## Contents:

1 Building MACSio
   1.1 Installing via Spack .................................................. 1
   1.2 Installing Manually .................................................... 1

2 Using MACSio
   2.1 Summary of Key Command Line Arguments ........................... 3
   2.2 MACSio Command Line Examples ........................................ 5
   2.3 Weak Scaling Study Command-Line Example ........................... 6
   2.4 Strong Scaling Study Command-Line Example ........................ 7
   2.5 Assessing Performance Achieved by MACSio ........................... 10
   2.6 Validating Data MACSio Produces ..................................... 13

3 Welcome to MACSio Documentation
   3.1 Main ................................................................. 15
   3.2 Mesh and Field Data Generation ....................................... 16
   3.3 Plugins .............................................................. 22
   3.4 Parallel I/O Paradigms ............................................... 40
   3.5 Command Line Argument Parsing ..................................... 46
   3.6 Timing ............................................................... 49
   3.7 Logging ............................................................. 58
   3.8 Contributing to MACSio Documentation ............................... 64
MACSio’s design is described in detail in an accompanying design document.

A key aspect about MACSio’s design relevant to the installation process is that MACSio is composed of a growing list of I/O plugins. Each plugin implements one or more strategies for marshalling MACSio’s data object(s) between primary (volatile) and secondary (non-volatile) storage (e.g. memory and disk). An I/O strategy may involve the use of one or more strategy-specific I/O libraries. Therefore, each plugin often introduces additional, but optional, build dependencies. A given installation of MACSio can include all or just a subset of the available plugins. This is all determined at MACSio build time.

1.1 Installing via Spack

The easiest way to build MACSio may be to use spack. In theory, the steps involved are as simple as

```bash
% git clone https://github.com/spack/spack.git
% . spack/share/spack/setup-env.sh
% spack install macsio
```

Note, however, that these few steps may install a lot of software dependencies, many which you may not ordinarily expect. The operation may take more than an hour to complete and the resulting package installations will be hashed according to Spack’s hashing specifications which may impact any subsequent software development activity you intend to do with the resulting MACSio installation.

Nonetheless, Spack is often the simplest way to install MACSio.

The Spack installation of MACSio attempts to build all of MACSio’s plugins.

1.2 Installing Manually

MACSio’s only required dependency is json_cwx. MPI is not even a required dependency. However, building without MPI will in all likelihood result in a MACSio installation that isn’t very useful in assessing I/O performance at scale. Building without MPI, however, is useful in debugging basic functionality of an installation.
MACSio has several **optional** dependencies, each one associated with a given *plugin*. For example, the Exodus plugin requires the *Exodus* and *netCDF* libraries which may, in turn, require the *HDF5* library. The Silo plugin requires the *Silo* library and, optionally, the HDF5 library.

For dependencies associated with each plugin, see the *CMakeLists.txt* file in the plugin directory for telltale notes about plugin dependencies.

Once json-cwx and any optional dependencies have been successfully installed, CMake is used to build MACSio and any of the desired plugins (builds with silo by default)

```
% mkdir build
% cd build
% cmake -DCMAKE_INSTALL_PREFIX=[desired-install-location] \ 
    -DWITH_JSON-CWX_PREFIX=[path to json-cwx] \ 
    -DWITH_SILO_PREFIX=[path to silo] ..
% make
% make install
```

**NOTE: Some options for the cmake line:**

- **Build docs:** `-DBUILD_DOCS=ON`
- **Build HDF5 Plugin:** `-DENABLE_HDF5_PLUGIN=ON -DWITH_HDF5_PREFIX=[path to hdf5]`
- **Build TyphonIO Plugin:** `-DENABLE_TYPHONIO_PLUGIN=ON -DWITH_TYPHONIO_PREFIX=[path to typhonio]`
- **Build PDB Plugin:** `-DENABLE_PBD_PLUGIN=ON`
- **Build Exodus Plugin:** `-DENABLE_EXODUS_PLUGIN=ON -DWITH_EXODUS_PREFIX=[path to exodus]`

Although MACSio is C Language, at a minimum it must be linked using a C++ linker due to its use of non-constant expressions in static initializers to affect the static plugin behavior. However, its conceivable that some C++’isms have crept into the code causing warnings or outright errors with some C compilers. If you encounter such a situation, please file an issue for it so we are aware of the issue and can fix it.

Note that the list of available plugins is determined by which plugins are compiled and linked with MACSio’s main. So, plugins are determined at link-time, not run-time.
MACSio is probably very different from many other I/O benchmarking tools you may be familiar with. To orient yourself, it may be useful to read the first sections of the original design document.

In particular, in MACSio_speak, when we talk about I/O requests, request sizes, frequencies, etc., we speak about them in terms of the operations of a real application performing dumps of its mesh and field data.

By default, MACSio’s command-line arguments are designed to maintain constant per-task I/O workload as task count is varied. This means MACSio, by default, exhibits weak scaling behavior. This does not mean, however, that strong scaling scenarios cannot also be handled. It means only that extra work is involved in constructing command-line arguments to ensure a strong scaling goal is achieved if that is desired.

MACSio has a large number of command-line arguments. In addition, each plugin may define additional command-line arguments. Full documentation of all MACSio’s command-line arguments and their meaning can be obtained with the command

```
% ./macsio --help | more
```

Be ready for a lot of output!

Here, we will describe only some of the basic arguments necessary to do initial testing that MACSio is installed correctly and to scale to large sizes.

All command-line arguments specified after the keyword argument --plugin_args are passed to the plugin and not interpreted by MACSio’s main.

### 2.1 Summary of Key Command Line Arguments

MACSio has many command-line arguments and that number is only likely to grow with time. We discuss here, only those the most relevant to initially getting started running MACSio tests.

MACSio is different from I/O benchmarking tools because it constructs and marshals data as real data objects commonly used in scientific computing applications. All of its command-line arguments are designed in these terms and one has to understand how those choices effect I/O workload created by MACSio.
In the descriptions below, default values for all command-line arguments are indicated in square brackets.

**–interface** [*--interface %s [miftmpl]] Specify the name of the interface (e.g. plugin) to be tested. Use keyword *list* to print a list of all known interface names and then exit. Examples:

- Get list of available plugins

  % ./macsio --interface list
  List of available I/O-library plugins...
  "miftmpl", "hdf5", "siro", "typhonio"

- Use the HDF5 plugin for a given test

  % ./macsio --interface hdf5

**–parallel_file_mode** [*--parallel_file_mode %s %d [MIF 4]] Specify the parallel file mode. There are several choices. Not all parallel modes are supported by all plugins. Use ‘MIF’ for Multiple Independent File (MIF) mode and then also specify the number of files. Or, use ‘MIIFFPP’ for MIF mode and one file per task and where macsio uses known task count. Use ‘MIFOPT’ for MIF mode and let MACSio determine an optimum file count based on heuristics. Use ‘SIF’ for SIngle shared File mode. If you also give a file count for SIF mode, then MACSio will perform a sort of hybrid combination of MIF and SIF modes. It will produce the specified number of files by grouping tasks in the the same way MIF does, but I/O within each group will be to a single, shared file using SIF mode and a subsetted communicator. When using SIF parallel mode, be sure you are running on a true parallel file system (e.g. GPFS or Lustre).

**–part_type** [*--part_type %s [rectilinear]] Options are ‘uniform’, ‘rectilinear’, ‘curvilinear’, ‘unstructured’ and ‘arbitrary’. Generally, this option impacts only the I/O workload associated with the mesh object itself and not any variables defined on the mesh. However, not all I/O libraries (or their associated MACSio plugins) support all mesh types and when making comparisons it is important to have the option of specifying various mesh types.

**–part_dim** [*--part_dim %d [2]] Spatial dimension of mesh parts; 1, 2, or 3. In most cases, 2 is a good choice because it makes downstream visualization of MACSio data easier and more natural. While MACSio is designed such that we would not ordinarily expect I/O workload to be substantially different for different spatial dimensions, this isn’t always known to be true for all possible plugins ahead of time.

**–part_size** [*--part_size %d [80000]] Per-task mesh part size. This becomes the nominal I/O request size used by each task when marshaling data. A following **B** | **K** | **M** | **G** character indicates ‘B’ytes, ‘K’ilo-, ‘M’ega- or ‘G’iga- bytes representing powers of either 1000 or 1024 depending on the selected units prefix system. With no size modifier character, ‘B’ytes is assumed. Mesh and variable data is then sized by MACSio to hit this target byte count in I/O requests. However, due to constraints involved in creating valid mesh topology and variable data with realistic variation in features (e.g. zone- and node-centering), this target byte count is hit exactly for only the most commonly used objects and approximately for other objects.

**–avg_num_parts** [*--avg_num_parts %f [1]] The average number of mesh parts per task. Non-integral values are acceptable. For example, a value that is half-way between two integers, K and K+1, means that half the task have K mesh parts and half have K+1 mesh parts, a typical scenario for multi-physics applications. As another example, a value of 2.75 here would mean that 75% of the tasks get 3 parts and 25% of the tasks get 2 parts. Note that the total number of parts is this number multiplied by the task count. If the result of that product is non-integral, it will be rounded and a warning message will be generated.

**–vars_per_part** [*--vars_per_part %d [20]] Number of mesh variables on each part. This controls the number of I/O requests each task makes to complete a given dump. Typical physics simulations run with anywhere from just a few effectively to several hundred mesh variables. Note that the choice in mesh part_type sets a lower bound on the effective number of mesh variables marshaled by MACSio due to the storage involved for the mesh coordinate and topology data alone. For example, for a uniform mesh this lower bound is effectively zero because there is no coordinate or topology data for the mesh itself. This is also almost true for rectilinear
meshes. For curvilinear mesh the lower bound is the number of spatial dimensions and for unstructured mesh it
is the number of spatial dimensions plus $2^n$ number of topological dimensions.

**Note:** uniform vs. rectilinear not fully defined here.

- **–num_dumps** [--num_dumps %d [10]] Total number of dumps to marshal
- **–dataset_growth** [--dataset_growth %f [1]] A multiplier factor by which the volume of data will grow
  between dump iterations. If no value is given or the value is <1.0 no dataset changes will take place.
- **–meta_type** [--meta_type %s [tabular]] Specify the type of metadata objects to include in each main dump.
  Options are ‘tabular’ or ‘amorphous’. For tabular type data, MACSio will generate a random set of tables of
  somewhat random structure and content. For amorphous, MACSio will generate a random hierarchy of random
  type and sized objects.
- **–meta_size** [--meta_size %d %d [10000 50000]] Specify the size of the metadata objects on each task and
  separately, the root (or master) task (MPI rank 0). The size is specified in terms of the total number of bytes in
  the metadata objects MACSio creates. For example, a type of tabular and a size of 10K bytes might result in
  3 random tables; one table with 250 unnamed records where each record is an array of 3 doubles for a total of
  6000 bytes, another table of 200 records where each record is a named integer value where each name is length
  8 chars for a total of 2400 bytes and a 3rd table of 40 unnamed records where each record is a 40 byte struct
  comprised of ints and doubles for a total of 1600 bytes.
- **–compute_work_intensity** [--compute_work_intensity %d [0]] Add some compute workload (e.g. give
  the tasks something to do) between I/O dumps. There are three levels of ‘compute’ that can be performed as
  follows:
  - Level 1: Perform a basic sleep operation (this is the default)
  - Level 2: Perform some simple FLOPS with randomly accessed data
  - Level 3: Solves the 2D Poisson equation via the Jacobi iterative method
  This input is intended to be used in conjunction with –compute_time which will roughly control how much
  time is spent doing work between dumps.
- **–time_randomize** [--time_randomize [0]] Make randomization in MACSio vary from dump to dump within
  a given run and from run to run by using PRNGs seeded by time.
- **–plugin-args** [--plugin_args] All arguments after this sentinel are passed to the I/O plugin plugin.

### 2.2 MACSio Command Line Examples

- To run with Multiple Independent File (MIF) mode to on 93 tasks to 8 HDF5 files...
  ```
  mpirun -np 93 macsio --interface hdf5 --parallel_file_mode MIF 8
  ```
- Same as above but a Single Shared File (SIF) mode to 1 HDF5 file (note: this is possible with the same plugin
  because the HDF5 plugin in MACSio has been designed to support both the MIF and SIF parallel I/O modes.
  ```
  mpirun -np 93 macsio --interface hdf5 --parallel_file_mode SIF 1
  ```
- Default per-task request size is 80,000 bytes (10K doubles). To use a different request size, use –part_size. For
  example, to run on 128 tasks, 8 files in MIF mode where I/O request size is 10 megabytes, use
  ```
  mpirun -np 128 macsio --interface hdf5 --parallel_file_mode MIF_ 8 --part_size 10M
  ```
Here, the M after the 10 means either decimal Megabytes (Mb) or binary Mibibytes (Mi) depending on setting for --units_prefix_system. Default is binary.

- To use H5Z-ZFP compression plugin (note: we’re talking about an HDF5 plugin here), be sure to have the plugin compiled and available with the same compiler and version of HDF5 you are using with MACSio. Here, we demonstrate a MACSio command line that runs on 4 tasks, does MIF parallel I/O mode to 2 files, on a two dimensional, rectilinear mesh with an average number of parts per task of 2.5 and a nominal I/O request size of 40,000 bytes. The args after --plugin-args are to specify ZFP compression parameters to the HDF5 plugin. In this case, we use H5Z-ZFP compression plugin in rate mode with a bit-rate of 4.

```
env HDF5_PLUGIN_PATH=<path-to-plugin-dir> mpirun -np 4 ./macsio --interface hdf5 -
\rightarrow-parallel_file_mode MIF 2 --avg_num_parts 2.5 --part_size 40000 --part_dim 2 --
\rightarrow-part_type rectilinear --num_dumps 2 --plugin_args --compression zfp rate=4
```

where path-to-plugin-dir is the path to the directory containing libh5zzfp.{a,so,dylib}

### 2.3 Weak Scaling Study Command-Line Example

Suppose you want to perform a weak scaling study with MACSio in MIF parallel I/O mode and where per-task I/O requests are nominally 100 kilobytes and each task has 8 mesh parts.

All MACSio command line arguments remain the same. The only difference is the task count you execute MACSio with.

```
for n in 32 64 128 256 512 1024 2048 4096
do
  mpirun -np $n macsio --interface hdf5 --avg_num_parts 8 --part_size 100K --
\rightarrow-parallel_file_mode MIF 32
done
```

Now, the above example started with a task count of 32 and 32 files in MIF mode and kept the file count constant. It is conceivable that if you continued this study to larger and larger scales, you may also want the MIF file count to vary somewhat as well. Here is an example of doing that.

#### 2.3.1 Example

```
#!/bin/sh
n=32
def nfiles()
{
  if [[ $n -le 32 ]]; then
    echo 32
  elif [[ $n -le 8192 ]]; then
    echo 64
  elif [[ $n -le 65536 ]]; then
    echo 128
  else
    echo 256
  fi
}
while [[ $n -le 262144 ]]; do
...
```
MACSio, Release 1.1.0

(continued from previous page)

nf=$(nfiles $n)

```
> cmd="mpirun -np $n mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --
> parallel_file_mode MIF $nf"
```

echo $cmd

n=$(expr $n \* 2)
done

Which when run, results in the following sequence of MACSio command-lines.

```
mpirun -np 32 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 32
mpirun -np 64 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 128 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 256 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 512 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 1024 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 2048 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 4096 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 8192 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 16384 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 32768 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 65536 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 64
mpirun -np 131072 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 256
mpirun -np 262144 mcsio --interface hdf5 --avg_num_parts 8 --part_size 100K --parallel_file_mode MIF 256
```

Because the default mesh type is rectilinear (which involves very little in the way of I/O for the mesh itself), the default number of mesh variables is 20, for a typical plugin running the MIF parallel I/O mode, each task will emit 8\*20 (there are 8 mesh parts per task), 100K writes for a given dump. The default number of dumps is 10, so this will be repeated 10 times. For a --parallel_file_mode MIF XX run, each dump will produce XX files for a total of 1000 files produced for that first run. In the first run, because file count and task count are 1:1, each file will contain just 8 mesh parts. For the last run, there are 256 files per dump but 262144 tasks, each task having 8 mesh parts. So, each file would, in that case, contain 8192 mesh parts.

Now, because the HDF5 plugin also supports the SIF parallel I/O mode, we could run the same sequence of tests in that mode by replacing MIF XX with SIF.

2.4 Strong Scaling Study Command-Line Example

Suppose we wish to perform a strong scaling study. In this case, we need to settle upon the global final mesh size and then construct MACSio command lines for each run such that the task count together with the per-task command-line arguments results in the same (or approximately so) global final mesh object in each run.

2.4. Strong Scaling Study Command-Line Example
In the preceding weak scaling example, MACSio generated a global mesh of size in the range \([8*32*100K, 8*262144*100K]\). Selecting a middle-of-the-range run of \(8*8192*100K\) (6,710,886,400 bytes) as a nominal global mesh size, we can then use a given task count to determine part size and average part count to hit that target global size. We demonstrate this in the following code block...

```bash
#!/bin/sh
# target total byte count
ttbc=6710886400

nparts_and_part_size()
{
    # start by assuming just one part per task
    nparts=1

    # nominal part size is total target size divided by
    # number of tasks (arg $1 to function). Note that
    # integer arithmetic here will cause some variation from
    # target ttbc
    psize=$(expr $ttbc / $1)

    # if the part size is bigger than the 100K we used in the weak study,
    # lets reduce it and then increase the number of parts
    if [[ $psize -ge 102400 ]]; then
        nparts=$(echo "$psize/102400" | bc -l)
        nparts=$(echo $nparts | cut -d'.' -f1)
        psize=102400
    fi

    echo $nparts $psize
}

n=32
while [[ $n -le 262144 ]]; do
    nparts=$(nparts_and_part_size $n | cut -d' ' -f1)
    psize=$(nparts_and_part_size $n | cut -d' ' -f2)
    nf=$n
    # allow file count to track task count to 1024 tasks
    # then keep it constant after that
    if [[ $nf -ge 1024 ]]; then
        nf=1024
    fi
    cmd="mpirun -np $n ./macsio --interface hdf5 --avg_num_parts $nparts --part_size $psize --parallel_file_mode MIF $nf"
    echo $cmd
    n=$(expr $n \* 2)
done
```

When this shell code is run, it results in the following sequence of MACSio command-lines. Note that we cap the total number of files at 1024. Depending on the particular system where this is run, this cap may be too low or too high. The best number is the number of I/O nodes a given instance of MACSio can see when running. Because this number typically varies with task count and where on the system the tasks are actually allocated, this number is not always easily known or obtained.

```bash
mpirun -np 32 ./macsio --interface hdf5 --avg_num_parts 2048 --part_size 102400 --parallel_file_mode MIF 32
mpirun -np 64 ./macsio --interface hdf5 --avg_num_parts 1024 --part_size 102400 --parallel_file_mode MIF 64
```
mpirun -np 128 ./macsio --interface hdf5 --avg_num_parts 512 --part_size 102400 --parallel_file_mode MIF 128
mpirun -np 256 ./macsio --interface hdf5 --avg_num_parts 256 --part_size 102400 --parallel_file_mode MIF 256
mpirun -np 512 ./macsio --interface hdf5 --avg_num_parts 128 --part_size 102400 --parallel_file_mode MIF 512
mpirun -np 1024 ./macsio --interface hdf5 --avg_num_parts 64 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 2048 ./macsio --interface hdf5 --avg_num_parts 32 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 4096 ./macsio --interface hdf5 --avg_num_parts 16 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 8192 ./macsio --interface hdf5 --avg_num_parts 8 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 16384 ./macsio --interface hdf5 --avg_num_parts 4 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 32768 ./macsio --interface hdf5 --avg_num_parts 2 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 65536 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 131072 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 51200 --parallel_file_mode MIF 1024
mpirun -np 262144 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 25600 --parallel_file_mode MIF 1024

To perform a strong scaling study in SIF parallel I/O mode, just replace the trailing MIF $nf in the above MACSio command line above with SIF.

This strong scaling scenario maintains constant part size (up to the last three iterations) and just varies the part count to hit the target global size. Instead, we may want to do a strong scaling scenario where we maintain just a single part per task and vary that single part’s size to hit the target global size. This is demonstrated with following code...

#!/bin/sh

# target total byte count
ttbc=6710886400

part_size()
{
    psize=$(expr $ttbc / $1)
    echo $psize
}

n=32
while [[ $n -le 262144 ]]; do
    psize=$(part_size $n)
    nf=$n
    # allow file count to track task count to 1024 tasks
    # then keep it constant after that
    if [[ $nf -ge 1024 ]]; then
        nf=1024
    fi
    cmd="mpirun -np $n ./macsio --interface hdf5 --avg_num_parts 1 --part_size $psize
--parallel_file_mode MIF $nf"
    echo $cmd
    n=$(expr $n \* 2)
done

2.4. Strong Scaling Study Command-Line Example
which produces the following sequence of MACSio command-lines...

```
mpirun -np 32 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 209715200 --.parallel_file_mode MIF 32
mpirun -np 64 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 104857600 --parallel_file_mode MIF 64
mpirun -np 128 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 52428800 --parallel_file_mode MIF 128
mpirun -np 256 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 26214400 --parallel_file_mode MIF 256
mpirun -np 512 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 13107200 --parallel_file_mode MIF 512
mpirun -np 1024 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 6553600 --parallel_file_mode MIF 1024
mpirun -np 2048 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 3276800 --parallel_file_mode MIF 1024
mpirun -np 4096 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 1638400 --parallel_file_mode MIF 1024
mpirun -np 8192 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 819200 --parallel_file_mode MIF 1024
mpirun -np 16384 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 409600 --parallel_file_mode MIF 1024
mpirun -np 32768 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 204800 --parallel_file_mode MIF 1024
mpirun -np 65536 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 102400 --parallel_file_mode MIF 1024
mpirun -np 131072 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 51200 --parallel_file_mode MIF 1024
mpirun -np 262144 ./macsio --interface hdf5 --avg_num_parts 1 --part_size 25600 --parallel_file_mode MIF 1024
```

The difference in these two strong scaling scenarios is that the first one will suffer from per-part overheads. Each task will have per-part memory overheads and per-part I/O request overheads which the second scenario does not suffer.

### 2.5 Assessing Performance Achieved by MACSio

On the one hand, MACSio tries to boil all the performance data down to a single, scalar metric; the average time to complete a dump or the average bandwidth for completing a dump. This information is captured in task 0’s section (header) MACSio’s main log file, macsio-log.log.

An example is shown below for 10 dumps from a 4 task run.

```
  Processors 000000------------------
  Info:"macsio_main.c":674:Dump 00 BW:  1.4365 Mi/140.9565 msecs = 10.1907 Mi/sec:::
  Info:"macsio_main.c":679:Dump 00 Stat BW:  4.5891 Mi/140.9565 msecs = 32.5569 Mi/sec:::
  Info:"macsio_main.c":674:Dump 01 BW:  1.4365 Mi/147.8431 msecs = 9.7161 Mi/sec:::
  Info:"macsio_main.c":679:Dump 01 Stat BW:  4.5892 Mi/147.8431 msecs = 31.0409 Mi/sec:::
  Info:"macsio_main.c":674:Dump 02 BW:  1.4365 Mi/159.3053 msecs = 9.0170 Mi/sec:::
  Info:"macsio_main.c":679:Dump 02 Stat BW:  4.5892 Mi/159.3053 msecs = 30.8075 Mi/sec:::
  Info:"macsio_main.c":674:Dump 03 BW:  1.4365 Mi/149.3391 msecs = 9.6187 Mi/sec:::
  Info:"macsio_main.c":679:Dump 03 Stat BW:  4.5892 Mi/149.3391 msecs = 30.7300 Mi/sec:::
```

(continues on next page)
2.5. Assessing Performance Achieved by MACSio
Info: "macsio_main.c":674: Dump 00 BW: 1.4365 Mi/149.2629 msecs = 9.6236 Mi/sec:::
Info: "macsio_main.c":679: Dump 00 Stat BW: 4.5095 Mi/149.2629 msecs = 30.2118 Mi/
sec:::
Info: "macsio_main.c":674: Dump 01 BW: 1.4365 Mi/152.9750 msecs = 9.3901 Mi/sec:::
Info: "macsio_main.c":679: Dump 01 Stat BW: 4.5094 Mi/152.9750 msecs = 29.4781 Mi/
sec:::
Info: "macsio_main.c":674: Dump 02 BW: 1.4365 Mi/158.0414 msecs = 9.0891 Mi/sec:::
Info: "macsio_main.c":679: Dump 02 Stat BW: 4.5094 Mi/158.0414 msecs = 28.5331 Mi/
sec:::
Info: "macsio_main.c":674: Dump 03 BW: 1.4365 Mi/144.5061 msecs = 9.9404 Mi/sec:::
Info: "macsio_main.c":679: Dump 03 Stat BW: 4.5094 Mi/144.5061 msecs = 31.2057 Mi/
sec:::
Info: "macsio_main.c":674: Dump 04 BW: 1.4365 Mi/148.0238 msecs = 9.7042 Mi/sec:::
Info: "macsio_main.c":679: Dump 04 Stat BW: 4.5094 Mi/148.0238 msecs = 30.4641 Mi/
sec:::
Info: "macsio_main.c":674: Dump 05 BW: 1.4365 Mi/148.8822 msecs = 9.6482 Mi/sec:::
Info: "macsio_main.c":679: Dump 05 Stat BW: 4.5094 Mi/148.8822 msecs = 30.2885 Mi/
sec:::
Info: "macsio_main.c":674: Dump 06 BW: 1.4365 Mi/147.0188 msecs = 9.7705 Mi/sec:::
Info: "macsio_main.c":679: Dump 06 Stat BW: 4.5094 Mi/147.0188 msecs = 30.6724 Mi/
sec:::
Info: "macsio_main.c":674: Dump 07 BW: 1.4365 Mi/148.5274 msecs = 9.6713 Mi/sec:::
Info: "macsio_main.c":679: Dump 07 Stat BW: 4.5094 Mi/148.5274 msecs = 30.3608 Mi/
sec:::
Info: "macsio_main.c":674: Dump 08 BW: 1.4365 Mi/150.3812 msecs = 9.5521 Mi/sec:::
Info: "macsio_main.c":679: Dump 08 Stat BW: 4.5094 Mi/150.3812 msecs = 29.9866 Mi/
sec:::
Info: "macsio_main.c":674: Dump 09 BW: 1.4365 Mi/152.4305 msecs = 9.4236 Mi/sec:::
Info: "macsio_main.c":679: Dump 09 Stat BW: 4.5094 Mi/152.4305 msecs = 29.5834 Mi/
sec:::
Info: "macsio_main.c":705: Overall BW: 14.3645 Mi/ 1.5000 secs = 9.5760 Mi/sec:::

(continued from previous page)
The key line in the file, for task 0, is the total bytes divided by the time between the last task to finish and the first task to start which looks similar to the line below...

```
Info:"macsio_main.c":727:Total Bytes: 57.4580 Mi; Last finisher - First starter = 1.5012 secs; BW = 38.2758 Mi/sec:::
```

### 2.5.1 Detailed Performance Data

Detailed performance data is captured by MACSio as timers and then dumped to strings (see `MACSIO_TIMING_DumpTimersToStrings()` for specific format of dumped timer data strings) for output to a log file upon exit.

The performance data gathered by MACSio is wholly dependent on the degree to which developers of MACSio or its plugins have instrumented MACSio with suitable timing calls. If timing information provided is not sufficient, the solution is to submit a PR with relevant timer calls added.

Each task maintains its own unique set of timers for its own activities. In addition, MACSio will reduce all the task-specific timers just prior to completion. All timer data, the task-specific timers together with the reduced timers, is then dumped to a MACSio log file with the name `macsio-timings.log`. That single file captures all of the performance data for a given run of MACSio.

### 2.6 Validating Data MACSio Produces

Most MACSio plugins produce valid scientific computing data which can be processed by other common workflow tools and, in particular, visualization tools such as VisIt, ParaView or Ensight.

The images here demonstrate MACSio, 2D data written with the Silo plugin.

![VisIt displaying MACSio produced data on 4 tasks and 2.5 parts per task with the Silo plugin. From left to right, the visualizations display the parallel decomposition of the mesh into blocks, a smoothly varying, node-centered variable on the mesh, a high noise node-centered variable and a low-noise node-centered variable.](image-url)
2048 Core Example on Vulcan

Fig. 2: VisIt displaying MACSio produced data on 2048 tasks and 2.5 parts per task with the Silo plugin. The left image shows the parallel decomposition of the mesh. The right image shows a smoothly varying node-centered variable. In this example, there were 2048 tasks and 5120 mesh parts.
Welcome to MACSio Documentation

3.1 Main

The design of MACSio is described in more detail in an accompanying design document. Here, we provide only a high level overview.

MACSio is divided into two halves; the MACSio main driver and the I/O plugin(s).

The main driver generates the scientific data objects (examples of which are pictured below) typical of HPC, multi-physics simulation codes. It also orchestrates a number of activities such as compute and/or communication workloads to be mixed with the I/O workload, time and space performance data gathering, event logging and other support operations.

![Fig. 1: Scientific computing data objects typical of mesh-based multi-physics simulation codes](image)

The other half of MACSio is the I/O plugin(s). A MACSio plugin orchestrates marshalling of data between memory and disk. Different plugins exist to implement different I/O proxies. For example, the Silo plugin serves as a proxy for many (not all) LLNL simulation codes. The Exodus plugin serves as proxy for many Sandia simulation codes. On the
other hand, instead of representing a proxy for any specific organization or code, the HDF5 plugin serves as a proxy for the best way to use HDF5 for various parallel I/O paradigms. It also provides a number of command-line options to control low-level features of the HDF5 library to help probe the impact of various options on the performance space.

Note: Should extend MACSio to support scripted sequences of dumps

MACSio’s main driver accepts a slew of command-line arguments. If MACSio has been built, you can obtain help for MACSio’s main driver arguments

MACSio’s command-line arguments are designed to give the user control over the nominal I/O request sizes emitted from MPI ranks for mesh bulk data and for amorphous metadata. The user specifies a size, in bytes, for mesh pieces. MACSio then computes a mesh part size, in nodes, necessary to hit this target byte count for double precision data. MACSio will determine an N dimensional logical size of a mesh piece that is a close to equal dimensional as possible. In addition, the user specifies an average number of mesh pieces that will be assigned to each MPI rank. This does not have to be a whole number. When it is a whole number, each MPI rank has the same number of mesh pieces. When it is not, some processors have one more mesh piece than others. This is common of HPC multi-physics applications. Together, the total processor count and average number of mesh pieces per processor gives a total number of mesh pieces that comprise the entire mesh. MACSio then finds an N dimensional arrangement (N=[1,2,3]) of the pieces that is as close to equal dimension as possible. If mesh piece size or total count of pieces wind up being prime numbers, MACSio will only be able to factor these into long, narrow shapes where 2 (or 3) of the dimensions are of size 1. That will make examination of the resulting data using visualization tools like VisIt a little less convenient but is otherwise harmless from the perspective of driving and assessing I/O performance.

3.2 Mesh and Field Data Generation

MACSio is capable of generating a wide variety of mesh and variable data and amorphous metadata typical of HPC multi-physics applications. Currently, MACSio can generate structured, rectilinear, unstructured and arbitrary polyhedral, multi-block meshes in 2 or 3 dimensions. In addition, regardless of the particular type of mesh MACSio generates for purposes of I/O performance testing, it stores and marshalls all of the resultant data in an uber JSON-C object that is passed around within MACSio and between MACSIO and its I/O plugins.

MACSio employs a very simple algorithm to generate and then decompose a mesh in parallel. However, the decomposition is also general enough to create multiple mesh pieces on individual MPI ranks and for the number of mesh pieces to vary, somewhat, between MPI ranks. At present, there is no support to explicitly specify a particular arrangement of mesh pieces and MPI ranks. However, such enhancement is planned for the near future.

Once the global whole mesh shape is determined as a count of total pieces and as counts of pieces in each of the logical dimensions, MACSio uses a very simple algorithm to assign mesh pieces to MPI ranks. The global list of mesh pieces is numbered starting from 0. First, the number of pieces to assign to rank 0 is chosen. When the average piece count is non-integral, it is a value between K and K+1. So, MACSio randomly chooses either K or K+1 pieces but being careful to weight the randomness so that once all pieces are assigned to all ranks, the average piece count per rank target is achieved. MACSio then assigns the next K or K+1 numbered pieces to the next MPI rank. It continues assigning pieces to MPI ranks, in piece number order, until all MPI ranks have been assigned pieces. The algorithm runs identically on all ranks. When the algorithm reaches the part assignment for the rank on which its executing, it then generates the K or K+1 mesh pieces for that rank. Although the algorithm is essentially a sequential algorithm with asymptotic behavior O(#total pieces), it is primarily a simple book-keeping loop which completes in a fraction of a second even for more than one million pieces.

Each piece of the mesh is a simple rectangular region of space. The spatial bounds of that region are easily determined. Any variables to be placed on the mesh can be easily handled as long as the variable’s spatial variation can be described in the global geometric space.
3.2.1 Data API

**group MACSIO_DATA**

Data (Mesh) Generation.

**Functions**

```c
struct json_object *MACSIO_DATA_GenerateTimeZeroDumpObject (struct json_object *main_obj, int *rank_owning_chunkId)
```

**Parameters**

- `main_obj`: The main JSON object holding mesh, field, amorphous data
- `rank_owning_chunkId`: missing this info

```c
int MACSIO_DATA_GetRankOwningPart (struct json_object *main_obj, int chunkId)
```

Given a chunkId, return rank of owning task.

```c
int MACSIO_DATA.ValidateDataRead (struct json_object *main_obj)
```

Not yet implemented.

```c
int MACSIO_DATA_SimpleAssignKPartsToNProcs (int k, int n, int my_rank, int *my_part_cnt, int **my_part_ids)
```

Not yet implemented.

```c
struct json_object *MACSIO_DATA_EvolveDataset (struct json_object *main_obj, int *dataset_evolved, float factor, int growth_bytes)
```

Vary mesh and field data including changing not only values but sizes.

3.2.2 Random (Amourphous) Object Generation

In addition to the data necessary to represent the main mesh and field MACSio produces, it is also common for HPC multi-physics code to generate modest amount of amourphous metadata dealing with such things as materials, material models and material properties tables, user input controls, performance tracking information, library versions and data provenance, debugging information, etc. This kind of information often takes the form of a large hierarchy of key-value pairs where the values can sometimes be arrays on the order of the size of the number of tasks, etc.

Two methods in MACSio’s data generation package help to create some random but nonetheless statistically similar kinds of data.

**group MACSIO_RANDOBJ**

Construct a random JSON object.

**Functions**

```c
struct json_object *MACSIO_DATA.MakeRandomObject (int maxd, int nraw, int nmeta, unsigned maxds)
```

**Parameters**

- `maxd`: maximum depth of object tree
- `nraw`: bytes of raw data (in extarr objects) in final object
- `nmeta`: bytes of meta data (in non-extarr objects) in final object
• **maxds**: maximum size of any given raw (extarr) object

```c
struct json_object *MACSIO_DATA_MakeRandomTable(int nrecs, int totbytes)
```
Construct a random table of some number of random JSON objects.

### Parameters
- **nrecs**: number of records in the table
- **totbytes**: total bytes in the table

### 3.2.3 Randomization

Randomization of various of MACSio’s behaviors may be explicitly enforced or prevented. However, to ensure such behaviors, random number generation needs to be handled carefully especially when dealing with reproducibility from run to run and/or agreement across parallel tasks or both.

For this reason, MACSio creates and initializes several default pseudo random number generators (PRNGs), all of which utilize the C standard library random() method.

Pseudo Random Number Generator (PRNG) support is handled by maintaining a series of state vectors for calls to initstate/setstate of C library random() and friends. Multiple different PRNGs are supported each having zero or more special properties depending on how they are initialized (e.g. the seed used from run to run and/or from task to task) as well as how they are expected to be used/called. For example, if a PRNG is intended to produce identical values on each task, then it is up to the caller to ensure the respective PRNG is called consistently (e.g. collectively) across tasks. Otherwise, unintended or indeterminate behavior may result.

The default PRNGs MACSio creates are...

**Naive** [MD_random_naive] The naive prng is fine for any randomization that does not need special behaviors. This is the one that is expected to be used most of the time. Think of it as a wholesale replacement for using C library random() directly. MD_random is a shorthand alias for MD_random_naive

**Naive, Time Varying** [MD_random_naive_tv] Like MD_random_naive except that it is seeded based on current time and will therefore ensure variation from run to run (e.g. is time varying). It should be used in place of MD_random_naive where randomization from run to run is needed.

**Naive, Rank Varying** [MD_random_naive_rv] Like MD_random_naive but ensures variation across tasks (e.g. is rank varying) by seeding based on task rank. It should be used in place of MD_random_naive where randomization from task to task is needed.

**Naive, Rank and Time Varying** [MD_random_naive_rtv] Like MD_random_naive_rv but also ensures results vary from rank to rank and run to run.

**Rank Invariant** [MD_random_rankinv] Is neither seeded based on rank or time nor shall be used by a caller in way that would cause inconsistency across ranks. It is up to the caller to ensure it is called consistently (e.g. collectively) across ranks. Otherwise indeterminate behavior may result.

**Rank Invariant, Time Varying** [MD_random_rankinv_tv] Like MD_random_rankinv except ensures variation in randomization from run to run. So, for each given run, all ranks will agree on randomization but from run to run, that agreement will vary.

If these PRNGs are not sufficient, developers are free to create other PRNGs for other specific purposes using MACSIO_DATA_CreatePRNG().

### 3.2.4 PRNG API

**group MACSIO_PRNGS**
Randomization and Pseudo Random Number Generators.
Defines

**MD_random_naive()**
Get next random value from naive PRNG. Same as *MD_random()*.

The naive prng is fine for any randomization that does not need special behaviors. This is the one that is expected to be used most of the time.

**MD_random()**
Convenience shorthand for *MD_random_naive()*

**MD_random_naive_tv()**
Get next random value from naive, time-variant PRNG.

The naive_tv PRNG is like naive except that it is seeded based on current time and so will change from run to run (e.g. is time-variant). It should be used in place of *MD_random_naive()* when randomization from run to run is required.

**MD_random_naive_rv()**
Get next random value from naive, rank-variant PRNG.

Like *MD_random_naive() but guarantees* variation across ranks.

**MD_random_naive_rtv()**
Get next random value from naive, rank- and time-variant PRNG.

Like *MD_random_naive_rv() but guarantees* variation from run to run.

**MD_random_rankinv()**
Get next random value from rank-invariant PRNG.

The rank_invariant PRNG is neither seeded based on rank nor shall be used in way that would cause inconsistency across ranks. Failing to call it collectively will likely cause indeterminent behavior. There is no enforcement of this requirement. However, a sanity check is performed at application termination...there is a collective MPI_BXOR reduce on the current value from this PRNG and if the result is non-zero, an error message is generated in the logs.

**MD_random_rankinv_tv()**
Get next random value from rank-invariant but time-variant PRNG.

Like the *MD_random_rankinv() except causes* variation in randomization from run to run.

Functions

```c
int MACSIO_DATA_CreatePRNG (unsigned seed)
Create a Pseudo Random Number Generator (PRNG)

Parameters
  • seed: seed for initializing random number generator

Return An integer identifier for the created PRNG
```

```c
void MACSIO_DATA_ResetPRNG (int prng_id)
Reset a PRNG This will cause the specified PRNG to restart the sequence of pseudo random numbers it generates.

Parameters
  • prng_id: identifier for the PRNG to reset
```
long MACSIO_DATA_GetValPRNG (int prng_id)
Get next value from PRNG.

Parameters
• prng_id: identifier of the desired PRNG to get a value from

Return Next value from the PRNG sequence. Logically equivalent to calling random().

void MACSIO_DATA_DestroyPRNG (int prng_id)
Free resources associated a PRNG.

Parameters
• prng_id: identifier of the desired PRNG to free

void MACSIO_DATA_InitializeDefaultPRNGs (unsigned mpi_rank, unsigned curr_time)
Initialize default Pseudo Random Number Generators (PRNGs) Should be called near beginning of application startup. Creates the (default) PRNGs described here. Applications are free to create others as needed.

Parameters
• mpi_rank: unsigned representation of MPI rank of calling processor
• curr_time: unsigned but otherwise arbitrary representation of current time but which is equal on all ranks yet guaranteed to vary from run to run.

void MACSIO_DATA_FinalizeDefaultPRNGs (void)
Free up resources for default PRNGs Should be called near the termination of application.

3.2.5 PRNG Example (tstprng.c)

```c
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#ifdef HAVE_MPI
#include <mpi.h>
#endif

#include <macsio_data.h>

int main(int argc, char **argv)
{
    int i, id1, id2, id3, id5;
    int parallel = 0, ckmfail = 0;
    long tseed;
    long series1[100], series2[100], series3[100], series4[100][5],
          series5[100][5];
    int rank = 0, size = 1;
    struct timeval tv;

    for (i = 0; i < argc; i++)
        \n```
MACSio, Release 1.1.0

if (!strcasestr(argv[i], "--parallel"))
    parallel = 1;
else if (!strcasestr(argv[i], "--ckfail"))
    ckfail = 1;

#ifdef HAVE_MPI
    if (parallel)
        {
            MPI_Init(&argc, &argv);
            MPI_Comm_size(MPI_COMM_WORLD, &size);
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        }
#endif

gmtimeofday(&tv, 0);
tseed = (long) (((unsigned int) (tv.tv_sec + tv.tv_usec)) & 0xFFFFFFFF);

MACSIO_DATA_InitializeDefaultPRNGs(rank, tseed);

id1 = MACSIO_DATA_CreatePRNG(0xDeadBeef);
id2 = MACSIO_DATA_CreatePRNG(tseed); /* time varying output */
id3 = MACSIO_DATA_CreatePRNG(0xBabeFace);
id5 = MACSIO_DATA_CreatePRNG(0xDeadBeef);

for (i = 0; i < 100; i++)
    {
        int j, id4 = MACSIO_DATA_CreatePRNG(0x0BedFace);
        series1[i] = MACSIO_DATA_GetValPRNG(id1);
        series2[i] = i%3?0:MACSIO_DATA_GetValPRNG(id2);
        series3[i] = i%5?0:MACSIO_DATA_GetValPRNG(id3);
        for (j = 0; j < 5; j++)
            series4[i][j] = MACSIO_DATA_GetValPRNG(id4);
        MACSIO_DATA_DestroyPRNG(id4);
        for (j = 0; j < 5; j++)
            series5[i][j] = MACSIO_DATA_GetValPRNG(id5);
        MACSIO_DATA_ResetPRNG(id5);
        if (i && !(i%50))
            MACSIO_DATA_ResetPRNG(id2);
        MD_random();
        MD_random_rankinv();
        if (ckfail && rank == 0 && i < 5)
            {
                MD_random_rankinv();
                MD_random_rankinv_tv();
            }
        if (memcmp(&series2[0], &series2[51], 40*sizeof(long)))
            return 1;
        if (memcmp(&series4[0], &series4[1], 5*sizeof(long)))
            return 1;
        if (memcmp(&series4[1], &series4[23], 5*sizeof(long)))
            return 1;

3.2. Mesh and Field Data Generation 21
MACSio, Release 1.1.0

75      return 1;
76      if (memcmp(&series5[0], &series5[1], 5*sizeof(long)))
77      return 1;
78      if (memcmp(&series5[1], &series5[23], 5*sizeof(long)))
79      return 1;
80
81      MACSIO_DATA_DestroyPRNG(id1);
82      MACSIO_DATA_DestroyPRNG(id2);
83      MACSIO_DATA_DestroyPRNG(id3);
84      MACSIO_DATA_DestroyPRNG(id5);
85
86      #ifdef HAVE_MPI
87      if (parallel)
88      {
89          /* Lets confirm the rank-invariant PRNGs agree and issue a_
80             message if not */
90          unsigned rand_check[2] = {MD_random_rankinv(), MD_random_rankinv_
91             tv()};
92          unsigned rand_check_r[2];
93          MPI_Reduce(rand_check, rand_check_r, 2, MPI_UNSIGNED, MPI_BXOR, 0,
94             MPI_COMM_WORLD);
95          assert(rand_check_r[0] == 0 && rand_check_r[1] == 0);
96      }
97      #endif
98
99      MACSIO_DATA_FinalizeDefaultPRNGs();
100
101      #ifdef HAVE_MPI
102      if (parallel)
103          MPI_Finalize();
104      #endif
105  

3.3 Plugins

Ordinarily, when we think of plugins in software, we think of shared libraries being opened with dlopen() at runtime. To avoid the requirement for shared libraries and dlopen(), MACSio uses a simpler link-time approach to handling plugins called static plugins. In this case, the plugin’s available to MACSio are determined when MACSio’s main() is linked. Whatever plugins are included on the link line are then available to MACSio. MACSio exploits a feature in C++ which permits initialization of static variables via non-constant expressions. In fact, the only reason MACSio requires a C++ compiler is for the final link to support this feature.

All symbols in a plugin source file are defined with static scope. Every plugin defines an int register_this_plugin(void) function and initializes a static dummy integer to the result of a call to register_this_plugin() like so...

```c
static int register_this_plugin(void)
{
    MACSIO_IFACE_Handle_t iface;
}
```

(continues on next page)
In the above code example, the call to `MACSIO_IFACE_Register()` calls a method in MACSio’s main which does the work of adding the plugin’s interface specification (`MACSIO_IFACE_Handle_t`) to a statically defined table of plugins. The plugin’s id is its location in that table.

Each plugin is defined by a file such as `macsio_foo.c` for a plugin named foo in the `plugins` directory. `macsio_foo.c` implements the `MACSIO_IFACE` interface for the foo plugin.

At the time the executable loads, the `register_this_plugin()` method is called. Note that this is called long before even `main()` is called. The call to `MACSIO_IFACE_Register()` from within `register_this_plugin()` winds up adding the plugin to MACSio’s global list of plugins. This happens for each plugin. The order in which they are added to MACSio doesn’t matter because plugins are identified by their (unique) names. If MACSio encounters a case where two different plugins have the same name, then it will abort and inform the user of the problem. The remedy is to adjust the name of one of the two plugins. MACSio is able to call static methods defined within the plugin via function callback pointers registered with the interface.

```c
strcpy(iface.name, iface_name);
strcpy(iface.ext, iface_ext);

if (!MACSIO_IFACE_Register(&iface))
    MACSIO_LOG_MSG(Die, ("Failed to register interface \"%s\"", iface.name));
}
static int dummy = register_this_plugin();
```

3.3. Plugins
Functions

int MACSIO_IFACE_Register (MACSIO_IFACE_Handle_t const *iface)
  Register a plugin with MACSIO.

Parameters
  • iface: The interface specification for a new plugin

This function should be called from within a code-block which is executed as part of a static initialization of the plugin’s local, static variables like so...

```c
static int register_this_plugin(void)
{
    MACSIO_IFACE_Handle_t iface;
    iface.name = "foobar";
    iface.ext = ".fb";
    .
    .
    MACSIO_IFACE_Register(&iface);
    return 0;
}
```

void MACSIO_IFACE_GetIds (int *cnt, int **ids)
  Return all registered plugin ids.

Parameters
  • cnt: the number of registered plugins available
  • ids: the list of ids of the registered plugins

void MACSIO_IFACE_GetIdsMatchingFileExtension (char const *ext, int *cnt, int **ids)
  Return plugin that recognize files with a given extension.

Parameters
  • ext: the file extension to query
  • cnt: the number of plugins that recognize the extension
  • ids: the list of ids of the matching plugins

int MACSIO_IFACE_GetId (char const *name)
  return id of plugin given its name

  Return  On error, returns -1. Otherwise a non-negative valid plugin id.

char const *MACSIO_IFACE_GetName (int id)
  Get name of plugin given its id.

  Return  On error, returns (char*)0. Otherwise a non-null name of valid plugin.

MACSIO_IFACE_Handle_t const *MACSIO_IFACE_GetByName (char const *name)
  Get interface handle for a plugin given its name.
Return On error, returns (MACSIO_IFACE_Handle_t*)0. Otherwise returns a valid plugin interface handle.

MACSIO_IFACE_Handle_t const *MACSIO_IFACE_GetById(int id)
Get interface handle for a plugin given its id.

Return On error, returns (MACSIO_IFACE_Handle_t*)0. Otherwise returns a valid plugin interface handle.

Variables

char name[MACSIO_IFACE_MAX_NAME]
Name of this plugin

char ext[MACSIO_IFACE_MAX_NAME]
File extension of files associated with this plugin

int slotUsed
[Internal] indicate if this position in table is used

ProcessArgsFunc processArgsFunc
Plugin’s command-line argument processing callback

DumpFunc dumpFunc
Plugin’s main dump (write) function callback

LoadFunc loadFunc
Plugin’s main load (read) function callback

QueryFeaturesFunc queryFeaturesFunc
Plugin’s callback to query its feature set (not in use)

IdentifyFileFunc identifyFileFunc
Plugin’s callback to indicate if it thinks it owns a file

struct MACSIO_IFACE_Handle_t
#include <macsio_iface.h> Plugin interface handle type.

Public Members

char name[MACSIO_IFACE_MAX_NAME]
Name of this plugin

char ext[MACSIO_IFACE_MAX_NAME]
File extension of files associated with this plugin

int slotUsed
[Internal] indicate if this position in table is used

ProcessArgsFunc processArgsFunc
Plugin’s command-line argument processing callback

DumpFunc dumpFunc
Plugin’s main dump (write) function callback

LoadFunc loadFunc
Plugin’s main load (read) function callback

QueryFeaturesFunc queryFeaturesFunc
Plugin’s callback to query its feature set (not in use)
IdentifyFileFunc `identifyFileFunc`

Plugin’s callback to indicate if it thinks it owns a file

### 3.3.1 Example (macsio_miftmpl.c)

Writing a new plugin for MACSio involves a minimum of two new files in the plugins directory. One is the C or C++ source code for the plugin. The other is a `.make` file that includes various plugin-specific variable definitions and logic to decode library dependencies that effect.

The `miftmpl` plugin is intended to serve as a template for how to create a basic MIF mode plugin for MACSio. This template code does indeed actually function correctly as a MACSio plugin. It does so by writing MACSio’s internal JSON objects representing each mesh part as ascii strings to the individual files.

Each processor in a MIF group serializes each JSON object representing a mesh part to an ascii string. Then, each of these strings is appended to the end of the file. For each such string, the plugin maintains knowledge of the mesh part’s ID, the filename it was written to and the offset within the file.

The filenames, offsets and mesh part IDs are then written out as a separate JSON object to a root or master file. Currently, there is no plugin in VisIt to read these files and display them. But, this example code does help to outline the basic work to write a MIF plugin.

In practice, this plugin could have simply written the entire JSON object from each processor to its MIF group’s file. However, in doing that, the resulting file would not “know” such things as how many mesh parts there are or where a given mesh part is located in the file set. So, we wind up writing JSON objects for each part individually so that we can keep track of where they all are in the fileset.

Some of the aspects of this plugin code exist here only to serve as an example in writing a MIF plugin and are non-essential to the proper operation of this plugin.

MACSio uses a static load approach to its plugins. The MACSio main executable must be linked with all the plugins it expects to use.

In writing any MACSio plugin (MIF or SSF) be sure to declare all of your plugin’s symbols (functions, local variables, etc.) as static. Each plugin is being linked into MACSio’s main and any symbols that are not static file scope will wind up appearing in and therefore being vulnerable too global namespace collisions. The plugin’s main interface methods to MACSio are handled via registration of a set of function pointers.

### Functions

**static int process_args** (int argi, int argc, char *argv[])

Process command-line arguments specific to this plugin.

**Parameters**

- `argi`: Argument index of first argument that is specific to this plugin
- `argc`: argc as passed into main
- `argv`: argv as passed into main

Uses `MACSIO_CLARGS_ProcessCmdline()` to do its work.

This example plugin is implemented to route command line arguments to memory locations (e.g. static variables) here in the plugin. Alternatively, a plugin can choose to route the results of `MACSIO_CLARGS_ProcessCmdline()` to a JSON object. MACSio’s main is implemented that way.

**static void *CreateMyFile** (const char *fname, const char *nsname, void *userData)

CreateFile MIF Callback.
Parameters

- **fname**: Name of the MIF file to create
- **nsname**: Name of the namespace within the file for caller should use.
- **userData**: Optional plugin-specific user-defined data

This implements the MACSIO_MIF_CreateFile callback needed for a MIF mode plugin.

**Return**  A void pointer to the plugin-specific file handle

```c
static void *OpenMyFile(const char *fname, const char *nsname, MACSIO_MIF_ioFlags_t ioFlags,
                        void *userData)
```

OpenFile MIF Callback.

Parameters

- **fname**: Name of the MIF file to open
- **nsname**: Name of the namespace within the file caller should use
- **ioFlags**: Various flags indicating behavior/options
- **userData**: Optional plugin-specific user-defined data

This implements the MACSIO_MIF_OpenFile callback needed for a MIF mode plugin.

**Return**  A void pointer to the plugin-specific file handle

```c
static int CloseMyFile(void *file, void *userData)
```

CloseFile MIF Callback.

Parameters

- **file**: A void pointer to the plugin specific file handle
- **userData**: Optional plugin specific user-defined data

This implements the MACSIO_CloseFile callback needed for a MIF mode plugin.

```c
static json_object *write_mesh_part(FILE *myFile, char const *fileName, json_object *part_obj)
```

Write a single mesh part to a MIF file.

Parameters

- **myFile**: The file handle being used in a MIF dump
- **fileName**: Name of the MIF file
- **part_obj**: The json object representing this mesh part

All this method does is serialize the JSON object for the given mesh part to an ASCII string and then appends/writes that string at the end of the current file.

After serializing the object to an ASCII string and writing it to the file, the memory for the ASCII string is released by json_object_free_printbuf().

**Return**  A tiny JSON object holding the name of the file, the offset at which the JSON object for this part was written in the file and the part’s ID.
static void main_dump (int argi, int argc, char **argv, json_object *main_obj, int dumpn, double dumpt)
Main MIF dump implementation for this plugin.

Parameters

• argi: Command-line argument index at which first plugin-specific arg appears
• argc: argc from main
• argv: argv from main
• main_obj: The main json object representing all data to be dumped
• dumpn: The number/index of this dump. Each dump in a sequence gets a unique, monotone increasing index starting from 0
• dumpt: The time to be associated with this dump (like a simulation’s time)

This is the function MACSio main calls to do the actual dump of data with this plugin.

It uses MACSIO_MIF twice; once for the main dump and a second time to create the root (or master) file. However, in the second use, the file count is set to 1. That means that the root file is effectively written using serial (e.g. non-parallel) I/O.

It is a useful exercise to ask how we might improve the implementation here to avoid writing the root file using serial I/O.

static int register_this_interface()
Method to register this plugin with MACSio main.

Due to its use to initialize a file-scope, static const variable, this function winds up being called at load time (e.g. before main is even called).

Its purpose is to add key information about this plugin to MACSio’s global interface table.

Variables

char const *iface_name = "miftmpl"
Name of the interface this plugin uses

char const *iface_ext = "json"
Default file extension for files generated by this plugin

int json_as_html = 0
Use HTML output instead of raw ascii

int my_opt_one
Example of a static scope, plugin-specific variable to be set in process_args to control plugin behavior

int my_opt_two
Another example variable to control plugin behavior

char *my_opt_three_string
Another example variable to control plugin behavior

float my_opt_three_float
Another example variable to control plugin behavior

int const dummy = register_this_interface()
Dummy initializer to trigger register_this_interface by the loader.

This one statement is the only statement requiring compilation by a C++ compiler. That is because it involves initialization and non constant expressions (a function call in this case). This function call is guaranteed to occur
during initialization (that is before even ‘main’ is called) and so will have the effect of populating the iface_map array merely by virtue of the fact that this code is linked with a main.

```c
#include <macsio_clargs.h>
#include <macsio_iface.h>
#include <macsio_log.h>
#include <macsio_main.h>
#include <macsio_mif.h>
#include <macsio_utils.h>

#include <stdio.h>

#ifdef HAVE_MPI
#include <mpi.h>
#endif

/*! *
 * @defgroup plugins Plugins *
 */

/*! *
 * @addtogroup MIF Template *
 * @brief A simple MIF Plugin Template *
 */

static char const *iface_name = "miftmpl"; /**< Name of the interface
this plugin uses */
static char const *iface_ext = "json"; /**< Default file extension
for files generated by this plugin */
static int json_as_html = 0; /**< Use HTML output instead
of raw ascii */
static int my_opt_one; /**< Example of a static scope,
plugin-specific variable to be set in
process_args to control */
static int my_opt_two; /**< Another example variable
-to control plugin behavior */
static char *my_opt_three_string; /**< Another example variable
-to control plugin behavior */
static float my_opt_three_float; /**< Another example variable
-to control plugin behavior */

/*! *
 * @brief Process command-line arguments specific to this plugin *
 */

Uses MACSIO_CLARGS_ProcessCmdline() to do its work.

This example plugin is implemented to route command line arguments to
-memory locations
(e.g. static variables) here in the plugin. Alternatively, a plugin can_
choose to
route the results of MACSIO_CLARGS_ProcessCmdline() to a JSON object.

3.3. Plugins
MACSio's main is implemented that way.

```c
static int process_args(
    int argi, /**< [in] Argument index of first argument that is specific to this plugin */
    int argc, /**< [in] argc as passed into main */
    char **argv[] /**< [in] argv as passed into main */
) {
    /* Can use MACSIO_CLARGS_TOJSON here instead in which case pass the pointer to a json_object* as first arg and eliminate all the pointers to specific variables. The args will be returned as a json-c object. */
    const MACSIO_CLARGS_ArgvFlags_t argFlags = {MACSIO_CLARGS_WARN, MACSIO_CLARGS_TOMEM};
    MACSIO_CLARGS_ProcessCmdline(0, argFlags, argi, argc, argv,
        "--json_as_html", "Write files as HTML instead of raw ascii [false]",
        "--my_opt_one", "Help message for my_opt_one which has no arguments. If present, local\nvar my_opt_one will be assigned a value of 1 and a value of zero otherwise.",
        "--my_opt_two %d", MACSIO_CLARGS_NODEFAULT,
        "Help message for my_opt_two which has a single integer argument",
        "--my_opt_two %d", MACSIO_CLARGS_NODEFAULT,
        "Help message for my_opt_three which has a single integer argument",
        "--my_opt_three %s %f", MACSIO_CLARGS_NODEFAULT,
        "Help message for my_opt_three which has a string argument and a float argument",
        MACSIO_CLARGS_END_OF_ARGS);
    return 0;
}
```

Chapter 3. Welcome to MACSio Documentation

MACSIO_MIF_CreateFile Callback

This implements the MACSIO_MIF_CreateFile callback needed for a MIF mode plugin.

```c
static void *CreateMyFile(
    const char *fname, /**< [in] Name of the MIF file to create */
    const char *nsname, /**< [in] Name of the namespace within the file for caller should use. */
    void *userData /**< [in] Optional plugin-specific user-
```c
/*
 * defined data */
85 }
86 {
87     FILE *file = fopen(fname, "w");
88     return (void *) file;
89 }
90
91 /*!
92 \brief OpenFile MIF Callback
93 This implems the MACSIO_MIF_OpenFile callback needed for a MIF mode plugin.
94
95 \return A void pointer to the plugin-specific file handle
96 */
97 static void *OpenMyFile(
98     const char *fname, / **< [in] Name of the MIF file to open */
99     const char *nsname, / **< [in] Name of the namespace within the file caller should use */
100     MACSIO_MIF_ioFlags_t ioFlags, / **< [in] Various flags indicating behavior/options */
101     void *userData / **< [in] Optional plugin-specific user-defined data */
102 )
103 {
104     FILE *file = fopen(fname, "a+");
105     return (void *) file;
106 }
107
108 /*!
109 \brief CloseFile MIF Callback
110 This implems the MACSIO_CloseFile callback needed for a MIF mode plugin.
111 */
112 static int CloseMyFile(
113     void *file, / **< [in] A void pointer to the plugin specific file handle */
114     void *userData / **< [in] Optional plugin specific user-defined data */
115 )
116 {
117     return fclose((FILE *) file);
118 }
119
120 /*!
121 \brief Write a single mesh part to a MIF file
122 All this method does is serialize the JSON object for the given mesh part to an ASCII string and then appends/writes that string at the end of the current file.
123 After serializing the object to an ASCII string and writing it to the file, the memory for the ASCII string is released by json_object_free_ */
```
return A tiny JSON object holding the name of the file, the offset at
which the JSON object for this part was written in the file and the_
part's ID.

static json_object *write_mesh_part(
    FILE *myFile, /**< [in] The file handle being used in a MIF_
dump */
    char const *fileName, /**< [in] Name of the MIF file */
    json_object *part_obj /**< [in] The json object representing this_
    mesh part */
) {
    json_object *part_info = json_object_new_object();
    // warning SOMEHOW SHOULD INCLUDE OFFSETS TO EACH VARIABLE
    /* Write the json mesh part object as an ascii string */
    fprintf(myFile, "%s\n", json_object_to_json_string_ext(part_obj, JSON_
        C_TO_STRING_PRETTY));
    json_object_free_printbuf(part_obj);
    /* Form the return 'value' holding the information on where to find_
    this part */
    json_object_object_add(part_info, "partid",
        json_object_new_int(json_object_path_get_int(part_obj, "Mesh/
            ChunkID")));
    json_object_object_add(part_info, "file",
        json_object_new_string(fileName));
    json_object_object_add(part_info, "offset",
        json_object_new_double((double) ftello(myFile)));
    return part_info;
}

static void main_dump(
    /* brief Main MIF dump implementation for this plugin
    This is the function MACSio main calls to do the actual dump of data with_
    this plugin.
    It uses MACSIO_MIF twice; once for the main dump and a second time_
    to create the root (or master) file. However, in the second use, the file count is set_
    to 1. That means that the root file is effectively written using serial (e.g. non-
    parallel) I/O.
    It is a useful exercise to ask how we might improve the implementation_
    here to avoid writing the root file using serial I/O.
    */
    static void main_dump(
int argi, /**< [in] Command-line argument index at which first plugin-specific arg appears */
int argc, /**< [in] argc from main */
char **argv, /**< [in] argv from main */
json_object *main_obj, /**< [in] The main json object representing all data to be dumped */
dumpn, /**< [in] The number/index of this dump. Each dump in a sequence gets a unique, monotone increasing index starting from 0 */
dumpt /**< [in] The time to be associated with this dump (like a simulation's time) */
);

int i, rank, numFiles;
char fileName[256];
FILE *myFile;
MACSIO_MIF_ioFlags_t ioFlags = {MACSIO_MIF_WRITE,(unsigned int)JsonGetInt(main_obj,"clargs/exercise_scr")&0x1};
MACSIO_MIF_baton_t *bat;
json_object *parts;
json_object *part_infos = json_object_new_array();

/* process cl args */
process_args(argi, argc, argv);

/* ensure we're in MIF mode and determine the file count */
#elif SIMPLIFY THIS LOGIC USING NEW JSON INTERFACE
json_object *parfmode_obj = json_object_path_get_array(main_obj,"clargs/parallel_file_mode");
if (parfmode_obj)
{
    json_object *modestr = json_object_array_get_idx(parfmode_obj, 0);
    json_object *filecnt = json_object_array_get_idx(parfmode_obj, 1);
    if (!strcmp(json_object_get_string(modestr), "SIF"))
        MACSIO_LOG_MSG(Die, ("miftmpl plugin cannot currently handle SIF mode");
    else
    {
        numFiles = json_object_get_int(filecnt);
    }
}
else
{
    char const * modestr = json_object_path_get_string(main_obj,"clargs/parallel_file_mode");
    if (!strcmp(modestr, "SIF"))
        MACSIO_LOG_MSG(Die, ("miftmpl plugin cannot currently handle SIF mode");
    else if (!strcmp(modestr, "MIFMAX")
        else if (!strcmp(modestr, "MIFMAX")
    
3.3. Plugins 33
numFiles = json_object_path_get_int(main_obj, "parallel/mpi_size");
else if (!strcmp(modestr, "MIFAUTO")) {
    /* Call MACSio utility to determine optimal file count */
}

bat = MACSIO_MIF_Init(numFiles, ioFlags, MACSIO_MAIN_Comm, 3,
            CreateMyFile, OpenMyFile, CloseMyFile, 0);

rank = json_object_path_get_int(main_obj, "parallel/mpi_rank");
/* Construct name for the silo file */
sprintf(fileName, "%s_json_%05d_%03d.%s",
    json_object_path_get_string(main_obj, "clargs/filebase"),
    MACSIO_MIF_RankOfGroup(bat, rank),
    dumpn,
    json_object_path_get_string(main_obj, "clargs/fileext"));
MACSIO_UTILS_RecordOutputFiles(dumpn, fileName);

myFile = (FILE *) MACSIO_MIF_WaitForBaton(bat, fileName, 0);
parts = json_object_path_get_array(main_obj, "problem/parts");
for (i = 0; i < json_object_array_length(parts); i++) {
    json_object *this_part = json_object_array_get_idx(parts, i);
    json_object_array_add(part_infos, write_mesh_part(myFile, fileName, this_part));
}
/* Hand off the baton to the next processor. This winds up closing
   * the file so that the next processor that opens it can be assured
   * of getting a consistent and up to date view of the file's contents.
   */
MACSIO_MIF_HandOffBaton(bat, myFile);
/* We're done using MACSIO_MIF for these files, so finish it off */
MACSIO_MIF_Finish(bat);
/* Use MACSIO_MIF a second time to manage writing of the master/root
file contents. This winds up being serial I/O but also means we
never collect all info on all parts to any single processor. */
#ifwarning THERE IS A BETTER WAY TO DO USING LOOP OF NON-BLOCKING RECIEVES
bat = MACSIO_MIF_Init(1, ioFlags, MACSIO_MAIN_Comm, 5,
            CreateMyFile, OpenMyFile, CloseMyFile, 0);
/* Construct name for the silo file */
sprintf(fileName, "%s_json_root_%03d.%s",
    json_object_path_get_string(main_obj, "clargs/filebase"),
    dumpn,
    json_object_path_get_string(main_obj, "clargs/fileext"));

MACSIO_UTILS_RecordOutputFiles(dumpn, fileName);

/* Wait for MACSIO_MIF to give this processor exclusive access */
myFile = (FILE *) MACSIO_MIF_WaitForBaton(bat, fileName, 0);

/* #warning FIX THE STRING THAT WE PRODUCE HERE SO ITS A SINGLE JSON ARRAY */

/* This processor's work on the file is just to write its part_infos */

fprintf(myFile, "%s\n", json_object_to_json_string_ext(part_infos, json_c_to_string_pretty));

MACSIO_MIF_HandOffBaton(bat, myFile);
MACSIO_MIF_Finish(bat);

/* decrement ref-count (and free) part_infos */
json_object_put(part_infos);
}

rief Method to register this plugin with MACSio main

Due to its use to initialize a file-scope, static const variable, this function winds up being called at load time (e.g. before main is even called).

Its purpose is to add key information about this plugin to MACSio's global interface table.

static int register_this_interface()
{
    MACSIO_IFACE_Handle_t iface;

    if (strlen(iface_name) >= MACSIO_IFACE_MAX_NAME)
        MACSIO_LOG_MSG(Die, ("Interface name "%s" too long", iface_name));

    strcpy(iface.name, iface_name);
    strcpy(iface.ext, iface_ext);
    iface.dumpFunc = main_dump;
    iface.processArgsFunc = process_args;

    /* Register this plugin */
    if (!MACSIO_IFACE_Register(&iface))
        MACSIO_LOG_MSG(Die, ("Failed to register interface "%s", iface_name));

    return 0;
}

3.3. Plugins
316 \brief Dummy initializer to trigger register_this_interface by the loader
317
318 This one statement is the only statement requiring compilation by
319 a C++ compiler. That is because it involves initialization and non
320 constant expressions (a function call in this case). This function
321 call is guaranteed to occur during *initialization* (that is before
322 even 'main' is called) and so will have the effect of populating the
323 iface_map array merely by virtue of the fact that this code is linked
324 with a main.
325 */
326 static int const dummy = register_this_interface();
327
328 /*!@*/
329
330 /*!@*/

3.3.2 HDF5 Plugin

The ‘HDF5’ plugin is designed to support both MIF and SSF parallel I/O paradigms. In SSF mode, writes can be
collective or independent. In MIF mode, only independent writes are supported because each task creates its own
datasets.

In addition, it supports Gzip, szip and zfp compression but only in MIF mode.

Finally, there are a number of command-line arguments to the plugin that allow control of a number of low-level
‘HDF5’ features. More can be easily added.

There have been some recent additions to help improve scalable performance of the HDF5_ library and support for
these features are yet to have been added to the plugin.

There is a minor difference in the representation of the mesh and its field between the MIF and SSF parallel I/O
modes. In MIF mode, each task outputs a chunk of mesh such that nodes on the exterior of the chunk are duplicates
of the equivalent nodes on neighboring chunks. However, in SSF mode, the whole mesh is assembled into a single,
monolithic whole mesh. The nodes that are duplicated in neighboring mesh chunks are factored out of the actual write
requests to HDF5 to avoid lower levels in the I/O stack from having to handle coincident writes.

Command-Line Arguments

Note: Need to auto-gen the command-line args doc from the source code or a run of the executable with –help

Dependencies

To use either gzip or szip compression with the ‘HDF5’_ plugin, the HDF5 library to which MACSio is linked must
have been compiled with support for zlib and szip.

However, to use zfp compression with the ‘HDF5’_ plugin, all that is necessary is that the H5Z-ZFP filter be available
to the ‘HDF5’_ library to load at run time as any other filter plugin. The H5Z-ZFP filter must have been compatably
compiled. In addition, the environment variable, HDF5_PLUGIN_PATH set to include a directory containining the
H5Z-ZFP filter plugin library. For example,

eenv HDF5_PLUGIN_PATH=<path-to-plugin-dir> mpirun -np 4 ./macsio ...

Chapter 3. Welcome to MACSio Documentation
Defines

**H5Pset_zfp_rate_cdata** (R, N, CD)
H5Z-ZFP generic interface for setting rate mode.

**H5Pset_zfp_precision_cdata** (P, N, CD)
H5Z-ZFP generic interface for setting precision mode.

**H5Pset_zfp_accuracy_cdata** (A, N, CD)
H5Z-ZFP generic interface for setting accuracy mode.

**H5Pset_zfp_expert_cdata** (MiB, MaB, MaP, MiE, N, CD)
H5Z-ZFP generic interface for setting expert mode.

**MAINZER_PARAMS**

**Typedefs**

```c
typedef struct _user_data user_data_t
User data for MIF callbacks.
```

**Functions**

```c
static hid_t make_fapl ()
create HDF5 library file access property list

static int get_tokval (char const *src_str, char const *token_to_match, void *val_ptr)
Utility to parse compression string command-line args.

Parameters

- src_str: CL arg string to be parsed
- token_to_match: a token in the string to be matched including a trailing scanf format specifier
- val_ptr: Pointer to memory where parsed value should be placed

Does a case-insensitive search of src_str for token_to_match not including the trailing format specifier. Upon finding a match, performs a scanf at the location of the match into temporary memory confirming the scan will actually succeed. Then, performs the scanf again, storing the result to the memory indicated in val_ptr.

Return 0 on error, 1 on success

static hid_t make_dcpl (char const *alg_str, char const *params_str, hid_t space_id, hid_t dtype_id)
create HDF5 library dataset creation property list

Parameters

- alg_str: compression algorithm string
- params_str: compression params string
- space_id: HDF5 dataspace id for the dataset
- dtype_id: HDF5 datatype id for the dataset
```
If the dataset size is below the \texttt{minsize} threshold, no special storage layout or compression action is taken. Chunking is initially set to \textit{single-chunk}. However, for szip compressor, chunking can be set by command-line arguments.

\textbf{static int process_args (int argi, int argc, char *argv[])}

Process command-line arguments an set local variables.

\textbf{Parameters}

- \textit{argi}: argument index to start processing \texttt{argv}
- \textit{argc}: argc from main
- \textit{argv}: argv from main

\textbf{static void main_dump_sif (json_object *main_obj, int dumpn, double dump))}

Single shared file implementation of main dump.

\textbf{Parameters}

- \textit{main_obj}: main json data object to dump
- \textit{dumpn}: dump number (like a cycle number)
- \textit{dumpn}: dump time

\textbf{static void *CreateHDF5File (const char *fname, const char *nsname, void *userData)}

MIF create file callback for HDF5 MIF mode.

\textbf{Parameters}

- \textit{fname}: file name
- \textit{nsname}: current task namespace name
- \textit{userData}: user data specific to current task

\textbf{static void *OpenHDF5File (const char *fname, const char *nsname, MACSIO_MIF_ioFlags_t ioFlags, void *userData)}

MIF Open file callback for HFD5 plugin MIF mode.

\textbf{Parameters}

- \textit{fname}: filename
- \textit{nsname}: namespace name for current task
- \textit{userData}: task specific user data for current task

\textbf{static int CloseHDF5File (void *file, void *userData)}

MIF close file callback for HDF5 plugin MIF mode.

\textbf{Parameters}

- \textit{file}: void* to hid_t of file to close
- \textit{userData}: task specific user data

\textbf{static void write_mesh_part (hid_t h5loc, json_object *part_obj)}

Write individual mesh part in MIF mode.
Parameters

- h5loc: HDF5 group id into which to write
- part_obj: JSON object for the mesh part to write

```c
static void main_dump_mif (json_object *main_obj, int numFiles, int dumpn, double dumpt)
Main dump output for HDF5 plugin MIF mode.
```

Parameters

- main_obj: main data object to dump
- numFiles: MIF file count
- dumpn: dump number (like a cycle number)
- dumpt: dump time

```c
static void main_dump (int argi, int argc, char **argv, json_object *main_obj, int dumpn, double dumpt)
Main dump callback for HDF5 plugin.
```

Parameters

- argi: arg index at which to start processing argv
- argc: argc from main
- argv: argv from main
- main_obj: main json data object to dump
- dumpn: dump number
- dumpt: dump time

Selects between MIF and SSF output.

```c
static int register_this_interface()
Function called during static initialization to register the plugin.
```

Variables

- char const *iface_name = "hdf5"
  name of this plugin
- char const *iface_ext = "h5"
  file extension for files managed by this plugin
- int use_log = 0
  Use HDF5's logging fapl
- int no_collective = 0
  Use HDF5 independent (e.g. not collective) I/O
- int no_single_chunk = 0
  disable single chunking
- int silo_block_size = 0
  block size for silo block-based VFD
- int silo_block_count = 0
  block count for silo block-based VFD
int sbuf_size = -1
    HDF5 library sieve buf size
int mbuf_size = -1
    HDF5 library meta blocck size
int rbuf_size = -1
    HDF5 library small data block size
int lbuf_size = 0
    HDF5 library log flags
const char *filename
hid_t fid
hid_t dspc = -1
int show_errors = 0
char compression_alg_str[64]
char compression_params_str[512]
int dummy = register_this_interface()
    Static initializer statement to cause plugin registration at link time.
    this one statement is the only statement requiring compilation by a C++ compiler. That is because it involves
    initialization and non constant expressions (a function call in this case). This function call is guaranteed to occur
    during initialization (that is before even ‘main’ is called) and so will have the effect of populating the iface_map
    array merely by virtue of the fact that this code is linked with a main.

struct _user_data
    User data for MIF callbacks.

    Public Members

    hid_t groupId
        HDF5 hid_t of current group

3.4 Parallel I/O Paradigms

MACSio provides utility packages to support the development of plugins handling a variety of parallel I/O paradigms
and variations thereof. These include

• Multiple Independent File (MIF) parallel I/O * With and without SCR * With and without naive Two-Phase
• Single, Shared File (SSF) parallel I/O
• Hybrid MIF/SSF where each MIF file is handled via SSF
• Multi-Pass
• Application managed/extra I/O nodes.

To be clear, MACSio itself does NOT implement any of these parallel I/O paradigms. Rather, MACSio provides
some utility packages that enable plugin developers to implement various parallel I/O paradigms. These packages are
described here.
3.4.1 Multiple Independent File (MIF) Parallel I/O

MACSio’s Multiple Independent File (MIF) package is designed to support the development of plugins utilizing the MIF paradigm.

In the multiple, independent file paradigm, parallelism is achieved through simultaneous access to multiple files. The application divides itself into file groups. For each file group, the application manages exclusive access among all the tasks of the group. I/O is serial within groups but parallel across groups. The number of files (groups) is wholly independent from the number of processors or mesh-parts. It is often chosen to match the number of independent I/O pathways available in the hardware between the compute nodes and the filesystem. The MIF paradigm is sometimes also called N->M because it is N tasks writing to M files (M<N).

In this paradigm I/O requests are almost exclusively independent. However, there are scenarios where collective I/O requests can be made to work and might even make sense in the MIF paradigm. MIF is often referred to as Poor Man’s parallel I/O because the onus is on the application to manage the distribution of data across potentially many files. In truth, this illuminates the only salient distinction between Single Shared File (SSF) and MIF. In either paradigm, if you dig deep enough into the I/O stack, you soon discover that data is always being distributed across multiple files. The only difference is where in the stack the distribution into files is handled. In the MIF paradigm, it is handled explicitly by the application itself. In a typical SSF paradigm, it is handled by the underlying filesystem.

In MIF, MPI tasks are divided into N groups and each group is responsible for creating one of the N files. At any one moment, only one MPI task from each group has exclusive access to the file. Hence, I/O is serial within a group. However, because one task in each group is writing to its group’s own file, simultaneously, I/O is parallel across groups.

A call to `MACSIO_MIF_Init()` establishes the mapping of tasks to file groups.

Within a group, access to the group’s file is handled in a round-robin fashion. The first MPI task in the group creates the file and then iterates over all mesh pieces it has. For each mesh piece, it creates a sub-directory within the file (e.g., a separate namespace for the objects). It repeats this process for each mesh piece it has. Then, the first MPI task closes the file and hands off exclusive access to the next task in the group. That MPI task opens the file and iterates over all
domains in the same way. Exclusive access to the file is then handed off to the next task. This process continues until all processors in the group have written their domains to unique sub-directories in the file.

Calls to `MACSIO_MIF_WaitForBaton()` and `MACSIO_MIF_HandOffBaton()` handle this handshaking of file creations and/or opens and bracket blocks of code that are performing MIF I/O.

After all groups have finished with their files, an optional final step may involve creating a master file which contains special metadata objects that point at all the pieces of mesh (domains) scattered about in the N files.

A call to `MACSIO_MIF_Finish()` frees up resources associated with handling MIF mappings.

The basic coding structure for a MIF I/O operation is as follows...

```c
MACSIO_MIF_baton_t bat = MACSIO_MIF_Init(...);
SomeFileType *fhndl = (SomeFileType *) MACSIO_MIF_WaitForBaton(bat, "GroupName", "ProcName");
    /* this task's work on the file */
MACSIO_MIF_HandOffBaton(bat, fhndl);
MACSIO_MIF_Finalize(bat);
```

Setting N to be equal to the number of MPI tasks, results in a file-per-process configuration, which is typically not recommended. However, some applications do indeed choose to run this way with good results. Alternatively, setting N equal to 1 results in effectively serializing the I/O and is also certainly not recommended. For large, parallel runs, there is typically a sweet spot in the selection of N which results in peak I/O performance rates. If N is too large, the I/O subsystem will likely be overwhelmed; setting it too small will likely underutilize the system resources. This is illustrated of files and MPI task counts.

![Raw Time to Write Restart File](image)

Fig. 3: Time to write a restart file from Ale3d as a function of MPI Task count and MIF file count (circa 1999)

This approach to scalable, parallel I/O was originally developed in the late 1990s by Rob Neely, a lead software architect on ALE3D at the time. It and variations thereof have since been adopted by many codes and used productively through several transitions in orders of magnitude of MPI task counts from hundreds then to hundreds of thousands today.

There are a large number of advantages to MIF-IO over SSF-IO.

- MIF enables the use of serial (e.g. non-parallel) I/O libraries. One caveat is that such libraries do need to provide namespace features (e.g. sub-directories).
• **MIF** is a much simpler programming model because it frees developers from having to think in terms of collective I/O operations. The code to write data from one MPI task doesn’t depend on or involve other tasks. For large multi-physics applications where the size, shape and even existence of data can vary dramatically among MPI tasks, this is invaluable in simplifying I/O code development.

• **MIF** alleviates any need for global-to-local and local-to-global remapping upon every exchange of data between the application and its file.

• Some Data-in-transit (DIT) services (e.g. those that may change size or shape such as compression) are easier to apply in a MIF setting because processors are freed from having to coordinate with each other on changes in data size/shape as it is moved to the file.

• Good performance demands very little in the way of extra/advanced features from the underlying I/O hardware and filesystem. A relatively dumb filesystem can get it right and perform well.

• Application controlled throttling of I/O is easily supported in a MIF setting because the number of concurrent operations is explicitly controlled. This can help to avoid overloading the underlying I/O subsystems.

• MIF is consistent with the way leading-edge commercial big data I/O in map-reduce operations is handled. Data sets are broken into pieces and stored in the filesystem as a collection of shards and different numbers of parallel tasks can process different numbers of shards.

### MACSio’s MIF Package and Naive Two-Phase I/O

**Warning:** FEATURE CURRENTLY IN DESIGN

An additional option available in MACSio’s MIF package is support for a *naive* form of two-phase I/O. In this mode, I/O requests from each parallel task are accumulated into a local ram-disk or other form of node-local storage. This process proceeds in an embarrassingly parallel way until either a threshold of storage is reached or the plugin explicitly indicates a flush is desired. At this point, all parallel tasks block while local data accumulated on each task is aggregated to aggregator tasks where the actual I/O is performed. To accomplish this, MACSio implements what amount to a virtual file such that all I/O operations performed by a task are captured in this virtual file. Aggregation is performed by message passing these virtual files to aggregator tasks which essentially turn around and replay.

### MACSio’s MIF Package and SCR

These MIF utilities are designed to support use in conjunction with the Scalable Checkpoint / Restart (SCR) library. However, use of SCR may place additional restrictions on the tasks-to-files mapping depending, partially, on whether SCR is configured to write to node-local storage. For example, SCR is typically supported only in file-per-processor mappings.

#### 3.4.2 MIF API

```c
struct _MACSIO_MIF_ioFlags_t
```

Bit Field struct for I/O flags.

**Public Members**

```c
unsigned int do_wr
   bit0: 1=write, 0=read
```

3.4. Parallel I/O Paradigms
unsigned int use_scr
    bit1: 1=use SCR, 0=don’t use SCR

group MACSIO_MIF
    Utilities supporting Multiple Independent File (MIF) Parallel I/O.

Defines

MACSIO_MIF_READ
MACSIO_MIF_WRITE

Typedefs

typedef struct _MACSIO_MIF_ioFlags_t MACSIO_MIF_ioFlags_t
    Bit Field struct for I/O flags.

typedef struct _MACSIO_MIF_baton_t MACSIO_MIF_baton_t
    Opaque struct holding private implementation of MACSIO_MIF_baton_t.

typedef void (*)(*MACSIO_MIF_CreateCB)(const char *fname, const char *nsname, void *udata)

typedef void (*)(*MACSIO_MIF_OpenCB)(const char *fname, const char *nsname, MACSIO_MIF_ioFlags_t ioFlags, void *udata)

typedef int (*)(*MACSIO_MIF_CloseCB)(void *file, void *udata)

Functions

MACSIO_MIF_baton_t *MACSIO_MIF_Init(int numFiles, MACSIO_MIF_ioFlags_t ioFlags, MPI_Comm mpiComm, int mpiTag, MACSIO_MIF_CreateCB createCb, MACSIO_MIF_OpenCB openCb, MACSIO_MIF_CloseCB closeCb, void *clientData)
 Initialize MACSIO_MIF for a MIF I/O operation.

Parameters

• numFiles: Number of resultant files. Note: this is entirely independent of number of tasks. Typically, this number is chosen to match the number of independent I/O pathways between the nodes the application is executing on and the filesystem. Pass MACSIO_MIF_MAX for file-per-processor. Pass MACSIO_MIF_AUTO (currently not supported) to request that MACSIO_MIF determine and use an optimum file count.

• ioFlags: See MACSIO_MIF_ioFlags_t for meaning of flags.

• mpiComm: The MPI communicator containing all the MPI ranks that will marshall data in the MIF I/O operation.

• mpiTag: MPI message tag MACSIO_MIF will use in all MPI messages for this MIF I/O context.

• createCb: Callback MACSIO_MIF should use to create a group’s file

• openCb: Callback MACSIO_MIF should use to open a group’s file

• closeCb: Callback MACSIO_MIF should use to close a group’s file

• clientData: Optional, client specific data MACSIO_MIF will pass to callbacks
Creates and returns a MACSIO_MIF baton object establishing the mapping between MPI ranks and file groups for a MIF I/O operation.

All processors in the mpiComm communicator must call this function collectively with identical values for numFiles, ioFlags, and mpiTag.

If a given execution context has multiple active MACSIO_MIF_baton_t objects, the caller must ensure that each passed a different mpiTag.

The resultant baton object is used in subsequent calls to WaitFor and HandOff the baton to the next processor in each group.

The createCb, openCb, closeCb callback functions are used by MACSIO_MIF to execute baton waits and handoffs during which time a group’s file will be closed by the HandOff function and opened by the WaitFor method except for the first processor in each group which will create the file.

Processors in the mpiComm communicator are broken into numFiles groups. If there is a remainder, $R$, after dividing the communicator size into numFiles groups, then the first $R$ groups will have one additional processor.

Return The MACSIO_MIF baton object

```c
void MACSIO_MIF_Finish(MACSIO_MIF_baton_t *bat)
```
End a MACSIO_MIF I/O operation and free resources.

Parameters

- bat: The MACSIO_MIF baton handle

```c
void *MACSIO_MIF_WaitForBaton(MACSIO_MIF_baton_t *Bat, char const *fname, char const *nsname)
```
Wait for exclusive access to the group’s file.

Parameters

- Bat: The MACSIO_MIF baton handle
- fname: The filename
- nsname: The namespace within the file to be used for this task’s objects.

All tasks in mpiComm argument to MACSIO_MIF_Init() call this function collectively. For the first task in each group, this call returns immediately. For all others in the group, it blocks, waiting for the task before it to finish its work on the group’s file and call MACSIO_MIF_HandOffBaton().

Return A void pointer to whatever data instance the createCb or openCb methods return. The caller must cast this returned pointer to the correct type.

```c
int MACSIO_MIF_HandOffBaton(MACSIO_MIF_baton_t const *Bat, void *file)
```
Release exclusive access to the group’s file.

Parameters

- Bat: The MACSIO_MIF baton handle
- file: A void pointer to the group’s file handle

This function is called only by the current task holding exclusive access to a group’s file and closes the group’s file for the calling task handing off control to the next task in the group.
Return  The integer value returned from the MACSIO_MIF_CloseCB callback.

```c
int MACSIO_MIF_RankOfGroup (MACSIO_MIF_baton_t const *Bat, int rankInComm)
```
Rank of the group in which a given (global) rank exists.

**Parameters**
- **Bat**: The MACSIO_MIF baton handle
- **rankInComm**: The (global) rank of a task for which the rank of its group is desired

Given the rank of a task in mpiComm used in the MACSIO_MIF_Init() call, this function returns the rank of the group in which the given task exists. This function can be called from any rank and will return correct values for any rank it is passed.

```c
int MACSIO_MIF_RankInGroup (MACSIO_MIF_baton_t const *Bat, int rankInComm)
```
Rank within a group of a given (global) rank.

**Parameters**
- **Bat**: The MACSIO_MIF baton handle
- **rankInComm**: The (global) rank of a task for which it’s rank in a group is desired

Given the rank of a task in mpiComm used in the MACSIO_MIF_Init() call, this function returns its rank within its group. This function can be called from any rank and will return correct values for any rank it is passed.

```c
struct _MACSIO_MIF_ioFlags_t
#include <macsio_mif.h> Bit Field struct for I/O flags.
```

**Public Members**

```c
unsigned int do_wr
   bit0: 1=write, 0=read

unsigned int use_scr
   bit1: 1=use SCR, 0=don’t use SCR
```

### 3.4.3 Single Shared File (SSF) Parallel I/O

### 3.5 Command Line Argument Parsing

The command line argument package simplifies a number of aspects of defining and also parsing command line arguments for MACSio’s main code or any of its plugins. The package supports command line arguments and their options with the following features...

- boolean arguments (the argument’s presence or absence maps to 0 or 1)
- integer, double and string valued arguments
- multiple arguments for a given command-line option
- default values for arguments
- option groupings
- help strings and printing of usage with --help option
• pretty (somewhat) formatting of long help strings to terminal width
• separating arguments into sub-command groups.
• routing the parsed results to caller supplied local variables or constructing a returned JSON object.
• error detection and logging
• collective execution in parallel * argc, argv inputs ensured consistent across all tasks * results collectively unified across all tasks in parallel

The package is designed to do parsing of simple command-line arguments and assign values associated with them either to caller-supplied scalar variables or to populate a json_object. The work-horse function, MACSIO_CLARGS_ProcessCmdline(), is used in the following manner...

After the argi, argc, argv trio of arguments, the remaining arguments come in groups of either 3 (when mapping to a json_object) or 4 (when mapping to scalar program variables).

The first of these is an argument format specifier much like a printf statement. It indicates the type of each parameter to the argument as well as the number of parameters. Presently, it understands only %d, %f and %s types.

The second is a string argument indicating the default value, as a string.

The third is a help string for the argument. Note, you can make this string as long as the C-compiler will permit. You should not embed any newline characters as the routine to print the help string will do that for you. However, you may embed newline characters if you wish to control specific line breaking when output.

The fourth argument is present only when mapping to scalar program variables and is a pointer to the variable location(s) where values will be stored.

Command line arguments for which only existence on the command-line is tested assume a caller-supplied return value of int and will be assigned 1 if the argument exists on the command line and 0 otherwise.

Do not name any argument with a substring help as that is reserved for obtaining help. Also, do not name any argument with the string end_of_args as that is used to indicate the end of the list of arguments passed to the function.

If any argument on the command line has the substring help, help will be printed by task 0 and then this function calls MPI_Finalize() (in parallel) and exit(1).

This function must be called collectively in MPI_COMM_WORLD. All tasks are guaranteed to complete with identical results.

3.5.1 CLARGS API

group MACSIO_CLARGS
    Command-line argument parsing utilities.

Defines

MACSIO_CLARGS_WARN
MACSIO_CLARGS_ERROR
MACSIO_CLARGS_TOOMEM
MACSIO_CLARGS_TOJSON
MACSIO_CLARGS_ASSIGN_OFF
MACSIO_CLARGS_ASSIGN_ON
MACSIO_CLARGS_HELP
MACSIO_CLARGS_OK
MACSIO_CLARGS_GRP_SEP_STR
MACSIO_CLARGS_GRP_BEG
MACSIO_CLARGS_GRP_END
MACSIO_CLARGS_ARG_GROUP_BEG (GRPNAME, GRPHELP)
    Begin option group (for TOJSON routing invocation)
MACSIO_CLARGS_ARG_GROUP_END (GRPNAME)
    End option group (for TOJSON routing invocation)
MACSIO_CLARGS_ARG_GROUP_BEG2 (GRPNAME, GRPHELP)
    Begin option group (for TOMEM routing invocation)
MACSIO_CLARGS_ARG_GROUP_END2 (GRPNAME)
    End option group (for TOMEM routing invocation)
MACSIO_CLARGS_END_OF_ARGS
    Moniker to terminate argument list.
MACSIO_CLARGS_NODEFAULT
    Value to indicate no default specified.

Typedefs

typedef struct _MACSIO_CLARGS_ArgvFlags_t MACSIO_CLARGS_ArgvFlags_t

Functions

int MACSIO_CLARGS_ProcessCmdline (void **retobj, MACSIO_CLARGS_ArgvFlags_t flags, int argi, int argc, char **argv, ...)

    Command-line argument parsing, default values and help output.

Parameters

    • retobj: Optional void pointer to a returned object encoding the command line options. Currently only JSON_C object is supported. Must specify TOJSON routing.
    • argi: First arg index at which to to start processing argv
    • argc: argc from main
    • argv: argv from main

Avoid defining any argument with the substring --help or --no-strict. These are command-line option keywords reserved by MACSio. --help is a request to print usage and exit. no-strict disables strict command-line option error response which ordinarily causes an abort and instead issues warnings.

In parallel, this function must be called collectively by all ranks in MPI_COMM_WORLD. argc and argv on task rank 0 are broadcast to all tasks. If request for help (e.g. --help is in argv) or error(s) are encountered, all tasks are broadcast this outcome. Only task rank 0 will print usage or error messages.

Return An integer value of either MACSIO_CLARGS_OK, MACSIO_CLARGS_HELP or MACSIO_CLARGS_ERROR.
struct _MACSIO_CLARGS_ArgvFlags_t
#include <macsio_clargs.h>

Public Members

unsigned int error_mode
0=warn, 1=abort

unsigned int route_mode
0=(TOMEM) scalar variables, 1=(TOJSON) json_object, 2,3 unused

unsigned int defaults_mode
0=assign defaults, 1=do not assign defaults

3.5.2 Example

```c
int doMultifile, numCycles, Ni, Nj;
double Xi, Xj;
MACSIO_CLARGS_ArgvFlags_t flags;
json_object *obj = 0;
int argi = 1; /* start at argv[1] not argv[0] */
flags.error_mode = MACSIO_CLARGS_ERROR;
flags.route_mode = MACSIO_CLARGS_TOMEM;

MACSIO_CLARGS_ProcessCmdline(obj, flags, argi, argc, argv,
    "--multifile", "", /* example boolean with no default */
    "if specified, use a file-per-timestep",
    &doMultifile,
    "--numCycles %d", "10", /* example integer with default of 10 */
    "specify the number of cycles to run",
    &numCycles,
    "--dims %d %f %d %f", "25 5 35 7", /* example multiple values */
    "specify size (logical and geometric) of mesh",
    &Ni, &Xi, &Nj, &Xj,
MACSIO_CLARGS_END_OF_ARGS);
```

3.6 Timing

MACSio's timing package includes utilities to support the creation of a number of timers to time sections of code.

Timers are initialized/started with a user-defined label, and an optional group mask and iteration number. A hash of the timer is computed from its label and group mask combined with the source file name and line number. The resulting hash value is used to identify the timer and returned as the timer's id when the timer is created/started.

Timers can be iterated and re-started. These are two different concepts. During iteration, the same timer can be used to time the same section of code as it gets executed multiple times. Each time represents a different iteration of the
timer. The timer will keep track of the minimum time, maximum time, average time and variance of times over all iterations it is invoked. However, apart from these running statistics, a timer maintains no memory of past values. If a timer is iterated 10 times, it does not maintain knowledge of all 10 individual times. It maintains only knowledge of the running statistics; min, max, avg, var. The algorithm it uses to maintain these running statistics is the Welford Online algorithm.

Within any timer iteration, a timer can be stopped and restarted. If a timer is being used to time a block of code that involves some occasional and perhaps irrelevant alternative logic path, the timer can be stopped until execution returns from the alternative path. Then the timer can be restarted. This can only be done within a given iteration.

Timers can also be grouped into a small number (<64) of different groups. A timer group is created with a call to MACSIO_TIMING_GroupMask(). Timers are included in one or more groups by or’ing the associated group masks into the gmask argument to MACSIO_TIMING_StartTimer(). A timer can be a member of multiple groups. For example, it is possible to maintain a set of timers for MIF file I/O apart from a set of timers being used to time a particular plugin’s operations to marshal a mesh or variable to/from persistent storage. Or, operations use to interact with filesystem metadata directly (e.g. mkdir, chdir, readdir, etc.) can be maintained separately from timers used for entirely other purposes.

Finally, timers can be reduced parallel tasks thereby creating a statistical summary of timer information across tasks.

A timer is initialized/started by a call to MACSIO_TIMING_StartTimer() or the shorthand macro MT_StartTimer. This call returns the timer’s ID which is used in a subsequent call to MACSIO_TIMING_StopTimer() to stop the timer.

```c
MACSIO_TIMING_TimerId_t tid = MT_StartTimer("my timer", MACSIO_TIMING_GROUP_NONE,
                                           MACSIO_TIMING_ITER_AUTO);

...do some work here...

MT_StopTimer(tid);
```

In the above code, the call to MT_StartTimer starts a timer for a new (automatic) iteration. In this simple example, we do not worry about timer group masks.

By default, the timing package uses MPI_Wtime() but a caller can set MACSIO_TIMING_UseMPI_Wtime to zero to instead use gettimeofday().

Examples of the use of the logging package can be found in Example (tsttiming.c).

### 3.6.1 Timing API

`group MACSIO_TIMING`

Timing utilities.

`Defines`

- **MACSIO_TIMING_ITER_AUTO**
  
  Automatic iteration numbering. Use for `iter` arg to `MT_StartTimer()` for automatic iteration numbering of the timer explicitly.

- **MACSIO_TIMING_ITER_IGNORE**
  
  What is this?

- **MACSIO_TIMING_INVALID_TIMER**
  
  May be returned from `MT_StartTimer()`

- **MACSIO_TIMING_NO_GROUP**
  
  Group mask when timer is not assigned to any group.
MACSIO_TIMING_ALL_GROUPS
Group mask representing all groups.

MT_Time
Shorthand for MACSIO_TIMING_GetCurrentTime()

_BASEFILE_
Same as basename(FILE) Used to strip unnecessary path terms from in messages.

MT_StartTimer (LAB, GMASK, ITER)
Shorthand for MACSIO_TIMING_StartTimer()

Parameters
• LAB: User defined timer label string
• GMASK: User defined group mask. Use MACSIO_TIMING_NO_GROUP if timer grouping is not needed.
• ITER: The iteration number. Use MACSIO_TIMING_ITER_IGNORE if timer iteration is not needed.

MT_StopTimer (ID)
Shorthand for MACSIO_TIMING_StopTimer()

Parameters
• ID: The timer’s hash id returned from a call to MT_StartTimer().

Typedefs
typedef unsigned int MACSIO_TIMING_TimerId_t
typedef unsigned long long MACSIO_TIMING_GroupMask_t

Functions
MACSIO_TIMING_GroupMask_t MACSIO_TIMING_GroupMask (char const *grpName)
Create a group name and mask.

Parameters
• grpName: Name of the group for which group mask is needed

A small number of groups (less than 64) can be defined into which timers can be grouped Timers can be assigned to multiple groups by or’ing the resulting group masks.

MACSIO_TIMING_TimerId_t MACSIO_TIMING_StartTimer (char const *label, MACSIO_TIMING_GroupMask_t gmask, int iter_num, char const *file, int line)
Create/Start a timer.

Parameters
• label: User defined label to be assigned to the timer
• gmask: Mask to indicate the timer’s group membership
• iter_num: Iteration number
• file: The source file name
• line: The source file line number

This call either creates a new timer and starts it or starts a new iteration of an existing timer.

**Return** A hash derived from a string concatenation the label, gmask, file and line.

```c
double MACSIO_TIMING_StopTimer(MACSIO_TIMING_TimerId_t id)
```

Stop a timer.

**Parameters**

• id: The timer's ID, returned from `MACSIO_TIMING_StartTimer()`

This call stops a currently running timer.

**Return** Returns the time for the current iteration of the timer

```c
double MACSIO_TIMING_GetTimerDatum(MACSIO_TIMING_TimerId_t tid, char const *field)
```

Get specific data field from a timer.

**Parameters**

• tid: The timer's ID, returned from `MACSIO_TIMING_StartTimer()`
• field: The name of the field from the timer to return

Possible field names are...

• "start_time" time last iteration was started
• "iter_count" number of times the timer has been triggered (iterated)
• "iter_time" total time over all iterations
• "min_iter" iteration at which minimum time was observed
• "max_iter" iteration at which maximum time was observed
• "min_rank" task rank where minimum time occurred (only valid on reduced timers)
• "max_rank" task rank where maximum time occurred (only valid on reduced timers)
• "depth" how deeply this timer is nested within other timers
• "total_time" total time spent in this timer
• "min_time" minimum time observed for this timer
• "max_time" maximum time observed for this timer
• "running_mean" current average time for this timer
• "running_var" current variance for this timer

Where applicable, returned values are over either

• all iterations (when using non-reduced timers) or
• all iterations and ranks (when using reduced timers).
double MACSIO_TIMING_GetReducedTimerDatum(MACSIO_TIMING_TimerId_t tid, char const *field)

Get data field a specific reduced timer.

Parameters

• tid: The timer’s ID, returned from MACSIO_TIMING_StartTimer()
• field: The name of the field from the timer to return

For field names, see MACSIO_TIMING_GetTimerDatum()

void MACSIO_TIMING_DumpTimersToStrings(MACSIO_TIMING_GroupMask_t gmask, char ***strs, int *nstrs, int *maxlen)

Dump timers to ascii strings.

Parameters

• gmask: Group mask to filter only timers belonging to specific groups
• strs: Array of strings, one for each timer, returned to caller. Caller must free.
• nstrs: Number of strings returned to caller
• maxlen: The maximum length of all strings

This call will find all used timers in the hash table matching the specified gmask group mask and dumps each timer to a string. For convenience, the maximum length of the strings is also returned. This is to facilitate dumping the strings to a MACSIO_LOG. Pass MACSIO_TIMING_ALL_GROUPS if you don’t care about timer groups.

Each timer is dumped to a string with the following informational fields catenated together and separated by colons...

• TOT=%10.5f summed total time spent in this timer over all iterations (and over all ranks when reduced)
• CNT=%04d summed total total number of iterations in this timer (and over all ranks when reduced)
• MIN=%8.5f(%.2f):%06d
  – minimum time recorded for this timer over all iterations (and ranks when reduced)
  – (%4.2f) the number of standard deviations of the min from the mean time
  – :%06d task rank where the minimum was observed (only valid when reduced)
• AVG=%8.5f the mean time of this timer over all iterations (and ranks when reduced)
• MAX=%8.5f(%.2f):%06d,
  – maximum time recorded for this timer over all iterations (and ranks when reduced)
  – (%4.2f) the number of standard deviations of the max from the mean time
  – :%06d task rank where the maximum was observed. (only valid when reduced)
• DEV=%8.8f standard deviation observed for all iterations of this timer
• FILE=%s the source file where this timer is triggered
• LINE=%d the source line number where this timer is triggered
• LAB=%s the user-defined label for this timer
void MACSIO_TIMING_ReduceTimers (MPI_Comm comm, int root)
Reduce timers across MPI tasks.

Parameters

• comm: The MPI communicator to use for the reduction
• root: The MPI rank of the root task to be used for the reduction

Computes a parallel reduction across MPI tasks of all timers.

void MACSIO_TIMING_DumpReducedTimersToStrings (MACSIO_TIMING_GroupMask_t gmask, char ***strs, int *nstrs, int *maxlen)
Dump reduced timers to ascii strings.

Parameters

• gmask: Group mask to filter only timers belonging to specific groups
• strs: Array of strings, one for each timer, returned to caller. Caller must free
• nstrs: Number of strings returned to caller
• maxlen: The maximum length of all strings

Similar to DumpTimersToStrings except this call dumps reduced timers

void MACSIO_TIMING_ClearTimers (MACSIO_TIMING_GroupMask_t gmask)
Clear a group of timers.

Parameters

• gmask: Group mask to filter only timers belonging to specific groups

Clears and resets timers of specified group. Pass MACSIO_TIMING_ALL_GROUPS to clear all timers.

double MACSIO_TIMING_GetCurrentTime (void)
Get current time.

Depending on current value of MACSIO_TIMING_UseMPI_Wtime, uses either MPI_Wtime() or gettimeofday().

Variables

int MACSIO_TIMING_UseMPI_Wtime
Integer variable to control function used to get timer values.

A non-zero value indicates that MACSIO_TIMING should use MPI_Wtime(). Otherwise, it will use gettimeofday().

3.6.2 Example (tsttiming.c)

1 #include <stdlib.h>
2 #include <time.h>
3 #include <unistd.h>
4
5 #ifdef HAVE_MPI
6 #include <mpi.h>
```c
#endif

#include <macsio_log.h>
#include <macsio_timing.h>

int MACSIO_MAIN_Rank;
int MACSIO_MAIN_Size;
int MACSIO_MAIN_Comm;

static void dsleep(double delay)
{
    double dsec = floor(delay);
    long nsec = (long) ((delay - dsec) * 1e+9);
    time_t sec = (time_t) dsec;
    struct timespec ts = {sec, nsec};
    nanosleep(&ts, 0);
}

void func2();

void func4()
{
    static int iter = 0;
    double delay = (random() % 1000) / (double) 10000; /* random up to 1/10th second of sleep */
    MACSIO_TIMING_TimerId_t tid = MT_StartTimer("func4", MACSIO_TIMING__ALL_GROUPS, iter++);
    dsleep(delay);
    MT_StopTimer(tid);
}

void func3()
{
    static int iter = 0;
    MACSIO_TIMING_TimerId_t tid = MT_StartTimer("func3", MACSIO_TIMING__ALL_GROUPS, iter++);
    func4();
    func2();
    MT_StopTimer(tid);
}

void func2()
{
    static int iter = 0;
    MACSIO_TIMING_TimerId_t tid = MT_StartTimer("func2", MACSIO_TIMING__ALL_GROUPS, iter++);
    dsleep(0.02);
    MT_StopTimer(tid);
}

void func1()
{
    static int iter = 0;
    MACSIO_TIMING_TimerId_t tid = MT_StartTimer("func1", MACSIO_TIMING__ALL_GROUPS, iter++);
    dsleep(0.02);
    MT_StopTimer(tid);
}
```

### 3.6. Timing
ALL_GROUPS, iter); 57 MACSIO_TIMING_TimerId_t tid2;
58 func2(); 60 dsleep(0.01);
59 tid2 = MT_StartTimer("call to func3 from func1", MACSIO_TIMING_ALL_ 61 \_GROUPS, iter++);
62 func3(); 63 MT_StopTimer(tid2);
64 func2();
65 MT_StopTimer(tid);
66 }
67 }
68 int main(int argc, char **argv)
69 {
70 int i, rank = 0, size = 1;
71 MACSIO_TIMING_TimerId_t a, b;
72 char **timer_strs;
73 int ntimer_strs, maxstrlen;
74 #ifdef HAVE_MPI
75 MPI_Init(&argc, &argv);
76 MPI_Comm_size(MPI_COMM_WORLD, &size);
77 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
78 #endif
79 #ifdef HAVE_MPI
80 MPI_Abort(MPI_COMM_WORLD, 1);
81 #endif
82 MACSIO_LOG_DebugLevel = 1; /* should only see debug messages level 1 83 \_and 2 */
84 srandom(0xDeadBeef); /* used to vary length of some timers */
85 if (size > 8)
86 {
87 if (!rank)
88 fprintf(stderr, "This test only appropriate for 8 or fewer_
89 \_processors\n");
90 #ifdef HAVE_MPI
91 MPI_Abort(MPI_COMM_WORLD, 1);
92 #endif
93 exit(1);
94 }
95 a = MT_StartTimer("main", MACSIO_TIMING_ALL_GROUPS, 0);
96 func1();
97 if (rank > 2)
98 func4();
99 b = MT_StartTimer("call to func3 from main", MACSIO_TIMING_ALL_GROUPS,
100 0);
101 func3();
102 MT_StopTimer(b);
103 Chapter 3. Welcome to MACSio Documentation
if (rank < 2) {
  func1();
  MT_StopTimer(a);
  MACSIO_TIMING_DumpTimersToStrings(MACSIO_TIMING_ALL_GROUPS, &timer_strs, &ntimer_strs, &maxstrlen);
}

#ifdef HAVE_MPI
  int rbuf[2], sbuf[2] = {ntimer_strs, maxstrlen};
  MPI_Allreduce(sbuf, rbuf, 2, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
  MACSIO_LOG_MainLog = MACSIO_LOG_LogInit(MPI_COMM_WORLD, "tsttiming.log", rbuf[1]+32, 2*rbuf[0]+4, 0);
#else
  MACSIO_LOG_MainLog = MACSIO_LOG_LogInit(0, "tsttiming.log", maxstrlen+4, ntimer_strs+4, 0);
#endif

for (i = 0; i < ntimer_strs; i++) {
  MACSIO_LOG_MSG(Dbg1, (timer_strs[i]));
  free(timer_strs[i]);
}
free(timer_strs);

#ifdef HAVE_MPI
  MACSIO_TIMING_ReduceTimers(MPI_COMM_WORLD, 0);
  if (!rank) {
    MACSIO_TIMING_DumpReducedTimersToStrings(MACSIO_TIMING_ALL_GROUPS, &timer_strs, &ntimer_strs, &maxstrlen);
    MACSIO_LOG_MSG(Dbg1, ("#####################Reduced Timer Values##
      #####################"));
    for (i = 0; i < ntimer_strs; i++) {
      MACSIO_LOG_MSG(Dbg1, (timer_strs[i]));
      free(timer_strs[i]);
    }
    free(timer_strs);
  }
#endif

MACSIO_LOG_LogFinalize(MACSIO_LOG_MainLog);
MACSIO_TIMING_ClearTimers(MACSIO_TIMING_ALL_GROUPS);
#ifdef HAVE_MPI
  MPI_Finalize();
#endif
type return 0;
3.7 Logging

MACSio’s logging package is a small utility for logging various kinds of messages from processors. This includes debugging messages, warnings and errors.

A log is a single text file that is divided into groups of message lines. Each processor gets its own group of lines within the file it may write to. Rank 0 gets the first group of lines. Rank 1, the next group of lines and so forth. Each line has a maximum length too.

When a log is created with `MACSIO_LOG_LogInit()`, the caller specifies the MPI communicator the log will be used for, the number of message lines per task to allocate plus a count of extra lines for rank 0 and the maximum length of any message. Rank 0 initializes the text file with all space characters except for header lines to distinguish each processor’s group of lines in the file. Note that for a large run on say $10^5$ processors, it may take rank 0 several seconds to initialize this file.

Messages are restricted to a single line of text. Any embedded new-line characters are removed from a message and replaced with a ‘!’ character. If a processor’s message is longer than the maximum length of a line for the log, the message is truncated to fit within the line. As processors issue messages, they are written to the processors group of lines in round-robin fashion. As the set of messages issued from a processor reaches the end of its group of lines within the file, it starts issuing new messages at the beginning of the group. So, older messages can wind up getting overwritten by newer messages. However, all the most recent messages prior to any significant error will at least be captured in the log.

Parallelism in writing to a MACSIO_LOG file is achieved by ensuring no two processor attempt to write data in overlapping regions in the file by using `pwrite()` to do the actual writes.

MACSio’s main creates a default log, `MACSIO_LOG_MainLog`, on the `MACSIO_MAIN_Comm`. That log is probably the only log needed by MACSio proper or any of its plugins. The convenience macro, `MACSIO_LOG_MSG`, is the only method one need to worry about to log messages to the main log. That macro will also capture more detailed information regarding error states around the time the message issue issued including the file and line number, the system errno and the most recent MPI error state.

If you use `MACSIO_LOG_MSG`, messages are allowed one of several severities; `Dbg1`, `Dbg2`, `Dbg3`, `Warn`, `Err` and `Die`. A severity of `Die` causes an abort. Depending on the current debug level setting, `Dbg1 - Dbg3` messages may or may not be recorded to a log.

MACSio’s main also creates a log for issuing messages to stderr, `MACSIO_LOG_StdErr`. However, there is no convenience macro like `MACSIO_LOG_MSG` for logging messages to `MACSIO_LOG_StdErr`. A caller simply has to use the logging interface methods to issue messages to `MACSIO_LOG_StdErr`.

However, any plugin or other portion of MACSio may, optionally, create its own, private, log using `MACSIO_LOG_LogInit()`. Once a log is created, any processor can independently issue messages to the log. There is no parallel communication involved in issuing messages to logs.

Examples of the use of the logging package can be found in `Example (tstlog.c)`.

3.7.1 Logging API

```
group MACSIO_LOG
    Message logging utilities.

Defines

    MACSIO_LOG_DEFAULT_LINE_COUNT
    MACSIO_LOG_DEFAULT_EXTRA_LINES
```
MACSIO_LOG_DEFAULT_LINE_LENGTH

__BASEFILE__

Same as basename(FILE) Used to strip unnecessary path terms from in messages.

MACSIO_LOG_MSG(SEV, MSG)

Convenience macro for logging a message to the main log.

Parameters

- SEV: Abbreviated message severity (e.g. ‘Dbg1’, ‘Warn’)
- MSG: Caller’s sprintf-style message enclosed in parenthises (e.g. ‘("Rank %d failed",rank)’)

MACSIO_LOG_MSGV(VSEV, MSG)

Alterantive to MACSIO_LOG_MSG when severity is a runtime variable.

Parameters

- VSEV: Runtime variable in which message severity is stored
- MSG: Caller’s sprintf-style message enclosed in parenthises (e.g. ‘("Rank %d failed",rank)’)

MACSIO_LOG_MSGL(LOG, SEV, MSG)

Convenience macro for logging a message to any specific log.

Parameters

- LOG: The log handle
- SEV: Abbreviated message severity (e.g. ‘Dbg1’, ‘Warn’)
- MSG: Caller’s sprintf-style message enclosed in parenthises (e.g. ‘("Rank %d failed",rank)’)

MACSIO_LOG_MSGLV(LOG, VSEV, MSG)

Convenience macro for logging a message with variable severity to any specific log.

Parameters

- LOG: The log handle
- VSEV: Runtime variable in which message severity is stored
- MSG: Caller’s sprintf-style message enclosed in parenthises (e.g. ‘("Rank %d failed",rank)’)

Typedefs

typedef struct _log_flags_t log_flags_t

typedef struct _MACSIO_LOG_LogHandle_t MACSIO_LOG_LogHandle_t

typedef enum_MACSIO_LOG_MsgSeverity_t MACSIO_LOG_MsgSeverity_t

typedef struct _MACSIO_LOG_LogHandle_t MACSIO_LOG_LogHandle_t

3.7. Logging
Enums

enum _MACSIO_LOG_MsgSeverity_t

Values:

MACSIO_LOG_MsgDbg1
   Debug level 1: For coarse grained debugging messages (rare enough performance isn’t effected)

MACSIO_LOG_MsgDbg2
   Debug level 2: For moderate grained debugging messages (may effect performance)

MACSIO_LOG_MsgDbg3
   Debug level 3: For fine grained debugging messages (most likely effects performance)

MACSIO_LOG_MsgInfo
   Informational messages

MACSIO_LOG_MsgWarn
   Warnings of minor problems that can be recovered from without undue effects

MACSIO_LOG_MsgErr
   Error conditions that result in unexpected behavior

MACSIO_LOG_MsgDie
   Unrecoverable errors

Functions

char const *MACSIO_LOG_MakeMsg (char const *format, ...)

Internal convenience method to build a message from a printf-style format string and args.

Parameters

• format: A printf-like error message format string.

This method is public only because it is used within the MACSIO_LOG_MSG convenience macro.

MACSIO_LOG_LogHandle_t *MACSIO_LOG_LogInit (MPI_Comm comm, char const *path,
   int line_len, int lines_per_proc, int extra_lines_proc0)

Initialize a log.

Parameters

• comm: MPI Communicator of tasks that will issue messages to this log
• path: The name of the log file
• line_len: The length of each message line in the log file
• lines_per_proc: The number of message lines for each MPI task
• extra_lines_proc0: The number of extra message lines for processor rank 0

All processors in the comm communicator must call this function collectively with identical values for path, line_len, and lines_per_proc.

void MACSIO_LOG_LogMsg (MACSIO_LOG_LogHandle_t const *log, char const *fmt, ...)

Issue a printf-style message to a log.

Parameters
log: The handle for the specified log
fmt: A printf-style format string for the log message

May be called independently by any processor in the communicator used to initialize the log.

```
void MACSIO_LOG_LogMsgWithDetails (MACSIO_LOG_LogHandle_t const *log, char const *linemsg, MACSIO_LOG_MsgSeverity_t sevVal, char const *sevStr, int sysErrno, int mpiErrno, char const *theFile, int theLine)
```

Convenience method for building a detailed message for a log.

Parameters

- **log**: Log handle to issue message to
- **linemsg**: Caller’s message string
- **sevVal**: Caller’s message severity value
- **sevStr**: Caller’s message severity abbreviation string
- **sysErrno**: Current (most recent) system’s errno
- **mpiErrno**: Current (most recent) MPI error
- **theFile**: Caller’s file name
- **theLine**: Caller’s line number within the file

```
void MACSIO_LOG_LogFinalize (MACSIO_LOG_LogHandle_t *log)
```

Finalize and close an open log. Should be called collectively by all processors that created the log.

Parameters

- **log**: The log to be closed

Variables

```
int mpi_errno
```
Error code returned by most recent MPI call.

MACSIO uses `mpi_errno` much like the system’s `errno`. However, there is no way to enforce that any particular MPI function calls set `mpi_errno`. MACSIO relies upon the honor system of developers to always make MPI calls by setting `mpi_errno` to the return value of those calls. Assuming this practice is followed throughout MACSIO and any of its plugins, then the global variable `mpi_errno` should always hold the MPI error return value of the most recent MPI call.

```
int MACSIO_LOG_DebugLevel
```
Filtering level for debugging messages.

MACSIO generates 3 levels of debug messages numbered 1, 2 and 3. Each level is intended to represent more and more detailed debugging messages. Level 1 messages are issued rare enough by MACSIO (e.g. coarse grained) that they will not impact performance. Level 3 messages can be issued often enough by MACSIO that they will almost certainly impact performance. Level 2 messages are somewhere in between. The global variable `MACSIO_LOG_DebugLevel` is used to filter what messages generated by MACSIO that will actually make it into a detailed log message. If `MACSIO_LOG_DebugLevel` is set to level N, then messages at and below N debug level will appear in a log when written via MACSIO `LogMsgWithDetails`. However, messages above this level will be discarded. As developers
write code blocks in MACSIO, developers must choose what kind of messages are issued and, for debugging kinds of messages, the appropriate debug level.

**MACSIO_LOG_LogHandle_t** MACSIO_LOG_MainLog

Log handle for MACSIO’s main log.

Typically, developers will use MACSIO_LOG_MSG macro to log messages. Such messages are automatically logged to the main log. The main log is created by MACSIO’s main.

**MACSIO_LOG_LogHandle_t** MACSIO_LOG_StdErr

Log handle for MACSIO’s stderr output.

In rare cases, developers may want to issues messages to stderr. In this case, MACSIO’s stderr log handle should be used.

```
struct _log_flags_t

Public Members

unsigned int was_logged
    Indicates if a message was ever logged to the log
```

```
struct _MACSIO_LOG_LogHandle_t

Public Members

MPI_Comm comm
    MPI Communicator of tasks that will issue messages to this log

char * pathname
    Name of the log file

int logfile
    Log file file descriptor

int rank
    Rank of the processor that created this log handle

int size
    Size of the communicator that created this log handle

int log_line_length
    Maximum length of a message line in the log file

int lines_per_proc
    Number of message lines allocated in the file for each processor

int extra_lines_proc0
    Additional number of message lines for processor with MPI rank 0

int current_line
    Index into this processor’s group of lines in the log file at which the next message will be written

log_flags_t flags
    Informational flags regarding the log
```
3.7.2 Example (tstlog.c)

1 #include <stdlib.h>
2 #include <strings.h>
3 
4 #include <macsio_log.h>
5 
6 #ifdef HAVE_MPI
7 #include <mpi.h>
8 #endif
9 
10 int MACSIO_MAIN_Rank;
11 int MACSIO_MAIN_Size;
12 int MACSIO_MAIN_Comm;
13 
14 int main (int argc, char **argv)
15 {
16     int i;
17     int rank=0, size=1;
18     int num_cols = 128, num_rows = 20, extra_lines = 20;
19 #ifdef HAVE_MPI
20         MPI_Comm comm = MPI_COMM_WORLD;
21 #else
22         int comm = 0;
23 #endif
24 
25     for (i = 0; i < argc; i++)
26         {
27             if (!strncasecmp(argv[i], "num_cols=", 9))
28                 num_cols = strtol(argv[i]+9, 0, 10);
29             else if (!strncasecmp(argv[i], "num_rows=", 9))
30                 num_rows = strtol(argv[i]+9, 0, 10);
31             else if (!strncasecmp(argv[i], "extra_lines=", 9))
32                 extra_lines = strtol(argv[i]+9, 0, 10);
33         }
34 
35 #ifdef HAVE_MPI
36         MPI_Init(&argc, &argv);
37         MPI_Comm_size(comm, &size);
38         MPI_Comm_rank(comm, &rank);
39 #endif
40 
41     MACSIO_LOG_DebugLevel = 2; /* should only see debug messages level 1  
   --and 2 */
42     MACSIO_LOG_MainLog = MACSIO_LOG_LogInit(comm, "tstlog.log", num_cols,
43         num_rows, extra_lines);
44     MACSIO_LOG_StdErr = MACSIO_LOG_LogInit(comm, 0, 0, 0, 0);
45     if (rank == 1)
46         {
47             MACSIO_LOG_MSG(Dbg1, ("I am staring with processor 1"));
48             MACSIO_LOG_MSG(Dbg2, ("Test output of a very, very, very, very,  
   --very, very, very, very, very, very, very, very, very, very, very, very,  
   --very, very, very, very, very, very, very, very, very, very, very, very,  
   --very, very, very, very, very, very, very, very, very, very, very, very,  
   --very, very, very, very, very, very, very, very, very, very, very, very,  
   --very long line");

3.7. Logging
```c
49 } 
50 else if (rank == 2) 
51 {
52     MACSIO_LOG_MSG(Warn, ("Starting on proc 2");
53     MACSIO_LOG_LogMsg(MACSIO_LOG_StdErr, "Logging a message to stderr for rank %d", rank);
54 }
55 else if (rank == 0) 
56 {
57     errno = EOVERFLOW;
58     #ifdef HAVE_MPI
59         mpi_errno = MPI_ERR_COMM;
60     #else
61         mpi_errno = 0;
62     #endif
63     MACSIO_LOG_MSG(Err, ("I am here on proc 0");
64 }
65 else 
66 {
67     int i;
68     MACSIO_LOG_MsgSeverity_t sev = rank%2?MACSIO_LOG_MsgDbg2:MACSIO_LOG_MsgDbg3;
69     for (i = 0; i < 25; i++)
70         {MACSIO_LOG_MSGV(sev, ("Outputing line %d for rank %d
", i, rank));
71     }
72     MACSIO_LOG_LogFinalize(MACSIO_LOG_StdErr);
73     MACSIO_LOG_LogFinalize(MACSIO_LOG_MainLog);
74 }
75 #ifdef HAVE_MPI
76     MPI_Finalize();
77 #endif
78 
79 return 0;
80 }
```

### 3.8 Contributing to MACSio Documentation

- Use :any: for links into Doxygen content instead of :c:func: as described in breathe documentation. The reason is that in order for breathe's :c:func: to work, one has to create breathe-specific sphinx directives for each and every Doxygen target (which IMHO kinda defeats the purpose of breathe). Also, it apparently only works for functions anyways, not types or macros or structs.

- Put Doxygen markup only in header files. Avoid using it in implementation (c) files.

- Keep Doxygen markup fairly terse and to the point and put more of the prose and lengthy descriptions in .rst files.

- Use .. only:: internals to conditionally include blocks that are used solely to document internal.
functionality of a given package. See example in `macsio_mif.rst`.

- To build locally, use: `env DONT_INSTALL_BREATHE=1 READTHEDOCS=1 sphinx-build -b html . _build -a [-t internals]`
  - genindex
  - modindex
  - search
  - genindex
  - modindex
  - search
Symbols

_MACSIO_CLARGS_ArgvFlags_t (C++ class), 48
_MACSIO_LOG_LogHandle_t (C++ class), 62
_MACSIO_MIF_ioFlags_t (C++ class), 43, 46
_BASEFILE_ (C macro), 51, 59
_log_flags_t (C++ class), 62
_user_data (C++ class), 40

C
CloseHDF5File (C++ function), 38
CloseMyFile (C++ function), 27
comm (C++ member), 62
compression_alg_str (C++ member), 40
compression_params_str (C++ member), 40
CreateHDF5File (C++ function), 38
CreateMyFile (C++ function), 26
current_line (C++ member), 62

d
defaults_mode (C++ member), 49
do_wr (C++ member), 43, 46
dspc (C++ member), 40
dummy (C++ member), 28, 40
dumpFunc (C++ member), 25
DumpFunc (C++ type), 23

E
error_mode (C++ member), 49
ext (C++ member), 25
extra_lines_proc0 (C++ member), 62

F
fid (C++ member), 40
filename (C++ member), 40
flags (C++ member), 62

G
get_tokval (C++ function), 37
groupId (C++ member), 40

H
H5Pset_zfp_accuracy_cdata (C macro), 37
H5Pset_zfp_expert_cdata (C macro), 37
H5Pset_zfp_precision_cdata (C macro), 37
H5Pset_zfp_rate_cdata (C macro), 37

I
identifyFileFunc (C++ member), 25, 26
IdentifyFileFunc (C++ type), 23
iface_ext (C++ member), 28, 39
iface_name (C++ member), 28, 39

J
json_as_html (C++ member), 28

L
lbuf_size (C++ member), 40
lines_per Proc (C++ member), 62
loadFunc (C++ member), 25
LoadFunc (C++ type), 23
log_flags_t (C++ type), 59
log_line_length (C++ member), 62
logfile (C++ member), 62

M
MACSIO_CLARGS_ARG_GROUP_BEG (C macro), 48
MACSIO_CLARGS_ARG_GROUP_BEG2 (C macro), 48
MACSIO_CLARGS_ARG_GROUP_END (C macro), 48
MACSIO_CLARGS_ARG_GROUP_END2 (C macro), 48
MACSIO_CLARGS_ArgvFlags_t (C++ type), 48
MACSIO_CLARGS_ASSIGN_OFF (C macro), 47
MACSIO_CLARGS_ASSIGN_ON (C macro), 47
MACSIO_CLARGS_END_OF_ARGS (C macro), 48
MACSIO_CLARGS_ERROR (C macro), 47
MACSIO_CLARGS_GRP_BEG (C macro), 48
MACSIO_CLARGS_GRP_END (C macro), 48
MACSIO_CLARGS_GRP_SEP_STR (C macro), 48
MACSIO_CLARGS_HELP (C macro), 47
MACSIO_CLARGS_NODEFAULT (C macro), 48