`mdaload.m` v1.0 manual

Dohn Alexander Arms
dohnarms@anl.gov

August 17, 2016

This MATLAB/Octave code 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. It will check for errors in the data file upon loading.

Installing the code is a matter of putting the file `mdaload.m` in the correct place for MATLAB or Octave to find it.

The function in the code, `mda = mdaload(filename)`, returns a data structure that mimicks the native MDA file structure. There first is a header describing the multidimensional scan, followed by the highest dimension scan, then all the lower dimension scans, and finally by an optional section of extra PVs.

1 Data Structure

The returned data structure is described below. Unless specified as a string, all values are numbers.

```
mda.version  [float32]
mda.scan_number  [int32]
mda.data_rank  [int16]
mda.dimensions(n)  [int32] , n = 1:mda->data_rank
mda.regular  [int16]
mda.scan  [scan structure]
mda.extra  [extra structure]
```

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 requested number of elements for each dimension of the scan, starting with the highest dimensional scan; this is more of a guidance array, as in practice, the first dimension of the top scan may be smaller, and individual scans may have different dimensionality (making them irregular). `regular` signifies whether the dimensions of any of the scans were changed while the overall scan was running. The `scan` structure holds the data in the form of a scan tree, and the format is shown below. If `extra` is not empty, then are extra PVs present in the file, and the format of this structure is also below.

1.1 Scans

```
scan.scan_rank  [int16]
scan.requested_points  [int32]
scan.last_point  [int32]
scan.name  [char]
scan.time  [char]
scan.number_positioners  [int16]
scan.number_detectors  [int16]
scan.number_triggers  [int16]
scan.positioners(n) : n = 1:scan.number_positioners
     positioners(n).number  [int16]
     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  [char]
scan.detectors(n) , n = 1:scan.number_detectors
     detectors(n).number  [int16]
     detectors(n).name  [char]
     detectors(n).description  [char]
     detectors(n).unit  [char]
scan.triggers(n) , n = 1:scan.numbers_triggers
     triggers(n).number  [int16]
     triggers(n).name  [char]
     triggers(n).command
scan.positioners_data(n,m)  [float64] ,
          n = 1:scan.number_positioners, m = 1:scan.last_point
scan.detectors_data(n,m)  [float32] ,
          n = 1:scan.number_detectors, m = 1:scan.last_point
scan.sub_scans(n)  [scan structure] ,
          n = 1:scan.last_point if (scan->scan_rank > 1) else []
```

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

`scan_rank` is the dimension of this scan. `requested_points` is how many points were wanted, while `last_point` tells how many actually were finished. Due to how saveData saves MDA files, only the top-level scan can be an incomplete scan, as higher scan values are not written until lower scans are finished. `name` is the name of the scanner in **EPICS**, while `time` is when this particular scan was started.

If a scan has a `scan_rank` greater than one, `sub_scans` will contain an array of the next lower dimensional scans, of length `last_point`. For a multidimensional scan this creates a tree structure, since each sub-scan can also have its own sub-scans. For a scan, the values for its positioners and detectors apply to its `sub_scans` if they exist.

Table 1: Extra PV data type

`pvs(n).type`	Description
char	string
int8	8-bit integer array
int16	16-bit integer array
int32	32-bit integer array
float32	floating-point array
float64	double-precision floating-point array

`number_positioners` tells how many positioners are moved as part of this scan. The `positioners` array, of length `number_positioners`, holds structures describing the positioners and readbacks. `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 of length `number_detectors`. For each detector, there is also a `number`, `name`, `description`, and `unit`.

The trigger information is again similar to the positioners, with a `triggers` array of length `number_triggers`. 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 a two dimensional matrix named `positioners_data`, of size `last_point` by `number_positioners`. The detector data values are in a similar matrix `detectors_data`, of size `last_point` by `number_detectors`.

1.2 Extra PV's

```
extra.number_pvs  [int16]
extra.pvs(n) , n = extra.number_pvs
      pvs(n).name  [char]
      pvs(n).description  [char]
      pvs(n).type  [char]
      pvs(n).count  [int16]
      pvs(n).unit  [char]
      pvs(n).values  [varied]
```

This section, which might be empty, 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. `count` gives the number of elements in the array (if `type` is *char*, it's the string length) and `unit` string gives the unit for the values. The values themselves are held in an array `values`.