Description of the SDF File Format for Version 1.1

Keith Bennett

September 4, 2024

1 The SDF File Format

The CFSA group at the University of Warwick actively maintains several complex codebases and has plans to expand the variety of codes in the future. To ease maintenance and facilitate sharing of data between codes it is beneficial for all of these codes to output results to the same file format. Such a format needs to be "self-describing", which means that each variable dumped to the file is accompanied by a description of the variable and enough information for a visualisation tool to know how to display its contents in a meaningful way.

There already exists a wide variety of output formats but all have been found to have some shortcomings. In particular, none of the popular file formats benchmarked were found to be as fast for reading and writing to disk as a native MPI-IO implementation. For this reason, we have chosen to design our own format which is as simple as possible whilst providing the basic features we require. An additional benefit to this approach is that it reduces the number of dependencies required to make use of the format.

We have chosen to call this format "SDF" which stands for "Self Describing Format". Files written to this format will have the suffix ".sdf".

This document should fully describe the contents of an SDF version 1.1 file. From this it should be possible to write a compliant library for reading and writing SDF files.

2 File Structure

The layout of an SDF file is as follows:

File header
- Block header #1
  - Block metadata #1
    - Block data #1
- Block header #2
  - Block metadata #2
    - Block data #2
    - ⋮
- Block header #1
  - Block metadata #1
- Block header #2
  - Block metadata #2
    - ⋮

The file header contains basic information about the simulation, number of blocks contained in the file and the location of the first block header. Each block header knows the location of the following block header.

In the outline shown above it can be seen that the block headers and metadata are repeated at the end of the file. An older version of the format just wrote the header and metadata for each block followed by

the data for that block and then moved on to the next block. This mode of operation is still supported but it has its shortcomings. It is often the case that the only data required from a file is a list of its contents. A good example of this is a visualisation tool such as VisIt. The first thing that it does is to scan the contents of a file so that it can build a list of menu entries. For this purpose it is much better to have all of the header and metadata in a contiguous block without having to skip the data block each time. This can be achieved by placing an extra copy of the metadata at the end of the file.

3 File Header

Every SDF file starts with a header containing basic global information about the file and its contents. The data written to the file header in version 1.1 of the format is as follows:

Offset (bytes)	Datatype & size	Variable name	Description
0	char 4	sdf	Always "SDF1" in a valid file
4	int 4	endianness	An integer used to indicate the endianness of the file. This is a fixed constant which should never need to be changed. It should written as 0x0f0e0201 (16911887) on little-endian machines and 0x01020e0f on big-endian machines. An opposite-endian machine will read it as being 252576257.
8	int 4	sdf_version	Version number. If this is higher than the version of the reader then the reader MUST fail as the file will almost certainly not be readable. If the version of the file is higher than version 1 then the items listed beyond "sdf_revision" may not correspond to the contents of the file header at all.
12	int 4	sdf_revision	Revision number. If this is higher than the version of the reader then it should try to continue. Revision changes only add data to structures, never remove or move data. If the revision of the file is higher than revision 1 then there may be more items than are listed here but these should not affect the ability to properly read the file.
16	char 32	code_name	The name of the program which generated the file. eg. "Epoch2d"
48	int 8	first_block_location	The location of the first block header relative to the start of the file. If a reader wishes to read the file by reading metadata followed by data on a block by block basis, then it should seek to this position to locate the first block header.

Offset (bytes)	Datatype & size	Variable name	Description
56	int 8	summary_location	The location of the first block header in the summary section relative to the start of the file. If a reader wishes to scan the contents of a file before reading any data then it should seek to this position to locate the first block header.
64	int 4	summary_size	The total size of all the metadata written to the summary.
68	int 4	nblocks	The number of blocks in the file. When writing this should be zero until the file is written and closed.
72	int 4	block_header_length	The length in bytes of the block headers. This is a fixed length which excludes the individual block's metadata.
76	int 4	step	The simulation step number.
80	real 8	time	The simulation time.
88	int 4	jobid1	This integer and the one which follows uniquely identify the simulation which wrote the data. All output files created by a simulation will have the same jobid flags. If a job is restarted then the jobid will be retained. Currently the time in seconds and milliseconds is used but this may change.
92	int 4	jobid2	See above
96	int 4	string_length	All long strings in the file are of this length. Allocate this much space for all future string variables. See below for further details.
100	int 4	code_io_version	A revision number for the program's output. The code I/O version is written in case the semantics of the output change in such a way that the code cannot interpret the data based on the self-describing semantics alone. It should rarely be required.
104	char 1	restart_flag	A single byte that is equal to one if this file can be used as a restart dump and zero otherwise.
105	char 1	subdomain_file	A single byte that is equal to one if this file represents a single subdomain and zero if it contains data for the entire simulation domain. Although not currently supported, a future version of the I/O library will be able to write one output file per process. This flag will then be used to dermine if the file was generated in such a manner.
106			

There may be padding at the end of the file header.
There are two fixed character lengths used in the dump. They are fixed to ease support for FORTRAN

which has fairly weak string handling support. Short strings are written using CHARACTER*32, chosen because 31 is the maximum identifier length in standard FORTRAN. The second length is "String length", which can be chosen at run time but remains fixed within a single dump file. It defaults to 64. For the sake of brevity, this length will be referred to as "s" throughout the remainder of this document.

4 Block Headers

All block headers are exactly the same size and contain the same information. They contain information which is required by all blocks to identify their contents, such as an identifier, data location, etc. They also contain a blocktype which is used to determine the size and contents of the metadata section since it is different for each block type. If there are any entries of variable size in the metadata section then the information required to allocate memory must be contained in the block header.

Offset (bytes)	Datatype & size	Variable name	Description
0	int 8	next_block_location	The location of the next block header relative to the start of the file.
8	int 8	data_location	The location of the data for the current block. Note that the data can be placed anywhere in the file and is not at a fixed location relative to the metadata.
16	char 32	block_id	The block identifier. This is a short string which uniquely identifies the block within the file.
48	int 8	data_length	The size of the data section in the current block. This can be used together with "datatype" to determine the total number of elements if the variable is an array.
56	int 4	blocktype	The type of the block. This determines the contents of the metadata section. If a reader does not have support for a given blocktype then it can just skip to the next block using next_block_location.
60	int 4	datatype	The basic datatype of the variable. This is specified using a named constant, described later.
64	int 4	ndims	The number of dimensions of the variable if it is an array. If the variable is not an array then this is equal to 1.
68	char s	block_name	The display name used for the variable. This is the name presented to a user in a visualisation tool so it must be human readable and descriptive of the variable.
68+s	int 4	block_info_length	The length of the block-specific header information. Introduced in version 1.1.
72+s			

Here, "s" is the size of "string_length" as specified in the file header. For the sake of brevity, the value of "ndims" will be referred to as "n" throughout the remainder of this document.

The block-id uniquely identifies a block with a machine readable identifier. Typically, this is just the variable name (in lower case). When data is presented using VisIt and other tools, the "name" field is used. This allows the presentation of data names to change without breaking code which relies on a fixed name. Previous versions of the format also used a separate "class" name. Such groupings can easily be accomplished by separating components using a "/" symbol. Since this convention allows greater flexibility, the class name has been dropped.

For multi-dimensional arrays, the value of "ndims" will be greater than one and the size of the array in each dimension will be determined by a "dims" array in the block's metadata section. It should be noted that ALL multi-dimensional arrays are written using column-major order. This is the ordering used by Fortran, Matlab and VisIt. The C programming language uses row-major order but since it doesn't really have proper support for multi-dimensional arrays it doesn't get a vote.

The possible values for "blocktype" are itemised below. They are defined as constants in the SDF Fortran module, as indicated by the "c_" prefix.

Constant name	Value	Description
c_blocktype_scrubbed	-1	Deleted block. Code should ignore.
c_blocktype_null	0	Unknown block type. This is an error.
c_blocktype_plain_mesh	1	Block describing a plain mesh or grid.
c_blocktype_point_mesh	2	Block describing a point mesh or grid.
c_blocktype_plain_variable	3	Block describing a variable on a plain mesh.
c_blocktype_point_variable	4	Block describing a variable on a point mesh.
c_blocktype_constant	5	A simple constant not associated with a grid.
c_blocktype_array	6	A simple array not associated with a grid.
c_blocktype_run_info	7	Information about the simulation.
c_blocktype_source	8	Embedded source code block.
c_blocktype_stitched_tensor	9	List of blocks to combine as a tensor or vector.
c_blocktype_stitched_material	10	List of blocks to combine as a multi-material mesh.
c_blocktype_stitched_matvar	11	List of blocks to combine as a multi-material variable.
c_blocktype_stitched_species	12	List of blocks to combine as a species mesh. This is similar to a multi-material mesh except there is no interface in a mixed cell.
c_blocktype_species	13	Information about a particle species.
c_blocktype_plain_derived	14	This blocktype is never actually written to an SDF file. It is used within the C-library and VisIt to represent a plain variable whose content is generated dynamically based on other data in the file.
c_blocktype_point_derived	15	As above, this blocktype is never actually written to an SDF file. It serves the same purpose as c_blocktype_plain_derived, except the variable is defined on a point mesh.
c_blocktype_multi_tensor	16	This is the same as c_blocktype_stitched_tensor, except that all the data for the stitched variables is contained in the data section of this block rather than the blocks which are referenced.
c_blocktype_multi_material	17	Same as above, for c_blocktype_stitched_material
c_blocktype_multi_matvar	18	Same as above, for c_blocktype_stitched_matvar
c_blocktype_multi_species	19	Same as above, for c_blocktype_stitched_species

The possible values for “datatype” are itemised below.

Constant name	Value	Description
c_datatype_null	0	No datatype specified. This is an error.
c_datatype_integer4	1	4-byte integers.
c_datatype_integer8	2	8-byte integers.
c_datatype_real4	3	4-byte floating point (ie. single precision).
c_datatype_real8	4	8-byte floating point (ie. double precision).
c_datatype_real16	5	16-byte floating point (ie. quad precision).
c_datatype_character	6	1-byte characters.
c_datatype_logical	7	Logical variables. (Represented as 1-byte characters).
c_datatype_other	8	Unspecified datatype. The type of data in the block must be inferred from the block type.

5 Block Types

In this section we describe the metadata and data contents for each of the possible block types. Note that as the format evolves more block types may be added, but that should not affect the ability of old readers to read only the block types which they know about. Since the metadata follows on from the block header, the offset given starts "m" bytes on from the start of the block. The value of "m" is that given by "block_header_length" in the file header. Note that the metadata may not necessarily continue directly after the last entry read from the block header, since the block header size may change in future revisions. For this reason you should always seek to a position "m" bytes after the start of the block before reading the block metadata.

5.1 c_blocktype_plain_mesh and c_blocktype_point_mesh

Since grid based meshes and point meshes are quite similar, we will describe them both here. A mesh defines the locations at which variables are defined. Since the geometry of a problem is fixed and most variables will be defined at positions relative to a fixed grid, it makes sense to write this position data once in its own block. Each variable will then refer to one of these mesh blocks to provide their location data.

Both grid based meshes and point meshes write the same basic information at the start of the header and this common metadata is given below.

Offset (bytes)	Datatype & size	Variable name	Description
m	real 8 * n	mults	The normalisation factor applied to the grid data in each direction.
m+8n	char 32 * n	labels	The axis labels for this grid in each direction.
m+40n	char 32 * n	units	The units for this grid in each direction after the normalisation factors have been applied.
m+72n	int 4	geometry_type	The geometry of the block.
m+72n+4	real 8 * n	minval	The minimum coordinate values in each direction.
m+80n+4	real 8 * n	maxval	The maximum coordinate values in each direction.
m+88n+4			

Here, "m" is the size of "block_header_length" as specified in the file header and "n" is the number of dimensions as specified in the block header.

The "geometry_type" specifies the geometry of the current block and it can take one of the following values:

Constant name	Value	Description
c_geometry_null	0	Unspecified geometry. This is an error.
c_geometry_cartesian	1	Cartesian geometry.
c_geometry_cylindrical	2	Cylindrical geometry.
c_geometry_spherical	3	Spherical geometry.

At the end of the metadata, grid based meshes and point meshes contain differing information. For a grid based mesh ("c_blocktype_plain_mesh") the last items in the header are as follows:

Offset (bytes)	Datatype & size	Variable name	Description
m+88n+4	int 4n	dims	The number of grid points in each dimension.
m+92n+4			

Here, "m" is the size of "block_header_length" as specified in the file header and "n" is the number of dimensions as specified in the block header.

The data is then written at the location specified by "data_location" in the block header. Note that this will not always be in the position immediately following the block header so a reader must always seek to this location explicitly.

For a grid based mesh, the data written is the locations of node points for the mesh in each of the simulation dimensions. Therefore for a 3d simulation of resolution (nx, ny, nz), the data will consist of a 1d array of X positions with $(nx + 1)$ elements followed by a 1d array of Y positions with $(ny + 1)$ elements and finally a 1d array of Z positions with $(nz + 1)$ elements. Here the resolution specifies the number of simulation cells and therefore the nodal values have one extra element. In a 1d or 2d simulation, you would write only the X or X and Y arrays respectively.

For a point mesh ("c_blocktype_point_mesh") the last items in the header are as follows:

Offset (bytes)	Datatype & size	Variable name	Description
m+88n+4	int 8	np	Number of points
m+88n+12			

Here, "m" is the size of "block_header_length" as specified in the file header and "n" is the number of dimensions as specified in the block header.

The data written is the locations of each point in the first direction followed by the locations in the second direction and so on. Thus, for a 3d simulation, if we define the first point as having coordinates (x_1, y_1, x_1) and the second point as (x_2, y_2, z_2), etc. then the data written to file is a 1d array with elements $(x_1, x_2, ..., x_{np})$, followed by the array $(y_1, y_2, ..., y_{np})$ and finally the array $(z_1, z_2, ..., z_{np})$ where "np" corresponds to the number of points in the mesh. For a 1d simulation, only the x array is written and for a 2d simulation only the x and y arrays are written.

The alternative method (which is NOT used) is to write the location data one point at a time so that you have a single array in the form $(x_1, y_1, z_1, x_2, y_2, z_2, ..., x_{np}, y_{np}, z_{np})$. For the time being it is thought that the first method will be the most convenient. At a later time, we may switch to using the second method if it proves more useful. This can easily be done by defining a new blocktype and incrementing the revision number.

5.2 c_blocktype_plain_variable and c_blocktype_point_variable

As with the mesh data described previously, the grid based variables and point variables are quite similar so we will describe them both here. Both blocktypes describe a variable which is located relative to the points given in a mesh block.

Both grid based variables and point variables write the same basic information at the start of the header and this common metadata is given below.

Offset (bytes)	Datatype & size	Variable name	Description
m	real 8	mult	The normalisation factor applied to the variable data.
8+m	char 32	units	The units for this variable after the normalisation factor has been applied.
40+m	char 32	mesh_id	The "id" of the mesh relative to which this block's data is defined.
72+m			

Here, "m" is the size of "block_header_length" specified in the file header.

At the end of the metadata, grid based variables and point variables contain differing information. For a grid based variable ("c_blocktype_plain_variable") the last items in the header are as follows:

Offset (bytes)	Datatype & size	Variable name	Description
72+m	int 4n	dims	The number of grid points in each dimension.
72+m+4n	int 4	stagger	The location of the variable relative to its associated mesh.
76+m+4n			

Here, "m" is the size of "block_header_length" specified in the file header and "n" is the number of dimensions as specified in the block header.

The mesh associated with a variable is always node-centred, ie. the values written as mesh data specify the nodal values of a grid. Variables may be defined at points which are offset from this grid due to grid staggering in the code. The "stagger" entry specifies where the variable is defined relative to the mesh. Since we have already defined the number of points that the associated mesh contains, this determines how many points are required to display the variable. The entry is represented by a bit-mask where each bit corresponds to a shift in coordinates by half a grid cell in the direction given by the bit position. Therefore the value "1" (or "0001" in binary) is a shift by $dx/2$ in the x direction, "2" (or "0010" in binary) is a shift by $dy/2$ in the y direction and "4" (or "0100" in binary) is a shift by $dz/2$ in the z direction. These can be combined to give shifts in more than one direction. The system can also be extended to account for more than three directions (eg. "8" for direction 4).

For convenience, a list of pre-defined constants are defined for the typical cases.

The "stagger" entry can take one of the following values:

Constant name	Value	Description
c_stagger_cell_centre	0	Cell centred. At the midpoint between nodes. Implies an (nx, ny, nz) grid.
c_stagger_face_x	1	Face centred in X. Located at the midpoint between nodes on the Y-Z plane. Implies an $(nx + 1, ny, nz)$ grid.
c_stagger_face_y	2	Face centred in Y. Located at the midpoint between nodes on the X-Z plane. Implies an $(nx, ny + 1, nz)$ grid.
c_stagger_face_z	4	Face centred in Z. Located at the midpoint between nodes on the X-Y plane. Implies an $(nx, ny, nz + 1)$ grid.
c_stagger_edge_x	6	Edge centred along X. Located at the midpoint between nodes along the X-axis. Implies an $(nx, ny + 1, nz + 1)$ grid.
c_stagger_edge_y	5	Edge centred along Y. Located at the midpoint between nodes along the Y-axis. Implies an $(nx + 1, ny, nz + 1)$ grid.
c_stagger_edge_z	3	Edge centred along Z. Located at the midpoint between nodes along the Z-axis. Implies an $(nx + 1, ny + 1, nz)$ grid.
c_stagger_vertex	7	Node centred. At the same place as the mesh. Implies an $(nx + 1, ny + 1, nz + 1)$ grid.

The data is then written at the location specified by "data_location" in the block header. Note that this will not always be in the position immediately following the block header so a reader must always seek to this location explicitly.

For a grid based variable, the data written contains the values of the given variable at each point on the mesh. This is in the form of a 1d, 2d or 3d array depending on the dimensions of the simulation. The size of the array depends on the size of the associated mesh and the grid staggering as indicated above. It corresponds to the values written into the "dims" array written for this block.

For a point variable ("c_blocktype_point_variable") the last items in the header are as follows:

Offset (bytes)	Datatype & size	Variable name	Description
72+m	int 8	np	Number of points
80+m			

Here, "m" is the size of "block_header_length" specified in the file header.

Similarly to the grid based variable, the data written contains the values of the given variable at each point on the mesh. Since each the location of each point in space is known fully, there is no need for a stagger variable. The data is in the form of a 1d array with "np" elements.

5.3 c_blocktype_constant

This block is used for writing a simple constant to the file. Since the data written is small, there is no need for a separate data block and the value is just written into the metadata section.

Offset (bytes)	Datatype & size	Variable name	Description
m	type t	data	The actual constant to be written.
m+t			

Here, "m" is the size of "block_header_length" as specified in the file header and "t" is the size of the datatype specified in the block header.

5.4 c_blocktype_array

This block is used for writing a simple array to the file. This array is not associated with any mesh and therefore cannot be plotted on a spatial grid.

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4n	dims	The dimensions of the array to be written.
m+4n	type t*e	data	The actual array data.
m+4n+t*e			

Here, "m" is the size of "block_header_length" as specified in the file header. "n" is the number of dimensions, "t" is the size of the datatype and "e" is the number of elements all specified in the block header.

5.5 c_blocktype_run_info

This block contains information about the code which was used to generate the SDF file and further details relating to the simulation, such as hardware on which it was run, etc.

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4	code_version	The version of the code.
4+m	int 4	code_revision	The revision of the code.
8+m	char s	commit_id	The revision control commit ID for the code.
8+m+s	char s	sha1sum	The SHA-1 checksum of the source code.
8+m+2s	char s	compile_machine	The machine name on which the code was compiled.
8+m+3s	char s	compile_flags	The compilation flags used when compiling the code.
8+m+4s	int 8	defines	A bitmask of the pre-processor define flags used. This is a 64-bit integer, so only 63 flags can be used in total.
16+m+4s	int 4	compile_date	The time at which the code was compiled, written as seconds since the UNIX epoch.
20+m+4s	int 4	run_date	The time at which the simulation first began running, written as seconds since the UNIX epoch.
24+m+4s	int 4	io_date	The time at which this output file began writing, written as seconds since the UNIX epoch.
28+m+4s			

Here, "m" is the size of "block_header_length" and"s" is the size of "string_length" both specified in the file header.

5.6 c_blocktype_source

This block contains the source code which was used to generate the binary which generated this SDF file. It is written as one long stream of bytes. In the current implementation, the byte stream consists of a gzipped tar file which is uuencoded to form one long character string. No further metadata is required to describe this output so the contents begin at "data_location". This data is a 1d character array of size given by "data_size" in the block header.

5.7 c_blocktype_stitched_tensor, _material, _matvar and _species

Variables in SDF files are written so that they correspond to spatial meshes wherever possible. This means that variables which have multiple components, such as tensors, are written with a separate block for each component. This data is then "stitched" back together and a separate block is used to instruct a reader as to which blocks need to be combined and how to combine them. This improves flexibility since a block can be used in multiple different ways. For example, you can have multi-material vectors and so on.

The "stitched" blocks just contain a list of all the blocks which need to be combined and the number of dimensions in the resulting variable. All of this data is small, so it is written as metadata and there is no data block for any of the "stitched" blocks.

A "c_blocktype_stitched_tensor" block defines a vector or tensor field. The blocks which are itemised to be combined into a tensor may be either point data or regular grid based data but the type used must be consistent with the mesh on which the resulting tensor is to be defined. For a

"c_blocktype_stitched_tensor" block, the metadata to be written is as follows:

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4	stagger	The location of the tensor variable relative to its associated mesh.
4+m	char 32	mesh_id	The "id" of the mesh on which the resulting tensor is defined.
36+m	char 32n	variable_ids	The "id" of each component in the tensor.
36+m+32n			

For this and the blocks which follow, "m" is the size of "block_header_length" specified in the file header and "n" is the number of dimensions as specified in the block header.

Note that there is no "_vector" blocktype since a vector is just a tensor with two or three components.

"c_blocktype_stitched_material" defines a material mesh. This is a set of volume fraction arrays, one for each material, which are combined to reconstruct the interfaces between materials for a multi-material simulation. The number of materials to combine is determined by the "ndims" entry in the block header. The metadata for this blocktype are as follows.

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4	stagger	The location of the material relative to its associated mesh.
4+m	char 32	mesh_id	The "id" of the mesh on which the resulting material mesh is defined.
36+m	s * n	material_names	The name of each material component.
36+s n+m	char 32n	vfm_ids	The "id" of each volume fraction component.
36+(32+s)n+m			

"c_blocktype_stitched_matvar" defines a set of material variables to combine into a single variable. This aggregate total is calculated by summing the values of each material variable multiplied by its associated volume fraction block. The variables to combine may be listed in any order. They are matched with their associated volume fraction by comparing the component of the "name" field following the final "/" character. A variable which is only defined on a subset of materials need only specify the materials for which it is defined.

The number of materials to combine is determined by the "ndims" entry in the block header. The metadata for this blocktype are as follows.

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4	stagger	The location of the material variable relative to its associated mesh.
4+m	char 32	mesh_id	The "id" of the mesh on which the resulting material variable is defined.
36+m	char 32	material_id	The "id" of the stitched material block for this material variable.
68+m	char 32n	variable_ids	The "id" of each material component.
68+32n+m			

A species mesh is similar to a material mesh except that the volume fraction arrays are used to determine the percentage of a particular species contained within a cell rather than define an interface. A species mesh is associated with a specific material from a multi-material mesh.

Offset (bytes)	Datatype & size	Variable name	Description
m	int 4	stagger	The location of the species variable relative to its associated mesh.
4+m	char 32	mesh_id	The "id" of the mesh on which the resulting species variable is defined.
36+m	char 32	material_id	The "id" of the stitched material block for this species variable.
68+m	s	material_name	The name of the material component on which this species is defined.
68+s+m	s * n	species_names	The names of each species component.
68+(n+1)s+m	char 32n	variable_ids	The "id" of each species component.
68+32n+(n+1)s+m			

5.8 c_blocktype_particle_species

This blocktype has not yet been finalised.

6 Ideal sequence for reading SDF files

This section details the sequence that an ideal SDF reader should follow to parse a SDF file.

- Open the file.
- Read the "SDF1" string to confirm that this is a SDF file.
- If the "SDF1" string isn't found then exit.
- Read the rest of the header info.

- Check that the endianness flag has the value c_endianness (16911887). If not, then this file was written on a different machine architecture. Either convert all the values read in from this point on or exit with an error message.
- Check that the version is less than or equal to the version of the standard that the reader is written to.
- If not then exit with an error message about getting a new reader.
- Check that the revision is less than or equal to the revision of the standard that the reader is written to.
- If not then print a warning message, but continue.
- If the "nblocks" field is zero then exit since the file hasn't been finished yet.
- If you wish to read the metadata followed by data for each block in turn, set the file pointer to "first_block_location". If instead you want to scan the contents of the file before reading any data, set the file pointer to "summary_info_location".
- Loop over "nblocks"
- Skip this step for the first block. For all subsequent blocks, set the file pointer to "next_block_location" given in the block header for the previous block.
- Store the current file pointer value as "block_start".
- Read the block header
- Determine whether the reader can deal with the type of block which is described by the header. Also check for user request for this variable etc.
- If not then the filepointer should be set to the value of "next_block_location" given in the block header and then cycle back to the start of the loop.
- If you only require block information stored in the block header then it is valid to skip the remaining steps and return to the start of the loop.
- Otherwise set the filepointer to be at "block_start + block_header_size".
- Read the block metadata for the correct type of block.
- If you only require block metadata then it is valid to skip the remaining steps and return to the start of the loop VisIt plugin works in this way when populating the VisIt metadata server).
- Move the filepointer to "data_location".
- Read the block data and return to the start of the loop.
- After "nblocks" have been iterated, close the file.

Since there is no requirement on the order in which blocks are written into the file, it is not possible to do sanity checks until the file has been parsed once, so a good reader should then do the following checks

- Check that all the meshes which are requested by mesh variables exist. If one of the meshes doesn't exist print a warning message and if the code cannot cope without a mesh then drop the variable. The code should NOT fail at this point if it is possible to continue.
- Check that the sizes of variables match the sizes of the associated meshes. If not then print a warning message and drop the variable. The code should NOT fail at this point if it's possible to continue.