ortec MCS-PCI (PCI Board)

- ESRF MUSST MCA (GPIB)
- Princeton Applied Research Model 283 (GPIB)
- Princeton Instruments ST116 PDA (GPIB)
- Oxford/Tennelec/Nucleus PCA Multiport (GPIB)
- Ortec 918A Multichannel Buffer (GPIB)
- Silena 7328 MCA (8K) (GPIB)
- Silena 7329 MCA (16K) (GPIB)

- Amptek Pocket MCA 8000/8000A (Serial)
- Amptek PX4 Digital Processor (Serial)
- MBraun PSD-50M (Serial)
- Bruker (Roentec) XFlash MAX MCA (Serial)

- XIA X10P MCA (parallel port)

- DSP 2190 MCS Averager (CAMAC)
- LeCroy 2301 Interface for qVT MCA (CAMAC)
- LeCroy 3512 Spectroscopy ADC (CAMAC)
- LeCroy 3521A Multichannel Scaler (CAMAC)
- LeCroy 3588 Fast Histogram Memory (CAMAC)
- XIA DXP (CAMAC)

- HasyLab Model 8701 MCA (VME)

- EPICS MCA Record
- TACO (ESRF) MCA Device Server

- AI Solutions DAQ-ATDC/NDAQ (USB)
- Amptek PX4 Digital Processor (USB)
- Canberra Multiport II (USB)
- XIA Saturn MCA (USB)

- Canberra LYNX (Ethernet)
- Canberra Multiport II (Ethernet)
- Dectris Mythen 1K (Ethernet)

Laboratory Equipment Corporation NT-2400 (Ethernet)  
Seiko/EG&G Orsim MCA 7700 (Ethernet)

Brookhaven Instruments BI-9000 Autocorrelator (ISA Board)  
Nicomp TC-100 Autocorrelator (RS-232C)

Keithley 2001 Multimeter (GPIB)

All MCA-type devices (except CAMAC) are assigned a device number from 0 to 31 in the *config* file. *spec*'s MCA functions `mca_get()`, `mca_put()` and `mca_par()` operate on the MCA device last selected with the `mca_sel()` function. The functions `mca_sget()`, `mca_sput()` and `mca_spar()` take the assigned device number as an additional initial parameter.

Currently, one cannot assign arbitrary device numbers to CAMAC MCA-type devices. Instead the numbers are automatically assigned starting with the lowest available device number not otherwise taken when *spec* reads the *config* file. Use the `mca_sel("?")` to see what device numbers have been assigned.

### DATA ARRAYS AND DATA GROUPS

The `mca_get()` and `mca_put()` functions, which transfer data between the MCA-type hardware and *spec*, can use either data arrays or data groups. (Data groups are an older method of handling arrays of numbers in *spec* and are explained in the *data* help file.)

The recommended method of handling MCA data is with data arrays. The *arrays* help file explains general features of the array data type. In brief, the arrays are declared with a type and dimension. For example,

```
SPEC.1> ulong array mca_data[8192]
```

declares an unsigned-long data array of 8,192 elements. The `mca_get()` and `mca_put()` functions can be invoked with the array name as an argument. If the declared type of the array matches that of the hardware, the data will be transferred directly to the array. If the type doesn't match, *spec* will create a temporary buffer of the correct type, read the data into that, then copy the data to the user's array, all transparently to the user. Either way will work, though there is a slight efficiency advantage in declaring the array to match the native type of the hardware. The command

```
SPEC.2> print mca_par("native_type")
long
```

shows the native type of the selected MCA device.

### FUNCTIONS

Not all functions or `mca_par()` options are implemented for all MCA devices. Some functions can't be implemented due to hardware limitations. Others just haven't been implemented yet. If the hardware-specific documentation doesn't yet exist, contact CSS to determine which features are implemented for particular devices.

`mca_sel(n)` – Selects which MCA-type device to use for subsequent calls of the `mca_get()`, `mca_put()` and `mca_par()` functions. The MCA device numbering is set in the *config* file with the string `@mca_N`, where *N* is the device number (from 0 to 31). Returns `-1` if not configured for device *n*, otherwise returns zero. It is not necessary to use `mca_sel()` if only one MCA-type device is configured and is configured as device 0.

`mca_sel("?")` – Lists the configured MCA devices and indicates which device is currently selected for the `mca_get()`, `mca_put()` and `mca_par()` functions with an asterisk. Also displays the MCA device number for use with the `mca_sget()`, `mca_sput()` and `mca_spar()` functions and displays whether the presence test found the unit

unresponsive or if the user has disabled the unit. Returns the total number of MCA devices recognized in the *config* file. Note, the value of the built-in global variable *MCAS* is always set to the number of MCA devices (as of *spec* release 5.02.02-7).

`mca_sel(n, "?")` – Returns a string containing one line of information about MCA device *n*, or 0 if *spec* isn't configured for device *n*.

`mca_get(arr [, roi_beg [, roi_end]])` – Gets data from the currently selected MCA-type device, and transfers it to the array *arr*. If the optional starting channel and ending channel are given, the data is read from those hardware channels and placed starting at the beginning of the array. For example,

```
SPEC.1> ulong array data[1024]
SPEC.2> mca_get(data)
SPEC.3> mca_get(data, 32, 128)
SPEC.4> mca_get(data[32:128], 32, 128)
```

The last example uses subarray syntax to position the data in array elements corresponding to the MCA channel positions.

`mca_get(g, e [, roi_beg [, roi_end]])` – As above, but transfers the data to element *e* of data group *g* instead of an array. Returns the number of points transferred.

`mca_sget(sel, ...)` – Like the above functions, but uses the MCA device numbered *sel* in the *config* file.

`mca_put(...)` and `mca_sput(sel, ...)` – These functions have the same syntax as the above, but transfer data to the MCA device (for devices that support transfer in that direction).

`mca_par(cmd [, arg])` – A function to access various features and parameters of the currently selected MCA device. The string *cmd* selects an option. The argument *arg* contains an optional value. Some values for *cmd* apply to all MCA devices, while some apply only to certain devices.

`mca_spar(sel, cmd [, arg])` – Like the above, but uses the MCA device numbered *sel* in the *config* file.

`mca_par("info")` – Displays the native type, the currently configured number of channels and the maximum number of hardware channels for the selected MCA device.

`mca_par("chans")` – Returns the currently configured number of channels.

`mca_par("max_chans")` or `mca_par("max_channels")` – Returns the maximum number of hardware channels.

`mca_par("disable" [, arg])` – With no arguments, returns nonzero if the selected MCA device has been disabled by the user, otherwise returns zero. If *arg* is 1, disables the MCA. If *arg* is 0, turns off the disabled mode. When the device is disabled, *spec* will not access the hardware. On startup, and after the standard *config* macro or the *reconfig* command is run, disabled mode is off.

`mca_par("auto_run" [, arg])` – With no arguments, returns nonzero if the selected MCA device has auto-run mode set, otherwise returns zero. If *arg* is 1, enables auto-run mode. If *arg* is 0, turns off auto-run mode. When auto-run mode is set the device is started and stopped with the counting functions `tcount()`, `mccount()`, etc. When not set, the counting functions are ignored, but the device can be controlled with the "run" and "halt" options described below. In addition, the device can be halted with the `stop()` function and will be halted with `^C`. Some devices default to auto-run mode on and some to auto-run mode off on startup and after the standard *config* macro or the *reconfig* command is run.

`mca_par("soft_preset" [, arg])` – With no arguments, returns nonzero if the selected MCA device has soft-preset mode set, otherwise returns zero. If *arg* is 1, enables soft-preset mode. If *arg* is 0, turns off soft-preset mode. When set, and if auto-run mode is enabled (see above), the MCA device is set to count for the time preset given as an argument to the `tcount()` function. The `wait()` function will wait until both the timer and the MCA device have counted to their respective presets. When soft-preset mode is not set, but auto-run mode is, the device is programmed for continuous data acquisition and will be stopped when the timer's preset is reached. Currently, when both soft-preset and auto-run modes are in effect, when counting to monitor (using `mcount()`) rather than to time, the device is also programmed for continuous data acquisition, as above.

`mca_par("auto_clear" [, arg])` – With no arguments, returns nonzero if the selected MCA device has auto-clear mode set, otherwise returns zero. If *arg* is 1, enables auto-clear mode. If *arg* is 0, turns off auto-clear mode. When set, `spec` automatically sends commands to clear the MCA data before acquisition is started. Note, for some MCA devices, the clear operation takes time, so may affect the duration of data acquisition when not in soft-preset mode. Auto-clear mode is set at start up and after each hardware reconfiguration. Auto-clear mode is currently implemented for the XIA DXP; Nucleus PCA II, PCA-3, Multiport and Microfast; Ortec Trump MCA (ISA and PCI) and MCS (PCI); Fast Comtec MCD; Seiko EG&G MCA 7700; MBraun PSD-50M; Silena 7328 and 7329; Roentec XFlash MAX; Amptek 8000 and 8000A; Labo 2400; the EPICS MCA record and the TACO (ESRF) MCA device server.

`mca_par("native_type")` – Returns one of the strings `byte`, `ubyte`, `short`, `ushort`, `long`, `ulong`, `float`, `double` or `unknown` to describe the native data type of the MCA device.

`mca_par("preset" [, arg])` – With no arguments, returns the current preset value. Otherwise, sets the preset to *arg*. The preset value is in seconds for live-time and real-time modes or number of counts for integral mode. If the preset is zero, the device will be programmed for continuous run, except when the soft-preset mode is in effect, as described above.

`mca_par("live")` – Sets the MCA device to live-time mode.

`mca_par("real")` – Sets the MCA device to real-time (or true-time) mode.

`mca_par("integral")` – Sets the MCA device to a mode where it runs until the counts in a region of interest reach the preset.

`mca_par("elapsed_live")` – Returns the elapsed live time in seconds.

`mca_par("elapsed_real")` – Returns the elapsed real time in seconds.

`mca_par("dead")` – Returns the dead-time percentage, calculated as  $100 \times (\text{real\_time} - \text{live\_time}) / \text{real\_time}$ .

`mca_par("run")` – Programs the MCA device with the appropriate parameters and starts acquisition.

`mca_par("halt")` – Halts the MCA device.

`mca_par("clear")` – Clears the data in the current group.

`mca_par("group_size")` – Returns the current group size.

`mca_par("group_size", size)` – Sets the group size to *size*.

`mca_par("select_group")` – Returns the currently active group. Groups are numbered starting at zero.

`mca_par("select_group", group)` – Sets the active group to *group*.

`mca_par("gain")` – Returns the current gain value used in pulse-height analysis mode.

`mca_par("gain", value)` – Sets the gain parameter to *value*.

`mca_par("send", msg)` – For message-based devices, sends the string *msg* to the device.

`mca_par("read", msg)` – For message-based devices, sends the string *msg* to the device, and returns the device's reply.

`mca_par("chan#")` – Returns the contents of channel number #.

`mca_par("chan#", value)` – Set the contents of channel number # to *value*.

`mca_par("set_hdw_roi", beg, end)` – Sets the hardware region of interest to the channels from *beg* to *end*. Several calls may be made to select noncontiguous regions.

`mca_par("clr_hdw_roi")` – Clears the hardware region of interest.

**SEE ALSO**

*amptek dxp hecus kisim lc2301 lc3588 mcspci microfast MPII mythen nicomp nt2400 pca px4*