`mda-load` library manual

Dohn Alexander Arms
`dohnarms@anl.gov`

August 4, 2005

This library is used to load MDA files created by the **saveData** program that is part of **EPICS**.

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

To use do the actual loading of an MDA file, use the template below to do so. There are only two functions, `mda_load()` and `mda_unload()`, with the rest of the work comes from accessing the MDA data structure.

```
#include "mda-load.h"
...
FILE *fptr;
struct mda_struct *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_struct`, 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.

3.1 Header

```
(struct header_struct *) mda->header
      (float)  header->version
      (long)   header->scan_number
      (short)  header->data_rank
      (short)  header->dimensions[n] , [n] = [0] to [data_rank - 1]
      (short)  header->regular
      (long)   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; for a 3-D scan, `dimensions[1]` is the number of elements to the 2-D scans. `regular` signifies whether the dimensions of any of the scans were changed while the overall scan was running. `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 scan_struct *) mda->scan
      (short)  scan->scan_rank
      (short)  scan->requested_points
      (short ) scan->last_point
      (long *) scan->offsets
      (char *) scan->name
      (char *) scan->time
      (short)  scan->number_positioners
      (short)  scan->number_detectors
      (short)  scan->number_triggers
      (struct positioner_struct *) scan->positioners[n] ,
                                [n] = [0] to [scan->number_positioners - 1]
            (short)  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 detector_struct * scan->detectors[n] ,
                                [n] = [0] to [scan->number_detectors - 1]
            (short)  detectors[n]->number
            (char *) detectors[n]->name
            (char *) detectors[n]->description
            (char *) detectors[n]->unit
```

```
(struct trigger_struct *) scan->triggers[n] ,
                          [n] = [0] to [scan->numbers_triggers - 1]
      (short)  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 scan_struct **) 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. 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 $5\times8\times20$ scan, where you want to access the $(3, 7, x)$ 1-D scan, you would access it as `mda->scan->subscans[2]->subscans[6]`. However, if the scan was aborted, `mda->scan->subscans[2]` or `mda->scan->subscans[2]->subscans[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_scan[last_point - 1]`. The reason that I say "officially valid" is that another scan *might exist* at `sub_scan[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 extra_struct *) mda->extra
      (short) extra->number_pvs
      (struct pv_struct *) extra->pvs[n] , [n] = [0] to [number_pvs - 1]
            (char *) pvs[n]->name
            (char *) pvs[n]->description
            (short)  pvs[n]->type
            (short)  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 `DBR_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
DBR_STRING	0	(char *)	zero-terminated string
DBR_CTRL_CHAR	32	(char *)	byte array
DBR_CTRL_SHORT	29	(short *)	short integer array
DBR_CTRL_LONG	33	(long *)	long integer array
DBR_CTRL_FLOAT	30	(float *)	floating-point array
DBR_CTRL_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 `DBR_CTRL_DOUBLE`, and you wanted to access its fifth member, this could be done using `((double *) pvs[2]->values)[4]`.