`mda-load` library v1.4.1 manual

Dohn Alexander Arms
`dohnarms@anl.gov`

August 12, 2016

This library is used to load MDA files created by the **saveData** program that is part of **EPICS**. It will load files with MDA versions of 1.2, 1.3, and 1.4, which are the only versions used currently.

1 Compiling

Compiling requires that you specify the **mda-load** library. This can be done directly by specifying the filename `libmda-load.a` and its path. If the library is in a standard search location for libraries, you can use the "`-l mda-load`" option with **gcc** or **ld**.

If you are using Solaris, you will also need to specify the Networking Services Library (**nsl**). This can be done with the "`-l nsl`" option.

2 Code

An MDA file structure has a defined structure that is mimicked by the data structures this library uses. There first is a header describing the scan(s), followed by the highest dimensional scan, then all the lower dimensional scans, and finally by an optional section of extra PVs.

2.1 Functions

There are functions for loading the entire file or just parts of it. The function and structure definitions are in the include file `mda-load.h`.

In the following functions, `fptr` is a `FILE` pointer corresponding to an open MDA file, normally gotten from `fopen()` and removed by `fclose()`. It doesn't matter what position the pointer is at in the file when calling these functions.

`struct mda_file *mda_load( FILE *fptr)`
This function loads the entire MDA file at once, returning a pointer to an `mda_file` structure.

`void mda_unload( struct mda_file *mda)`
This will free the memory occupied by the `mda_file` to which `mda` points.

`struct mda_header *mda_header_load( FILE *fptr)`
This function loads the MDA file header, returning a pointer to a `mda_header` structure.

`void mda_header_unload( struct mda_header *header)`
This will free the memory occupied by the `mda_header` to which `header` points.

`struct scan *mda_scan_load( FILE *fptr)`
This function loads the entire scan structure (all dimensions) for an MDA file, returning a pointer to a `mda_scan` structure. It is the same as `mda_subscan_load()`, where `depth` is zero and `recursive` is one.

`struct mda_scan *mda_subscan_load( FILE *fptr, int depth, int *indices, int recursive)`
This function loads only a part of an MDA file's scans, returning a pointer to the resulting `mda_scan` structure. The `depth` refers to how many dimensions down into the scan the subscan is located (it has to be

less than the number of dimensions in the file), and to the number of members in `indices`. The `indices` array contains the location of the subscan, starting with the highest dimensional index; if `depth` is zero, `indices` can be set to `NULL`. The `recursive` flag determines whether the lower dimensional scans will also be read (if any exist).

`void mda_scan_unload( struct mda_scan *scan)`
This will free the memory occupied by the `mda_scan` to which `scan` points.

`struct mda_extra *mda_extra_load( FILE *fptr)`
This function attempts to load the extra PVs for an MDA file. If they exist, it returns a pointer to an `mda_extra` structure, otherwise it returns `NULL`.

`void mda_extra_unload( struct mda_extra *extra)`
This will free the memory occupied by the `mda_extra` to which `extra` points. It is safe to call this function if `extra` is set to `NULL`.

`struct mda_file *mda_update( FILE *fptr, struct mda_file *previous_mda)`
This function is the same as `mda_load()`, except that it will attempt to use the data from a previous `mda_load()` or `mda_update()` to reduce the amount of data that needs to be read. This is intended for use when an MDA file is being read while the scan is in progress, allowing only the newly written part of the file to be read, reducing the CPU load. The `previous_data` pointer that is passed needs to be either point to a valid `struct mda_struct` or is NULL. If `previous_data` is not `NULL`, its value after the function call is invalid, even if the function returned a `NULL` due to an error; the data to which it pointed was either taken by the newly returned `struct mda_struct` or was released from memory, .

2.2 Example

The simplest way to use this library is to simply load the entire MDA file, using the template below. Then there are only two functions needed, `mda_load()` and `mda_unload()`, with the rest of the work coming from accessing the MDA data structure.

```
#include "mda-load.h"
...
FILE *fptr;
struct mda_file *mda;
...
if( (fptr = fopen(filename, "r")) == NULL)
    exit(1);
if( (mda = mda_load(fptr)) == NULL)
    exit(1);
fclose(fptr);
...
/* Access the mda structure here. */
...
mda_unload(mda);
...
```

3 Data

Once you have a pointer to the allocated `mda_file`, you can access its members directly. The definition of the various data structures are in `mda-load.h`. Assuming that the data pointer is called `mda`, here's how you access the data.

The code takes advantage of the C99 integer data types in `stdint.h`, which allows explicitly declaring the size of an integer. Any of the `int8_t`, `int16_t`, or `int32_t` types can simply be used as an `int` if it makes your code simpler (such as using `%i` with printf).

3.1 Header

```
(struct mda_header *) mda->header
      (float)   header->version
      (int32_t) header->scan_number
      (int16_t) header->data_rank
      (int32_t) header->dimensions[n] , [n] = [0] to [data_rank - 1]
      (int16_t) header->regular
      (int32_t) header->extra_pvs_offset
```

This section contains the global data values for the MDA file. `version` signifies the MDA format version, normally 1.3. `scan_number` is the number assigned by **saveData** to the scan. `data_rank` show the number of dimensions to the scan (for a 3-D scan, this is 3). The `dimensions` array (with `data_rank` elements) contains the number of elements for each dimension of the scan, starting with the highest dimensional scan; for a 4-D scan, `dimensions[1]` is the number of elements to the 3-D scans. `regular` signifies whether the dimensions of any of the scans were changed while the overall scan was running; a nonzero value signifies that the dimensions array is not totally correct, and the number of requested points needs to be checked in each scan. `extra_pvs_offset` gives, for the section of extra PV's, the offset in bytes from the beginning of the file; if that section does not exist, this will be `0`.

3.2 Scans

```
(struct mda_scan *) mda->scan
      (int16_t)   scan->scan_rank
      (int32_t)   scan->requested_points
      (int32_t)   scan->last_point
      (int32_t *) scan->offsets
      (char *)    scan->name
      (char *)    scan->time
      (int16_t)   scan->number_positioners
      (int16_t)   scan->number_detectors
      (int16_t)   scan->number_triggers
      (struct mda_positioner *) scan->positioners[n] ,
                                [n] = [0] to [scan->number_positioners - 1]
            (int16_t) positioners[n]->number
            (char *)  positioners[n]->name
            (char *)  positioners[n]->description
            (char *)  positioners[n]->step_mode
            (char *)  positioners[n]->unit
            (char *)  positioners[n]->readback_name
            (char *)  positioners[n]->readback_description
            (char *)  positioners[n]->readback_unit
      (struct mda_detector * scan->detectors[n] ,
```

```
                          [n] = [0] to [scan->number_detectors - 1]
      (int16_t) detectors[n]->number
      (char *)  detectors[n]->name
      (char *)  detectors[n]->description
      (char *)  detectors[n]->unit
(struct mda_trigger *) scan->triggers[n] ,
                          [n] = [0] to [scan->numbers_triggers - 1]
      (int16_t) triggers[n]->number
      (char *)  triggers[n]->name
      (float)   triggers[n]->command
(double *) scan->positioners_data[n] , [n] = [0] to [scan->number_positioners - 1]
      (double) scan->positioners_data[n][m] ,
                          [m] = [0] to [scan->requested_points - 1]
(float *) scan->detectors_data[n] , [n] = [0] to [scan->number_detectors - 1]
      (float) scan->detectors_data[n][m] ,
                          [m] = [0] to [scan->requested_points - 1]
(struct mda_scan **) scan->sub_scans
```

This section includes the scan data. It is also recursive in nature due to it being able to handle arbitrary dimensions.

3.2.1 Structure

The overall structure for multidimensional files is dictated by `scan_rank` and `sub_scans`. As long as `scan_rank` is greater than one, `sub_scans` will not be `NULL` and will contain an array of the next lower dimensional scans (it will be `NULL` if `mda_subscan_load()` was used to retrieve the scan when the `recursive` parameter set to zero). For a multidimensional scan, this takes the form of a tree, since each sub-scan can also have its own sub-scans. For a higher dimensional scan, the values for the positioners and detectors apply to all scan with its `sub_scans`.

Suppose a 5×8×20 scan, where you want to access the $(3, 7, x)$ 1-D scan, you would access it as `mda->scan->sub_scans[2]->sub_scans[6]`. However, if the scan was aborted, `mda->scan->sub_scans[2]` or `mda->scan->sub_scans[2]->sub_scans[6]` *might* be `NULL`, depending on where the scan was aborted. Using `last_point` can let you know the last "officially valid" sub-scan is `sub_scans[last_point - 1]`. The reason that I say "officially valid" is that another scan *might exist* at `sub_scans[last_point]`, as it was the scan in progress that was aborted; one can use this data, but should take care.

3.2.2 Variables

As described before, `scan_rank` is the dimensionality of this scan. `requested_points` is how many points were wanted, while `last_point` tells how many actually were finished. `offsets` is an array of `requested_points` members, showing the distance from the beginning of the MDA file to the subscans; if the value is zero, then that scan does not exist. `name` is the name of the scanner in **EPICS**, while `time` is when this particular scan was started.

`number_positioners` tells how many positioners are moved as part of this scan. The `positioners` array, holding `number_positioners` members, has a description of each positioner and its readback. `number` is the internal number the **scanRecord** uses to identify this positioner, while `name` is what its called, and `description` describes it. `step_mode` is how the scan determined what step to use: it can be *linear*, where the spacing between steps is equal; *table*, where the step positions are read from an array; or *fly*, where the step positions are read back during an on-the-fly scan. `unit` is the associated unit of the positioner. Similarly, for the readback, there is `readback_name`, `readback_description`, and `readback_unit`.

The detector information is very similar to the positioners, as there is a `detectors` array with `number_detectors` elements. For each detectors, there is also a `number`, `name`, `description`, and `unit`.

The trigger information is again similar to the positioners, with a `triggers` array with `number_triggers` elements. Each trigger has a `number` and `name` associated to it, as well as a `command`, which is a value sent to `name` to trigger.

The positioner data values are held in an two dimensional array named `positioners_data`. Since one can't allocate a two dimensional array directly, it's actually an array of pointers (corresponding to each detector), pointing to arrays of more pointers (corresponding the the data). To access the 8th data point of the 12th detector, one would type `(scan->positioners_data[11])[7]`; the parentheses are not optional.

The detector data values, in `detectors_data`, is accessed similarly to the positioner data values.

The `sub_scans` variable is used for accessing lower dimensional scans (if they exist). It's described in Sec. 3.2.1.

3.3 Extra PV's

```
(struct mda_extra *) mda->extra
      (int16_t) extra->number_pvs
      (struct mda_pv *) extra->pvs[n] , [n] = [0] to [number_pvs - 1]
            (char *)  pvs[n]->name
            (char *)  pvs[n]->description
            (int16_t) pvs[n]->type
            (int16_t) pvs[n]->count
            (char *)  pvs[n]->unit
            (void *)  pvs[n]->values
```

This section, which doesn't always exist (signified by `extra` being `NULL`), contains extra PV's recorded during the scan. `number_pvs` is the number of PV's contained, with the PV's being held in an array `pvs`.

For each PV, there is the `name` string and `description` string. `type` lets you know what kind of data type it is, with the correspondence seen in Table 1. If `type` isn't `EXTRA_STRING`, `count` gives the number of elements to the array and `unit` string gives the unit for the values. The values themselves are held in an array `values`.

Table 1: Extra PV data type

`type` name	`type` value	C type	Description
`EXTRA_PV_STRING`	0	`(char *)`	zero-terminated string
`EXTRA_PV_INT8`	32	`(int8_t *)`	8-bit integer array
`EXTRA_PV_INT16`	29	`(int16_t *)`	16-bit integer array
`EXTRA_PV_INT32`	33	`(int32_t *)`	32-bit integer array
`EXTRA_PV_FLOAT`	30	`(float *)`	floating-point array
`EXTRA_PV_DOUBLE`	34	`(double *)`	double-precision floating-point array

Accessing the values is done by setting pointer `values` to the correct type, according to `type` and Table 1. Suppose the third extra PV was of type `EXTRA_DOUBLE`, and you wanted to access its fifth member, this could be done using `((double *) pvs[2]->values)[4]`.

4 Basic Information Routines

It is possible to load only the basic information of an MDA file, saving load time and memory. This information includes everything in the header, as well as the detector, positioner, and trigger descriptions for each scan dimension. As this information is taken from the first scan of each dimension, there is an assumption that every other scan is correctly represented by the first scan of its dimensionality. This also means that for a multidimensional file, not all of the file is checked for errors; this can be done with `mda_test()`, which will cost some additional time.

4.1 Functions

`int mda_test( FILE *fptr)`
This function will check a file's integrity by loading the entire file structure (but not keeping all the information in memory to keep memory usage minimal), using the same error checking as `mda_load()`. A returned value of `0` means that there was no error, while `1` means there was an error.

`struct mda_fileinfo *mda_info_load( FILE *fptr)`
This function loads a fileinfo of the MDA file, returning a pointer to an `mda_fileinfo` structure. It does not check the validity of the entire file, which can be done with `mda_test()`.

`void mda_info_unload( struct mda_fileinfo *fileinfo)`
This will free the memory occupied by the `mda_fileinfo` to which `fileinfo` points.

4.2 Structure

```
(struct mda_fileinfo *) fileinfo
      (float)   fileinfo->version
      (int32_t) fileinfo->scan_number
      (int16_t) fileinfo->data_rank
      (int32_t) fileinfo->dimensions[n] , [n] = [0] to [data_rank - 1]
      (int16_t) fileinfo->regular
      (int32_t) fileinfo->last_topdim_point
      (char *)  fileinfo->time
      (struct mda_scaninfo *) fileinfo->scaninfos[n] , [n] = [0] to [data_rank - 1]
            (int16_t) scaninfos[n]->scan_rank
            (int32_t) scaninfos[n]->requested_points
            (char *)  scaninfos[n]->name
            (int16_t) scaninfos[n]->number_positioners
            (int16_t) scaninfos[n]->number_detectors
            (int16_t) scaninfos[n]->number_triggers
            (struct mda_positioner *) scaninfos[n]->positioners[n] ,
                                  [n] = [0] to [scan->number_positioners - 1]
                  (int16_t) positioners[n]->number
                  (char *)  positioners[n]->name
                  (char *)  positioners[n]->description
                  (char *)  positioners[n]->step_mode
                  (char *)  positioners[n]->unit
                  (char *)  positioners[n]->readback_name
                  (char *)  positioners[n]->readback_description
                  (char *)  positioners[n]->readback_unit
            (struct mda_detector * scaninfos[n]->detectors[n] ,
                                  [n] = [0] to [scan->number_detectors - 1]
                  (int16_t) detectors[n]->number
                  (char *)  detectors[n]->name
                  (char *)  detectors[n]->description
                  (char *)  detectors[n]->unit
            (struct mda_trigger *) scaninfos[n]->triggers[n] ,
                                  [n] = [0] to [scan->numbers_triggers - 1]
                  (int16_t) triggers[n]->number
                  (char *)  triggers[n]->name
                  (float)   triggers[n]->command
```

The structure of the `mda_fileinfo` structure is nonrecursive, and is thus easier to access. The variables `version`, `scan_number`, `data_rank`, `dimensions`, and `regular` are the same as described in Section 3.1. The variable `last_topdim_point` is the number of completed scans in the highest dimensional scan, while `time` is the data and time of the start of the overall measurement. The information for each dimension's scans are in the `scaninfos` array. The variables contained in each entry is a subset of those found in the earlier describes `scan`, and all are described in in Section 3.2.2.

5 Library Changes

Between version 0.3 and 1.0, the structure names were changed to be prefixed by `mda_`.

From 1.1 to 1.2, integers use types that denote how many bits they use: `char` (as an 8-bit integer) → `int8_t`, `short` → `int16_t`, and `long` → `int32_t`. The most likely change needed in user code for this is with `printf()`, where one would use `%d` instead of `%ld`, etc. The Extra PV data type names were also renamed, to have them start with `EXTRA_PV_` and have the integer length in the relevant names.