EM Field Software Specifications

Contents

Source code of this software is in Github at

https://github.com/DavidGeUSA/TSS/tree/master/Source%20Code

The purpose of this software is for doing digital experiments on electromagnetic fields, such as simulations, statistics, FFT, 3D/2D drawings, etc. It is not intended to be a software product; rather, it uses open source to provide a framework and tools for you to experiment your own ideas and coding. Below this set of software will be referred to as "Electromagnetic Field Analyzer", or **EMFA**.

Software Structure

EMFA consists of several relatively independent modules as outlined below listed by source code folder names.

Executable Applications

- **EMconsole**. This is a console application. It is coded in C++. It is an entry point for running tasks. It loads modules from dynamic link libraries to perform various tasks in different ways. You can do experiments by implementing your ideas in dynamic link libraries and run supported tasks; you can also define new tasks. It uses command line parameters to specify folders required by tasks. It uses a text file, task file, to specify configuration values needed by all modules and tasks. In your own modules you may define and use new configuration values.
- **OpenGL3D**. This is a 3D drawing application. It is coded in C#. It uses Open GL library to draw 3D fields. It automatically adjusts zooming factors for space locations and vector sizes to produce good 3D field views. You can use mouse to rotate 3D views to examine EM fields from different angles. You can visually see EM fields evolve over time.
- **Draw2D**. This is a 2D drawing application. It is coded in C#. It can be used to compare simulation errors of different algorithms.

Dynamic Link Libraries

The purpose to use dynamic link libraries is to make it easy to do experiments on different ideas and coding. It also reduces external library dependency of EMFA. All dynamic link libraries are coded in C++.

Currently following modules can be implemented in dynamic link libraries and used by tasks.

- **FDTD** module – it implements an FDTD algorithm
- **Initial Value** module – it provides EM field values for any given space points
- **Field Source** module – it implements an EM field source
- **Boundary Condition** module – it implements a boundary condition
- **TF/SF** module – it implements a total field/scattered field boundary

By specifying different DLL files, class names and providing different parameters for each module, the console application can carry out tasks in different ways.

Sample projects show how to implement these modules.

Static Link Libraries

Static link libraries provide utilities for developing modules.

Math Tools

It provides math related functions.

File Utilities and TaskFile

It provides file related utilities, including a TaskFile class for reading task parameters from a task file.

TaskFile class

An instance of TaskFile is passed into an initialization function of a plugin module.

The TaskFile class provides following public functions:

```
double getDouble(const char *name, bool optional);
int getInt(const char *name, bool optional);
unsigned int getUInt(const char *name, bool optional);
long int getLong(const char *name, bool optional);
char *getString(const char *name, bool optional);
bool getBoolean(const char *name, bool optional);

int getErrorCode(){return ret;}
void resetErrorCode(void);

void setNameOfInvalidValue(char *name);
```

A task file contains lines of strings
Each line is formed by "name"="value"
"name" is a string. All names in a task file are unique.
"value" is a value of a primary type, i.e. double, integer, unsigned integer, long integer, Boolean, and string.
A line can be a comment if it starts with "//"

Each module, i.e. simulation console, boundary condition, TFSF, FDTD, field source, and initial condition, defines names and expected values

A task file defines a task by the values assigned to names

This class provides functions, get???(name, optional), for searching and parsing values in a task file, where "name" identifies a task parameter.

If a given "name" does not exist in a task file then a function get???(name, optional) returns a default value; the default value for a string is NULL; the default value for a number is 0; the default value for a Boolean is false.

function getErrorCode returns ERR_OK if "optional" is true;
function getErrorCode returns ERR_TASK_PARAMETER_NAME if "optional" is false, and subsequent calling of get???(name, optional) will do nothing.

The designed usage of this class is that when get???(name, false) is called (that is, optional is false) then getErrorCode should be called to check whether the name exists; if the name does not exist then the program should bail out and return the error code. But you do not have to call getErrorCode after each call of get???(name, optional); you may make calls of get???(name, optional) for all the task parameters and then make one call of getErrorCode to do checking.

If you want to know whether a "name" exists but also want to treat the task parameter as optional then use false for the "optional" when calling get???(name, optional), and call getErrorCode to examine whether the "name" exist, then call resetErrorCode before calling get???(name, optional) for other task parameters.

Your code needs to verify that value of a task parameter is in your expected data range. If it is not within the expected range then your code should call setNameOfInvalidValue(char *name), passing the task parameter name, and then bail out. The console application will display an error message and the task parameter name.

Future modifications of this class should keep the above code logic so that the modifications do not break existing code using this class.

Memory Manager

EM fields may need large amount of memories. The memory sizes may be much larger than the standard C/C++ memory allocation functions in a specific operating system can handle.
This static library uses file mapping to allocate large memories. The memory size is only limited by drive size.

A C++ class, MemoryManager, is acting as a memory management utility. For a class, i.e., a plugin module, to use the memory manager, the class may add a base class of MemoryManUser. The console application should pass an instance of MemoryManager to MemoryManUser. Following macros can be used within the class for allocating memories.

#define **MEMMANEXIST** (_mem==NULL)?ERR_MEM_MAN_NULL:ERR_OK

Before using other memory management functions, use this macro to test whether an instance of the MemoryManager has being assigned by the console application.

#define **AllocateMemory**(size) _mem->Allocate(size)

It allocates a block of memory and returns a pointer to the memory. “size” is a size_t integer, it is a desired memory size. A temporary file is created in a folder specified by “/W” command line parameter. The file is mapped to the memory. The memory size limit is the capacity of the drive where the folder is in. Therefor the memory size can be very large. The file will be deleted on freeing the memory.

#define **FreeMemory**(e) _mem->Free((e))

Free a block of memory. “e” is the memory pointer. If “e” was allocated via AllocateMemory then the temporary file is deleted; otherwise the mapped file for the memory is not deleted.

#define **ReadFileIntoMemory**(fname, sizeRet, errRet) _mem->ReadFileIntoMemoryItem((fname), (sizeRet), (errRet))

Map an existing file into a read-only block of memory and return a pointer to the memory. Freeing the memory will not delete the file. “fname” is a full file path; “sizeRet” is a pointer to a size_t, it returns file size; “errRet” is a pointer to an integer, it returns an error code.

#define **CreateFileIntoMemory**(fname, size, errRet) _mem->CreateFileIntoMemoryItem((fname), (size), (errRet))

Create a new file and map it to a writable memory, return the pointer to the memory. Freeing the memory will not delete the file. “fname” is a full file path; “size” is a size_t, it is a desired memory size; “errRet” is a pointer to an integer, it returns an error code.

Software Configurations

Configuration values are needed to run a task in a desired way.

Those configurations depending on a computer environment are provided via command line parameters to the console application. Those parameters not depending on a computer environment are provided via a task file.

Command line parameters

Following are supported command line parameters.

/T – specify a task file. Example: /T”c:\tasks\task1.tsk”

/W – specify a folder for creating temporary files. For allocating large size memories, temporary files are used if memory contents are not to be kept on exiting the console application. Example: /W”d:\temp”

/L – specify a folder where Dynamic Link Library files for plug-in modules are located. Example: /D”c:\EM modules”. If it is missing then the console application folder is used.

/D – specify a folder for reading and writing data files. EM field data files are created in this folder.

/E – specify a second folder for reading and writing data files when needed.

Task File

A task file is a text file. Each line of the file defines one task parameter value in a format of “name”=”value”.

The “name” is a string identifying a parameter. The “name” and expected data type and data range of “value” are defined and used by a specific implementation of each plug-in module. There is a C++ class, **TaskFile**, for reading back “value” by “name”. An instance of TaskFile is passed to an initialization function of each plug-in module.

Each plug-in module is represented by a C++ class; the class must provide a parameter-less constructor. Configurations of the class must be done through task parameters in an initialization function.

When you are developing a plug-in module, you may want to define and use some configuration values. You need to choose a unique name for each configuration value. To ensure unique naming among all plug-in modules, it is a good idea to use a prefix in each name. Following prefixes can be used for each module.

- "SIM." – for the console application
- "FDTD." – for FDTD modules
- "FS." – for Field Source modules
- "IV." – for Initial Value modules
- "BC." – for Boundary Condition modules
- "TFSF." – for Total Filed/Scattered Field boundary modules

Common Task Parameters

Some parameters are commonly used by several tasks. For example, DLL file names and class names for plug-in modules. These parameters are defined here.

SIM.FDTD_DLL

It is a DLL file name, without path, for the FDTD module. For example,

SIM.FDTD_DLL=YeeFDTD.DLL

SIM.FDTD_NAME

It is a class name for the FDTD module. For example,

SIM.FDTD_NAME= YeeFDTDSpaceSynched

SIM.FS_DLL

It is a DLL file name, without path, for the Field Source module.

SIM.FS_NAME

It is a class name for the Field Source module.

SIM.BC_DLL

It is a DLL file name, without path, for the Boundary Condition module.

SIM.BC_NAME

It is a class name for the Boundary Condition module.

SIM.TFSF_DLL

It is a DLL file name, without path, for the TF/SF module.

SIM.TFSF_NAME

It is a class name for the TF/SF module.

SIM.IV_DLL

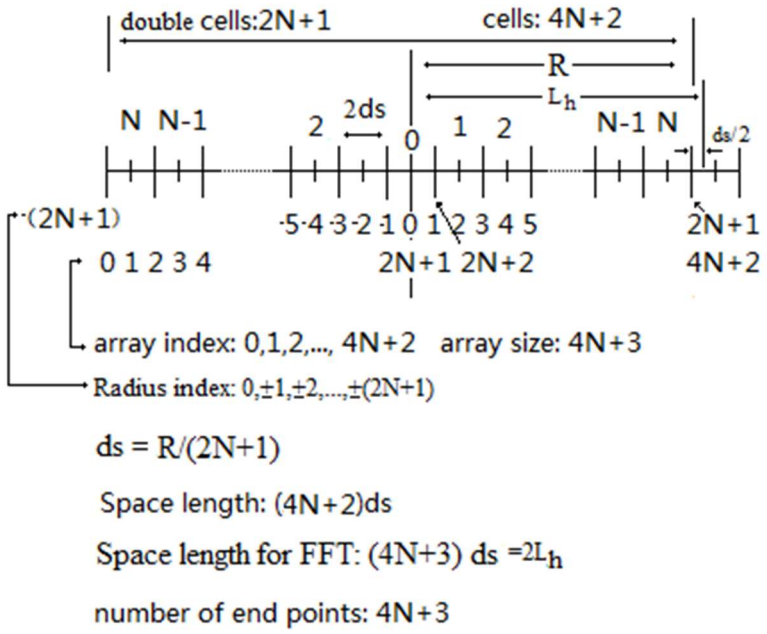
It is a DLL file name, without path, for the Initial Value module.

SIM.IV_NAME

It is a class name for the Initial Value module.

FDTD.N and FDTD.R

These two parameters, $N\ and\ R$, define space digitization as shown in the figure:



$$N: number\ of\ double\ space\ intervals\ in\ one\ side\ of\ an\ axis, excluding\ one\ interval\ at\ 0$$

$$R: space\ range\ in\ one\ side\ of\ an\ axis$$

There are $4N+3$ space points in one dimension. There are $(4N+3)^3$ space points in 3 dimensions.

$$space\ step\ size = \frac{R}{2N+1}$$

Example:

FDTD.N=100

FDTD.R=50

FDTD.HALF_ORDER_SPACE

It specifies half estimation order for space derivative estimation. Example:

FDTD.HALF_ORDER_SPACE=1

It specifies a second-order space derivative estimation.

FDTD.HALF_ORDER_TIME

It specifies half estimation order for time advancement estimation.

SIM.TASK

It is a task number which is an integer identifying a task to be executed by the console application. Example in a task file:

SIM.TASK=1

Supported task numbers are listed below. To add new tasks, add new task numbers to taskdef.h and add handling code in simConsole.cpp. Each task requires some command line parameters and task parameters. Note that plug-in modules may also need additional task parameters, which are not shown below; documentation for each plug-in module should list needed task parameters.

Task 0: show help

Task 1: verify that functions IndexToIndexes and IndexesToIndex work correctly; it requires a command line parameter “/W”; it requires a task parameter “FDTD.N”

Task 2: verify that function RadiusIndexToSeriesIndex works correctly; it requires a command line parameter “/W”; it requires a task parameter “FDTD.N”

Task 3: speed comparison between using function RadiusIndexToSeriesIndex and using a 3D array; it requires a command line parameter “/W”; it requires a task parameter “FDTD.N”

Task 4: verify a field Initial Value module works correctly. It requires command line parameter “/W”; “/L” is optional. It requires following task parameters: “FDTD.N”, “FDTD.R”, “SIM.IV_DLL” and “SIM.IV_NAME”

Task 5: verify an Initial Value module by divergences. It requires command line parameter “/W”; “/L” is optional. It requires following task parameters: “FDTD.N”, “FDTD.R”, “SIM.IV_DLL”, “SIM.IV_NAME” and “FDTD.HALF_ORDER_SPACE”.

Task 100: execute an EM field simulation. It requires command line parameters “/W” and “/D”; “/L” is optional. It requires following task parameters: “FDTD.N”, “FDTD.R” , “SIM.FDTD_DLL”, “SIM.FDTD_NAME”, “SIM.BC_DLL”, “SIM.BC_NAME”, “SIM.IV_DLL” and “SIM.IV_NAME”. Following task parameters are optional: “SIM.TFSF_DLL”, “SIM.TFSF_NAME”, “SIM.FS_DLL”, and “SIM.FS_NAME”. It also requires a task parameter “SIM.BASENAME” for specifying base file name, which does not include file name extension. Suppose “SIM.BASENAME” is specified as

SIM.BASENAME= simA

And command line uses “/Dc:\simulation\data” then for each simulation time step, the electromagnetic field is saved in a file “c:\simulation\data\simA{n}.em”, where {n} is time step which can be 0, 1, 2, …

Task 110: compare two simulations and generate a report file for each time step. It requires command line parameters “/W”, “/D” and “/E”. Use task parameters “SIM.FILE1” and “SIM.FILE2” to specify the base file names used by the two simulations; “/D” specifies folder for “SIM.FILE1” and “/E” specifies folder for “SIM.FILE2”. Use task parameter “SIM.THICKNESS” to specify boundary thickness to be excluded from the comparison.

Task 120: create a report file for each data file, It requires command line parameters "/W" and "/D". It also requires a task parameter "SIM.BASENAME" for specifying base file name, which does not include file name extension. It searches data files by {path by /D}\{base file name}{n}.em, where {n}=0,1,2,...; it uses an optional task parameter, "FDTD.HALF_ORDER_SPACE", to specify half estimation order for divergence estimations; default value is 1.

Task 130: pick field points with the largest strengths from each data file and generate a new data file. the new file contains items of 9 doubles: 3 for space location, 6 for EM field. It requires command line parameters "/W" and "/D". It requires following task parameters: "FDTD.R", "SIM.BASENAME", "SIM.POINTS" for the number of field points to pick, and "SIM.MAXTIMES" for maximum number of data files to process, use 0 to process all data files.

Task 140: merge two summary files into one file. a summary is generated by task 120. It requires command line parameters "/W", "/D" and "/E". Use task parameters "SIM.FILE1" and "SIM.FILE2" to specify the names of the summary files; "/D" specifies folder for "SIM.FILE1" and "/E" specifies folder for "SIM.FILE2". Use an optional task parameter "SIM.THICKNESS" to specify boundary thickness to be excluded from the merge; if it is missing then 0 is assumed.

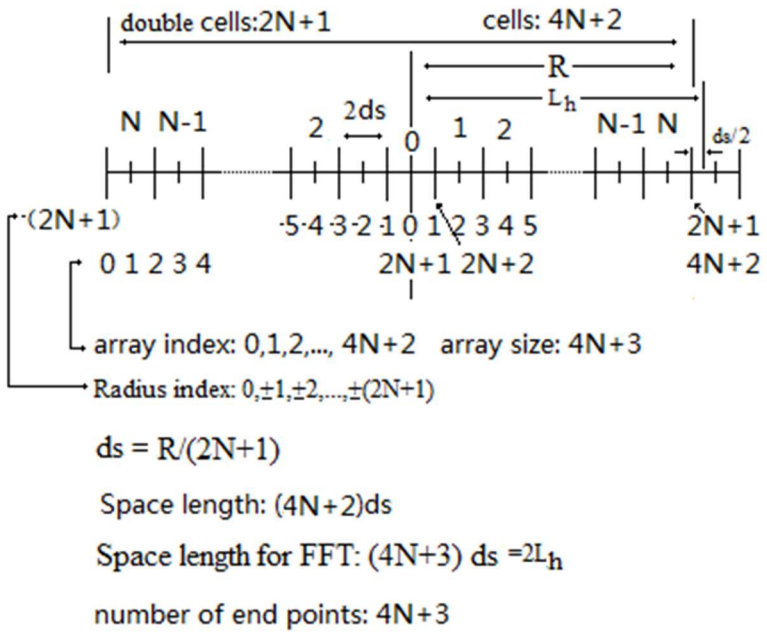
Task 160: merge two summary files into one file. a summary is generated by task 120. It requires command line parameters "/W", "/D" and "/E". Use task parameters "SIM.FILE1" and "SIM.FILE2" to specify the names of the summary files; "/D" specifies folder for "SIM.FILE1" and "/E" specifies folder for "SIM.FILE2". the space steps of the two simulations are ds1 and ds2, and ds1 = ds2/2, and maxRadius1 + 1 = 2 * maxRadius2

To add new tasks, modify taskdef.h to add new task definitions, and modify simConsole.cpp to handle new tasks.

Space Digitization

Space Point Indexing

Space digitization is represented by space range R and number of space points, as shown in the figure:



"Array indexing" is commonly used to represent 3D space points:

$$(i,j,k) represents\ (x_i, y_i, z_i) = \big((i-(2N+1))ds, (j-(2N+1))ds, (k-(2N+1))ds\big)$$

$$i,j,k = 0,1,2,\dots,(4N+2)$$

EMFA introduces a new approach of "radius indexing":

$$(m,n,p) represents\ (x_m, y_m, z_m) = (m\ ds, n\ ds, p\ ds)$$

$$m,n,p = 0,\pm1,\pm2,\dots,\pm(2N+1)$$

A "pseudo radius" is defined as

$$pseudo\ radius = \max(|m|,|n|,|p|)$$

In this document, "pseudo radius" is simply referred to as "radius".

Comparing (x_i, y_i, z_i) with (x_m, y_m, z_m) we can see that to use "array indexing" to represent a space point, we must know value N , but using "radius indexing", it is independent of value N. For a fixed ds, N represents space domain size. Therefore, radius indexing is domain size independent. It is one advantage of using radius indexing.

Value N determines the maximum radius by

$$maximum\ radius = 2N+1$$

Value N also determines relationships between array indexing and radius indexing by

$$i = m + 2N + 1$$

$$j = n + 2N + 1$$

$$k = p + 2N + 1$$

Going through space points

Usually 3 nested loops are used to go through all space points:

for(i=0;i<4N+3;i++){for(j=0;j<4N+3;j++){for(k=0;k<4N+3;k++){…**handleData(I,j,k)**;…}}}

In most places EMFA does not use the above approach; it goes through all space points using the radius indexing. It goes through space points radius by radius, starting from 0 to the maximum radius.

To enable a class to handle all space points, add GoThroughSphereByIndexes or GoThroughSphereBySpaces to its base class list.

These base classes have a function gothroughSphere; calling this function will start a process of going through every space point, radius by radius.

Before going through points at a radius, it calls a virtual function setRadius; the default implementation of it is

```
virtual RadiusHandleType setRadius(int radius){r=radius; return NeedProcess;}
```

You may override this function to instruct the system whether you want to process the points at the radius. Following return values are supported:

- NeedProcess – points at the radius will be processed.
- DoNotProcess – points at the radius will not be processed.
- Finish - points at the radius will not be processed. No other radiuses will be processed. The process should finish.
- ProcessAndFinish - points at the radius will be processed. But no other radiuses will be processed. The process should finish.

For a boundary condition module, it needs to process points when radius is the maximum radius. If boundary thickness is larger than 1 then it needs to process points at radiuses equal or larger than the maximum radius minus the thickness. It is easy to handle boundaries in this way. It is another advantage of using radius indexing.

If points at a radius should be processed then function handleData will be called repeatedly for every point at the radius.

Your classes must override function handleData to process each space point.

If GoThroughSphereByIndexes is used as a base class then handleData is given radius index for the space point:

```
virtual void handleData(int m, int n, int p)=0;
```

If GoThroughSphereBySpaces is used as a base class then space location is passed to handleData:

```
virtual void handleData(double x, double y, double z)=0;
```

Memory indexing

EM field memory is a one-dimensional array of field items. Each field item consists of 6 double precision numbers; 3 numbers for a 3D electric field vector and 3 numbers for a 3D magnetic field vector. The array position of a field item determines the space location of the field item.

EM field memory is arranged by radius. Space points of smaller radiuses go first in the array. Therefore the first item is for space point of origin $(0,0,0)$, because it is for radius 0. But for a radius > 0 this rule does not uniquely determine the relationship between array index and space point.

This rule just says that the array is arranged block by block, each block is for one radius. But it does not define item sequence within each block.

In source code RadiusIndex.h there are two functions which define unique item sequence within each block.

```
size_t IndexesToIndex(unsigned radius, int m, int n, int p, int *ret);
```

It returns an item index within the block from a set of radius index (m, n, p). Passing radius into the function is for function performance, actually $radius = \max(|m|, |n|, |p|)$. "ret" passes back an error code. The return value is between 0, including 0, and the number of items in the block, not including the number of items in the block.

```
int IndexToIndexes(unsigned radius, size_t a, int *m, int *n, int *p);
```

It gets a set of radius index (m, n, p) from an item index within the block of the radius. "a" is the item index with the block. It returns an error code.

There are radius index related utility functions you may find useful:

```
size_t pointsAt(unsigned r)
```

It returns a number of space points at a given radius.

```
size_t totalPointsInSphere(unsigned r)
```

It returns total number of space points at and below the given radius.

Memory array index can be calculated by

$$array\ index(m,n,p) = totalPointsInSphere(r-1) + IndexesToIndex(r,m,n,p,\&ret)$$

$$r = \max(|m|,|n|,|p|)\,, r > 0$$

EMFA provides an efficient way of getting memory array index from radius indexing or from 3D array indexing. It uses a 3D array to remember mapping between radius indexes and memory array indexes. To use this service in a class, add RadiusIndexCacheUser to the class' base class list. Make sure an instance of RadiusIndexToSeriesIndex is assigned to RadiusIndexCacheUser. Then, your class may use following macros to get memory array index:

```
#define RADIUSINDEXMAPEXIST (seriesIndex==NULL)?ERR_RADIUS_INDEX_CACHE:ERR_OK
```

Use this macro to make sure that an instance of RadiusIndexToSeriesIndex is assigned.

```
#define SINDEX(m,n,p) seriesIndex->Index((m),(n),(p))
```

Use this macro to get memory array index from radius index.

```
#define CINDEX(i,j,k) seriesIndex->CubicIndex((i),(j),(k))
```

Use this macro to get memory array index from 3D array index.

Note that when going through all space points, a virtual handleData(…) is called repeatedly for every space point in the sequence of array items of the field memory array. Therefore, inside handleData(…) you do not need to use above macros.

Dynamic Link Library Modules

Creating Plugin Classes in a DLL

In a DLL, multiple plugin classes may be created.

You need to add 3 pieces of coding to make a class into a plugin class.

Add Plugin Instance Pointers

The first piece of code you need to add is a pointer array of the class and an unsigned integer for the size of the array, as shown below:

```
YeeFDTD **yeeList = NULL;
unsigned int yeeCount = 0;
YeeFDTDSpaceSynched **yeesynchList = NULL;
unsigned int yeesynchCount = 0;
```

In the above sample code, two plugin classes are supported, YeeFDTD and YeeFDTDSpaceSynched.

Exporting Modules

The second piece of code you need to add is to create an exported function named **CreatePluginInstance** and add a macro in this function for each plugin class, as shown below:

```
#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

        __declspec (dllexport) void* CreatePluginInstance(char *name, double *params);
        __declspec (dllexport) void RemovePluginInstances();

#ifdef __cplusplus
}
#endif /* __cplusplus */

__declspec (dllexport) void* CreatePluginInstance(char *name, double *params)
{
        FDTD *p = NULL;
  if(strcmp(name,"YeeFDTD") == 0)
        {
                CREATEPLUGININSTANCE(YeeFDTD, yeeCount, yeeList);
        }
        else if(strcmp(name, "YeeFDTDSpaceSynched") == 0)
        {
                CREATEPLUGININSTANCE(YeeFDTDSpaceSynched, yeesynchCount, yeesynchList);
        }
        if(p != NULL)
        {
                //class name will be used in forming data file names
                p->setClassName(name);
        }
        return p;
}
```

In the macro, the first argument is the class name, the second is the array size name, and the third is the pointer array name.

In the above code, it also exports a function named RemovePluginInstances. That is the third piece of code you need to add.

Remove Modules

The third piece of code you need to add is a function named RemovePluginInstances and add a macro to the function for each plugin class, as shown below:

```
__declspec (dllexport) void RemovePluginInstances()
{
        REMOVEALLPLUGINS(YeeFDTD, yeeCount, yeeList);
```

```
        REMOVEALLPLUGINS(YeeFDTDSpaceSynched, yeesynchCount, yeesynchList);
}
```

In the macro, the first argument is the class name, the second is the array size name, and the third is the pointer array name.

FDTD module

The abstract class for FDTD is defined in FDTD.h. The name of the abstract class is FDTD; it has a base class FdtdMemory, which provides field memory management service. A field memory pointer points to the field memory:

```
FieldPoint3D *HE;      //field data
```

For a field simulation, if a base data file name is not provided then a temporary file is created with a desired memory size, and the field pointer is mapped to the file. The temporary file is deleted after simulation, that is, when simulation time step reaches the maximum time step.

If a base data file name is provided then a new data file is created for each simulation time step. It closes the data file for the previous time step; re-map the field memory pointer to the new data file.

To keep your module compatible with other modules in using field memory, please assume radius indexing for the field memory. See “Memory Indexing” section of the previous chapter.

There are 4 abstract functions you must implement in your FDTD classes.

```
virtual void cleanup()=0;     //free memory
```

If your class allocates resources which need to be freed then do it in this function.

```
virtual int onInitialized(TaskFile *taskParameters)=0; //called after initialize(...) returns ERR_OK
```

If your class needs initialization then do it in this function. If your class defines configuration values then read those values via “taskParameters”.

```
virtual int moveForward()=0;
```

Your class assumes that the field memory holds the field at the current time; this function assumes that the time moves one step forward and updates the field memory accordingly.

```
virtual void OnFinishSimulation()=0;
```

This function is called when the maximum time step is reached.

Boundary Condition Module

The abstract class for this module is named BoundaryCondition which is defined in BoundaryCondition.h.

You may override an initialization function to do desired initialization such as reading task parameters, for example, set desired boundary thickness:

```
virtual int initialize(double Courant, int maximumRadius, TaskFile *taskParameters);
```

You must override handleData to implement your boundary condition algorithm:

```
virtual void handleData(int m, int n, int p)=0;
```

Because the way function setRadius is coded, function handleData(m,n,p) will only be called for space points at the boundary.

Initial Value Module

The abstract class for this module is named FieldsInitializer, which is defined in EMField.h:

```
/*
        abstract class to provide fields at time=0.
        a subclass should be implemented in a dynamic link library
        to be plugged into a simulation system at runtime.
*/
class FieldsInitializer: public virtual MemoryManUser
{
public:
        virtual int initialize(TaskFile *taskParameters)    = 0;          //read back configurations from a
task file
        virtual double funcE0x(double x, double y, double z) = 0;          //Initial Ex at the point
        virtual double funcE0y(double x, double y, double z) = 0;          //Initial Ey at the point
        virtual double funcE0z(double x, double y, double z) = 0;          //Initial Ez at the point
        virtual double funcB0x(double x, double y, double z) = 0;          //Initial Bx at the point
        virtual double funcB0y(double x, double y, double z) = 0;          //Initial By at the point
        virtual double funcB0z(double x, double y, double z) = 0;          //Initial Bz at the point
        virtual void getField(double x, double y, double z, FieldPoint3D *f)=0; //initialize EM fields at the
point
};
```

Field Source Module

The abstract class for this module is named FieldSource, which is defined in FieldSource.h.

You may override an initialization function to do desired initialization such as reading task parameters:

```
virtual int initialize(double Courant, int maximumRadius, TaskFile *taskParameters);
```

It provides following members:

```
protected:
        double Cdtds;       //Courant number = 1.0 / sqrt(3.0)
        FieldPoint3D *_fields; //EM field. the source will be applied to it
        size_t _timeIndex;    //time step index, in case a source may need it
        double _time;       //time value, in case a source may need it
```

You may override following functions to implement your field source:

```
	virtual RadiusHandleType setRadius(int radius){r = radius; return NeedProcess;}
	virtual void handleData(int m, int n, int p)=0;
```

TF/SF Module

The abstract class for this module is TotalFieldScatteredFieldBoundary, which is defined in TotalFieldScatteredFieldBoundary.h.

```
/*
	abstract class for implementing Total Field/Scattered Field Boundary
	A TF/SF boundary should be implemented in a dynamic link library
	to be plugged into a simulation system at runtime.
*/
class TotalFieldScatteredFieldBoundary: public virtual RadiusIndexCacheUser, public virtual
MemoryManUser
{
protected:
	double Cdtds; //Courant number = 1.0 / sqrt(3.0)
	FieldPoint3D *_fields; //fields to work on
	int maxRadius; //maximum radius
	int gridSize;  //number of space points on one axis, it is 2*maxRadius + 1
	//TFSF boundary
	int firstX, firstY, firstZ, // indices for first point in TF region
		lastX, lastY, lastZ;    // indices for last point in TF region
	//
public:
	TotalFieldScatteredFieldBoundary(void);
	//
	/*
		initialize TF/SF boundary object
		if a derived class overrides this function then it should call the base function first.
		a derived class may define runtime parameters. each parameter appears in a task file as
a line of name=value
		in the derived function the parameter values are read back via taskParameters
	*/
	virtual int initialize(double Courant, int maximumRadius, TaskFile *taskParameters);
	virtual int applyTFSF(FieldPoint3D *fields);
	//relationship of radius indexing m,n,p and cubic indexing i,j,k:
	//i = m + gridSize; j = n + gridSize; k = p + gridSize
	//m,n,p=0,+-1,+-2,...,+-maxRadius
	//i,j,k=0,1,2,...,2*maxRadius, use _fields[CINDEX(i,j,k)] to access fields
	virtual void applyOnPlaneX0(int j, int k)=0;
	virtual void applyOnPlaneX1(int j, int k)=0;
	virtual void applyOnPlaneY0(int i, int k)=0;
	virtual void applyOnPlaneY1(int i, int k)=0;
	virtual void applyOnPlaneZ0(int i, int j)=0;
```

```
        virtual void applyOnPlaneZ1(int i, int j)=0;
};
```

Project List

The following projects are listed in order of dependency. You can download Full source code of these projects from GitHub. You may contribute new code to these projects according to project intentions.

- FileUtil – It is a static library. It provides file related functions. The TaskFile class is in this project.
- ProcessMonitor – It is a static library. It includes utilities for monitoring running of tasks.
- OutputUtility – It is a static library. It includes utilities for showing messages.
- MemoryMan – It is a static library. It includes utilities for managing large size memories.
- MathTools – it is a static library. It includes math related utilities.
- EMField – It is a static library. It defines EM field related classes, including all abstract classes for plugin modules.
- TssInSphere – It is a static library. It provides space derivative estimations of arbitrary orders. It implements Time Space Synchronized FDTD algorithm.
- YeeFDTD – it is a dynamic link library. It is a sample implementation of FDTD plugin module. It provides two subclasses of the abstract class FDTD. Class YeeFDTD implements Yee's FDTD algorithm. Class YeeFDTDSpaceSynched uses 6 YeeFDTD to produce a space synchronized FDTD implementation.
- FieldProviders – It is a dynamic link library. It is a sample implementation of Initial Value plugin module. It provides a class to populate EM field with a Gaussian function.
- FieldDataComparer – it is a static library. It compares two EM field simulation data and generates report files to show comparisons of precisions.
- BoundaryConditionA – it is a dynamic link library. It is a sample implementation of Boundary Condition plugin module.
- TotalFieldScatteredFieldSample – It is a dynamic link library. It is a sample implementation of TF/SF plugin module.
- TssFDTD – it is a dynamic link library. It is a sample implementation of FDTD plugin module. It turns the static project TssInSphere into the dynamic link library, which implements Time Space Synchronized FDTD algorithm.
- FieldSourceSamples – it is a dynamic link library. It is a sample implementation of Field Source plugin module.
- EMconsole – it is a console application. It is an entry point to execute tasks.
- OpenGL3D – it is 3D drawing application. It uses Open GL library to draw 3D fields. It automatically adjusts zooming factors for space locations and vector sizes to produce good 3D field views. You can use mouse to rotate 3D views to examine EM fields from different angles. You can visually see EM fields evolve over time.

- Draw2D – It is a 2D drawing application. It can be used to compare simulation errors of different algorithms or different estimation orders.