

TMain - CRHM main window.

Routines

```
void __fastcall FormCreate(TObject *Sender);
```

1. Changes Application->OnHelp to AppHelp.
2. Sets default directories to CRHM program directory.
3. Global::crhmMain set to Main->Handle for messaging.
4. Creates Global Lists.
5. Moves modules to CRHM by calling MoveModulesToGlobal();
6. Updates models in **Build** menu.
7. Determines if CRHM is DLL version and enables **DLL** menu.

```
void __fastcall FormActivate(TObject *Sender);
```

- 1.

```
void __fastcall FormDestroy(TObject *Sender);
```

1. If (_HHwinHwnd) call *HtmlHelp*(...) and null _HHwinHwnd.
2. Close observation files by deleting ClassData instances in ObsFilesList->Objects.
3. Delete ObsFilesList.
4. Call *ModVarRemove*(*Global::MapVars*) to remove all variable (observation and module).
5. Delete ProjectList.
6. Delete Global::OurModulesList.
7. Delete Global::ModelModulesList.
8. Free all TChart series.
9. Call *ClearListBox4*(...) to
10. If CRHM_DLL defined call *FreeLibrary*(...) for each DLL in OpenDLLs list.
11. Delete OpenDLLs.

```
void __fastcall ExitClick(TObject *Sender);
```

1. calls Application->terminate() to close the CRHM application.

```
void __fastcall DatePicker1Change(TObject *Sender);
void __fastcall Datepicker2Change(TObject *Sender);
void __fastcall UpDownHRUIIdxClick(TObject *Sender, TUDBtnType Button);
1. Updates HRU # label when HRU changed.

void __fastcall UpDownOBSIndxCClick(TObject *Sender, TUDBtnType Button);
1.

void __fastcall OpenClick(TObject *Sender);
1. calls OpenDialog for observation file
2. calls routine OpenObsFile

void __fastcall ObsFileClose(TObject *Sender);
1.

void __fastcall ObsCloseClick(TObject *Sender);
Purpose - close all observation files.
1. generates "First observation File affects Obs timebase" prompt.
2. clears ListBox2
3. deletes all ClassData instances.

bool __fastcall OpenObsFile(TObject *Sender, String FileName);
Purpose - read observation file.
1. Checks if file is already open.
2. If the first observation file sets Global::DTstart and Global::DTend = 0.
3. Status bar text set to "Reading Data File".
4. Instantiates ClassData object.
5. Status bar text set to "Idle".
6. If the first observation file sets Global and DateTimePickers.
7. Remove any of the obsevation variables currently in ListBox1 and
```

ListBox3.

8. Clear ListBox2 and rescan MapVars for observation variables.
9. Update **Observation** menu.

```
void __fastcall BldModelClick(TObject *Sender);
```

Purpose - build current model.

1. Call *ClearModulesClick()*.
2. Load Global::OurModulesList from
3. Search Global::AllModulesList->Strings for Global::OurModulesList modules and put their addresses in Global::OurModulesList->Objects list.
4. Call *InitModules()*

```
void __fastcall RunClick(TObject *Sender);
```

Purpose - Execute CRHM model.

1. If any "Declaration Errors" exit.
2. Call *ClearRunLogs()*.
3. Free any series currently in TChart.
4. Exit if no model ouput requested in ListBox3.
5. Instantiate TChart series required by ListBox3 count.
6. Instantiate array of pointers to be set to variable value pointers.
7. Process ListBox3 variables, find object, decode dimension and layer from string and save in long or float array.
8. Call *InitReadObs()* to clear storage for observation read and function lists.
9. Call *initbase()* to initialise and declare modules.
10. Exit after cleaning up, if there are LogForm errors or the Start time of the model run is greater or equal to the End time.
11. Calculate the long indices, DTmin and DTmax from DTstart, DTEnd and DateTimePicker values DTstartR and DTendR.
12. Initialise try exceptions for "Runtime Error" (Exception class) and module errors (CHMException class).
13. Loop from DTmin to DTmax the following:
 1. Every 168 steps display date and time in status bar.
 2. Call *ReadObs()* to read current observation values.
 3. Loop through all *modules->run()* to execute module code.
 4. Loop through all variables to be displayed and add latest value to

TChart series.

14. Catch exceptions for "Runtime Error" and module errors.

```
void __fastcall ExportClick(TObject *Sender);
```

```
void __fastcall AboutClick(TObject *Sender);
```

Purpose - display About box.

```
void __fastcall ListBox1Click(TObject *Sender);
```

Purpose -Change UpDownOBSIndex to display LAY# and update status bar.

1. If not previously displaying module variable, load UpDownOBSIndex control and updat LabelOBSIndex.
2. Display in status bar module variable help and units.
3. If one or fewer layers reset to "LAY 1".

```
void __fastcall AddListBox1Click(TObject *Sender);
```

Purpose - Handle ListBox1 pop-up.

1. If ClassName == "TListBox" and Name == "ListBox1" do the following:
2. If item in ListBox1 is selected do the following:
3. Create name string with dimension and optional layer.
4. If name string not already in ListBox3 add.

```
void __fastcall HRUsAddListBox1Click(TObject *Sender);
```

```
void __fastcall AddListBox2Click(TObject *Sender);
```

Purpose - Handle ListBox2 pop-up.

1. If ClassName == "TListBox" and Name == "ListBox2" or If ClassName == " TLabel" and Name == "LabelFunct"do the following:
2. If item in ListBox2 is selected do the following:
3. Create name string with dimension and function.
4. If name string is not already in ListBox4 add as follows.
5. Create new TChart line series or point series if sparse data.
6. Add to ListBox4 making the Object a pointer to the TChart series.

7. Call *AddObsPlot(...)* to generate series on TChart.

```
void __fastcall ListBox2Click(TObject *Sender);
```

Purpose - Change UpDownOBSIndex to display OBS# and update status bar.

1. If not previously displaying observation, load UpDownOBSIndex control and updat LabelOBSIndex.
2. Display in status bar observation variable help, units and file name.
3. Update function label.

```
void __fastcall ListBox3Click(TObject *Sender);
```

Purpose - delete module variable from ListBox3.

1. If model not yet run, i.e. SeriesCnt <= 0, simply delete item from ListBox3 and exit.
2. Search TChart cdSeries for variable name and then free series.
3. Move following series up to fill hole.
4. Delete module variable from ListBox3.

```
void __fastcall DeleteListBox3Click(TObject *Sender);
```

```
void __fastcall NegateListBox3Click(TObject *Sender);
```

```
void __fastcall ListBox4Click(TObject *Sender);
```

```
void __fastcall DeleteListBox4Click(TObject *Sender);
```

```
void __fastcall NegateListBox4Click(TObject *Sender);
```

```
void __fastcall ClearListBox4(TObject *Sender);
```

1. Free all Tchart series defined by ListBox4.
2. Call *ListBox4-Clear()*.

```
void __fastcall FunctionListBox2Click(TObject *Sender);
```

Purpose - Display RadioGroup1

1. Set RadioGroup1 = Funct.
2. Show RadioGroup1 control.

```
void __fastcall RadioGroup1Click(TObject *Sender);
```

Purpose - update observation function display label.

1. Set Funct = RadioGroup1->ItemIndex.
2. Set LabelFunct->Caption = Function.
3. Hide RadioGroup1 control.

```
void __fastcall ConstructClick(TObject *Sender);
```

```
void __fastcall InitModules(TObject *Sender);
```

```
void __fastcall ClearModules1Click(TObject *Sender);
```

```
void __fastcall Analysis1Click(TObject *Sender);
```

```
void __fastcall PrjOpenClick(TObject *Sender);
```

```
void __fastcall PrjExitClick(TObject *Sender);
```

```
void __fastcall PrjCloseClick(TObject *Sender);
```

```
void __fastcall PrjSaveAsClick(TObject *Sender);
```

1. Call SaveDialogPrj-Execute().
2. Call *SaveProject()*.

```
void __fastcall PrjSaveClick(TObject *Sender);
```

1. Call *SaveProject()* if project file name not empty,
2. Else call *PrjSaveAsClick()*,

```
void __fastcall PrjAutoRunClick(TObject *Sender);
```

```
void __fastcall PrjReportClick(TObject *Sender);
```

```
void __fastcall StatSaveStateAsClick(TObject *Sender);
```

```
void __fastcall StatSaveClick(TObject *Sender);
void __fastcall StatOpenInitClick(TObject *Sender);
void __fastcall LogClick(TObject *Sender);
void __fastcall ParametersClick(TObject *Sender);
void __fastcall SqueezeParams(TObject *Sender);
void __fastcall HTMLhelp1Click(TObject *Sender);
void __fastcall ShapeClick(TObject *Sender);
void __fastcall Recall1Click(TObject *Sender);
void __fastcall Chart1ClickBackground(TCustomChart *Sender, TMouseButton Button, TShiftState Shift, int X, int Y);
void __fastcall DLL1OpenClick(TObject *Sender);
void __fastcall DLL1CloseALLClick(TObject *Sender);
void __fastcall LabelFunctClick(TObject *Sender);
```

Purpose - to select next observation function

1. If function is last goto first otherwise increment Funct.
2. Set LabelFunct->Caption to Fstrings[Funct];

```
void __fastcall VariablesClick(TObject *Sender); private: // User declarations
void __fastcall AddObsPlot(TObject *Sender, ClassVar *thisVar, TLineSeries
*cdSeries, String S, TFun Funct);
```

Purpose - to display observation or function of observation on the TChart.

1. If the desired end time after defined data exit.
2. Calculate the DTmin and DTmax data indices.
- 3.

```
void __fastcall FreeChart1(TObject *Sender);
```

```
void __fastcall SaveProject(TObject *Sender);
```

Purpose to save model to file. All strings are inserted into an instance ProjectList of type TStringList. When completed the list is saved to file using *ProjectList->SaveToFile()*. Finally *ProjectList->clear()* is called

1. Display on screen Project file name.
2. Add Description.
3. Add Dimensions.
4. Add Observations.
5. If CRHM_DLL defined add DLLs.
6. Add Modules.
7. Add Dates.
8. Add Parameters.
9. Add Initial_State.
10. Add Display_Variable.
11. Add Display_Observation.
12. Call *ProjectList->SaveToFile()*.
13. *ProjectList->clear()*.

```
void __fastcall TMain::SaveState(TObject *Sender);
```

```
void __fastcall TMain::ReadStateFile(TObject *Sender, bool & GoodRun);
```

```
void __fastcall TMain::OnHint(TObject *Sender);
```

```
bool __fastcall AppHelp(Word Command, int Data, bool &CallHelp);
```

```
void __fastcall Update_Main(TDim Dim, long dim);
```

```
bool __fastcall DllinUse(String FileName);
```

```
void __fastcall DllFileClose(TObject *Sender);
```

```
void __fastcall CompactDlls(void);
```

```
bool __fastcall OpenDLLFile(String FileName);
```

```
void __fastcall DllDelete(String FileName);

void __fastcall UpDateModelMenu(void);

protected: void __fastcall WMGetMinMaxInfo(TWMGetMinMaxInfo &Msg); // prototype for msg handler

void __fastcall WMMainUpdate(TMessage &Message); // prototype for msg handler

void __fastcall WMMainStatus(TMessage &Message); // prototype for msg handler

BEGIN_MESSAGE_MAP
MESSAGE_HANDLER(WM_GETMINMAXINFO,TWMGetMinMaxInfo,
WMGetMinMaxInfo)
MESSAGE_HANDLER(WM_CRHM_Main_UPDATE,TMessage,
WMMainUpdate)
MESSAGE_HANDLER(WM_CRHM_Main_STATUS,TMessage,
WMMainStatus) END_MESSAGE_MAP(TForm)

public: // User declarations __fastcall TMain(TComponent* Owner);

TLineSeries **cdSeries; long SeriesCnt;

TStringList *ObsFilesList; TStringList *ProjectList; TStringList *OpenDLLs;

bool SaveStateFlag; bool OpenStateFlag; bool ProjectOpen;

typedef void __declspec(dllimport) LoadModuleType(String DllName);

LoadModuleType *LoadUserModules;

TDateTime ProjectFileDialog;
```

TBldForm - Window to assemble models.

```
class TBldForm : public TForm{
    __published: // IDE-managed Components
        TListBox *ListBox1; // modules available.
        TListBox *ListBox2; // modules selected
        TLabel *Label1; // "modules available".
        TLabel *Label2; // "modules selected".
        TBitBtn *BitBtn1; // "Build".
        TBitBtn *BitBtn2; // "Cancel".
        TStringGrid *StringGrid1;
        TButton *Check; // "check".
        TStatusBar *StatusBar1;
    void __fastcall FormActivate(TObject *Sender);
    void __fastcall ListBox2Click(TObject *Sender);
    void __fastcall BitBtn1Click(TObject *Sender);
    void __fastcall BitBtn2Click(TObject *Sender);
    void __fastcall ListBox1MouseDown(TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y);
    void __fastcall CheckClick(TObject *Sender);
    void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
    void __fastcall FormDeactivate(TObject *Sender);
```

```

private: // User declarations

    TStringList *List;

    TStringList *ListNotFound;

public: // User declarations

    __fastcall TBldForm(TComponent* Owner);

    void __fastcall DisplayModule(String S);

    long __fastcall CheckModel();

    bool Execute;

};

typedef String KeyType2; //

typedef pair<KeyType2, KeyType2> Pairstr; //

typedef multimap<KeyType2, KeyType2> Mapstr; //

```

Member Functions.

void __fastcall FormActivate(TObject *Sender);

1. Clears status bar.
2. Clears Mapgetvar, Mapreadvar, Mapdeclvar and Mapdeclpar.
3. Sets Global::BuildFlag = BUILD.
4. Traverses Global::AllModulesList calling each modules *decl()* function thus populating the MAPs.
5. Sets Global::BuildFlag = DECL.
6. Traverses Global::AllModulesList copying module names and pointers to ListBox1.
7. Clears ListBox2.
8. Sets **GiveWarning** if Global::OurModulesList is not empty.
9. Traverses Global::OurModulesList copying module names and pointers to ListBox2.

10. Sets **Execute** to false.

void __fastcall ListBox2Click(TObject *Sender);

Calls *DisplayModule(S)* where S is item selected in ListBox2.

void __fastcall BitBtn1Click(TObject *Sender); // Build

1. Calls *CheckModel()*
2. Calls *return* if any errors reported and displays an informational MessageBox.
3. If existing model (**GiveWarning** true) displays an informational MessageBox asking if parameter values should be saved first.
4. Call *Main->ClearModulesClick(Sender)*.
5. If

void __fastcall BitBtn2Click(TObject *Sender);

Closes the **BldForm** form.

**void __fastcall ListBox1MouseDown(TObject *Sender,
TMouseButton Button, TShiftState Shift, int X, int Y);**

If mbRight then Add/Delete module from ListBox2, otherwise call *DisplayModule()*.

void __fastcall CheckClick(TObject *Sender);

Calls *CheckModel()*.

**void __fastcall FormClose(TObject *Sender, TCloseAction
&Action);**

Clears ListBox1 and ListBox2.

void __fastcall FormDeactivate(TObject *Sender);

CRHM variables and routines.

```
TStringList *List;  
TStringList *ListNotFound;  
bool Execute;  
void __fastcall DisplayModule(String S);
```

S is the name of module to be displayed.

1. Set TStringGrid options.
2. Reset all TStringGrid cells to "".
3. Set **ColCount** = 5 and **RowCount** = 1 and insert titles.
4. Insert Observations used by module.
5. Insert Variables used by module.
6. Insert Variables declared by module.
7. Inser Parameters declared by module.

```
long __fastcall CheckModel();
```

Checks Model for errors.

Local Storage.

TStringList *List - variables declared earlier <srcName + varName, useage count>

TStringList *ListUsed - variables needed <srcName + varName, index to ListBox2 of module using variable>

index is made negative if "srcName + varName" is not found or wild '*'.
If the wild variable is determine the index is made positive again.

Check that any module used exists in model.

1. Creates TStringList List and sets options todupAccept and Sorted = false.

2. Creates TStringList ListNotFound and sets options todupAccept and Sorted = false.
3. Clears status bar.
4. Traverses ListBox2 and check that the model modules 'declgetvar' modules are in the ListBox2 module list.
 1. If not, if the module exists in ListBox1 add the module to ListBox2,
 2. else if the module is not specified explicitly '*', then ignore,
 3. else log error that required module does not exist.
5. If any additional added to model repeat traverse of ListBox2.

Check that any module 'declgetvar' exists as a modules 'declvar'.

1. Traverses ListBox2 and for each module do the following:
 1. Insert in TStringList List all the current modules 'declvar' variables.
 2. Loop through current modules 'declgetvar' and verify they exist in List from earlier modules.
 1. If not add to ListNotFound,
 2. Else increment useage in List and .

Check throw

CRHM major class objects.

Global - contains CRHM shared objects.

ClassData - observation file object.

ClassVar - observation or module variable object.

ClassPar - CRHM parameter object.

Administer - object to link user DLL's to CRHM program.

ClassModel - simple object holding model name and required modules.

ClassModule - all user modules derived from this object.

parser - object to parse parameter strings.

ClassClark - object to implement lag and storage for modules.

Classmacro - object to implement filters.

Classfilter - basic object to implement observation data processing.

Classadd - add a constant to an observation variable.

Classsub - subtract a constant from an observation variable.

Classmul - multiply an observation variable by a constant.

Classdiv - divide an observation variable by a constant.

Classconst - define a new observation variable equal to a

constant.

ClassFtoC - temperature conversion

ClassRH_WtoI -

Classea - calculate vapour pressure from temperature and humidity.

Classrefwind - reference a wind to a different height.

ClassMissing - replace value outside of range with last 'good' value.

ClassMissing0 - replace value outside of range with zero.

ClassExpand

ClassShrink

ClassSmear

ClassSim - create a simulation observation file.

Classine - generate a series of sine waves.

Classsquare - generate a series of square waves.

Classramp - generate a series of ramps

Classpulse - generate a pulse.

ClassVar - observation or module variable object.

```
class __declspec(dllexport) ClassVar {  
  
typedef void (ClassVar:: *SomeFunct) (void);  
  
public:  
  
string module; // module creating variable  
  
string name; // variable name  
  
string help; // description given in call  
  
string units; // units given in call  
  
TVar varType; // enum TVar {none, Int, Float, notused, Read, ReadI,  
ReadF}  
  
long dim; // number of HRUs  
  
long lay; // layer dimension, 0 - no layers else the number of layers  
  
bool local; // inhibits display in ListBox1  
  
bool optional; // when an observation indicates that they variable is not  
mandatory.  
  
bool PointPlot; // normal plotting mode is line. Used for sparse data.  
  
float * values; // simple floating point variable  
  
long * ivalue; // simple long variable  
  
float ** layvalues; // multi-layer floating point variable  
  
long **ilayvalues; // multi-layer long variable  
  
long offset; // when an observation variable gives data column.
```

```

long cnt; // when an observation variable gives dimension of observation.

ClassData *FileData; // when an observation points to file.

ClassVar *FunctVar; // pointer to variable used by UserFunct

SomeFunct UserFunct; // pointer to variable function. typedef void
(ClassVar:: *SomeFunct) (void);

ClassVar(string module = "none", string name = "none"): module(module),
name(name), varType(none), dim(0), lay(0), optional(false),values(NULL),
ivalue(NULL), layvalues(NULL), ilayvalues(NULL),
local(false),offset(0), cnt(0), FileData(NULL), UserFunct(NULL),
FunctVar(NULL), PointPlot(false) {};

ClassVar(string module, string name, TDim dimen,string help, string units,
TVar varType, bool PointPlot = false);

ClassVar(string module, string name, long dim,string help, string units,
TVar varType, bool PointPlot = false);

ClassVar(string module, string name, long cnt, long offset, ClassData *
FileData): module(module), name(name), varType(Read), dim(cnt), lay(0),
optional(false), local(false), offset(offset), cnt(cnt), FileData(FileData),
UserFunct(NULL), FunctVar(NULL), values(NULL), ivalue(NULL),
layvalues(NULL), ilayvalues(NULL), PointPlot(false) {};

virtual __fastcall ~ClassVar();

ClassVar(const ClassVar & Cl);

ClassVar & operator=(const ClassVar & Cl);

float &operator[](int ii) {return values[ii];}

virtual void ClassVarData(); // reads current interval values from
observation file data buffer into variable.

virtual void ReadVar(void);

```

```

virtual string UserName(void){return module + ' ' + name;};

virtual void Avg(void); // function handling average value.

virtual void Min(void); // function handling minimum value.

virtual void Max(void); // function handling maximum value.

virtual void Dtot(void); // function handling daily sum of value.

virtual void Tot(void); // function handling sum of value.

virtual void First(void); // function handling first value of day.

virtual void Last(void); // function handling last value of day.

};


```

General.

Ways of creating variables are:

1. Modules can create module variables by calling *declvar(...)* in their *.dec routine.
2. Observation variables are created by a call to *declread(...)* in *ClassData::DataReadFile(...)*.
3. *Tmain::AddObsPlot(...)* calls the ClassVar copy constructor when displaying observation functions.
- 4.

Data Members.

TVar varType.

```
// enum TVar {none, Int, Float, notused, Read, ReadI, ReadF};
```

The variable is created as long or floating point depending on the value of varType.

When observation files are loaded before the model modules, the observation variables have initial values of varType equal to **Read** as the data type is indeterminate until a module using the observation is opened.

TDim dimen.

```
// enum TDim {ONE=1, TWO, THREE, FOUR, FIVE, SIX, NHRU,  
NOBS, NLAY, NDEF};
```

For variables only NHRU and NLAY are permissible for values for **dimen**.

The following variables are set as follows depending upon the value of TDim in the call to *declvar(...)*.

```
long dim; // NHRU dimension.  
  
long lay; // secondary array dimension.  
  
// dim = getdim(NHRU);  
  
// if (dimen == NLAY) lay = getdim(dimen); else lay = 0;
```

N.B.

Only one of the following 4 pointers (values, ivalue, layvalues and ilayvalues) is allocated storage and the remainder are left as NULL. This is different from how they are handled for ClassPar.

float * values.

Pointer to allocated one dimensional floating point array. NULL if long parameter.

long *ivalue;

Pointer to allocated long array one dimensional. NULL if floating point parameter.

float ** layvalues;

Two dimensional pointer array to which storage is assigned when the parameter is floating point, otherwise is NULL. The first dimension is *lay* and the second is *dim*.

long **ilayvalues;

Two dimensional pointer array to which storage is assigned when the parameter is long, otherwise is NULL. The first dimension is *lay* and the second is *dim*.

Member Functions.

Constructors.

1. default

```
ClassPar(string module = "none", string param = "none") :  
    module(module), param(param), dim(0), values(NULL), ivalue(NULL),  
    layvalues(NULL), ilayvalues(NULL) {};
```

2. used

```
ClassPar(string module, string param, TDim dimen, string valstr, float  
minVal, float maxVal, string help, string units, TVar varType, int defdim);
```

Destructor

```
virtual __fastcall ~ClassVar();
```

Copy constructor

```
ClassVar(ClassVar &p);
```

Create copy allocating new storage for data members.

Operator =

```
float &operator=(ClassVar &p);
```

Operator []

```
float &operator[](int ii) {return values[ii];}
```

ClassPar - CRHM parameter object.

```
class __declspec(dllexport) ClassPar {  
  
public:  
  
    string module; // module creating/using the parameter.  
  
    string param; // parameter name.  
  
    string help; // description given in module call.  
  
    string units; // units given in module call.  
  
    TVar varType; // enum TVar {none, Int, Float, notused, Read, ReadI,  
    ReadF};  
  
    TDim dimen; // enum TDim {ONE=1, TWO, THREE, FOUR, FIVE, SIX,  
    NHRU, NOBS, NLAY, NDEF};  
  
    long dim; // primary array dimension.  
  
    long lay; // secondary array dimension.  
  
    // value of varType determines parameter type and determines if.  
  
    float * values;  
  
    long *ivalue;  
  
    float ** layvalues;  
  
    long **ilayvalues;  
  
    string valstr; // fill string passed to parser to set default parameter values.  
  
    float minVal; // minimum parameter value defined by module author.  
  
    float maxVal; // maximum parameter value defined by module author.
```

```

ClassPar(string module = "none", string param = "none")

    : module(module), param(param), dim(0), values(NULL),
  ivalues(NULL), layvalues(NULL), ilayvalues(NULL)

{};

ClassPar(string module, string param, TDim dimen, string valstr, float
minVal, float maxVal, string help, string units, TVar varType, int defdim);

ClassPar(ClassPar &p); // copy constructor

virtual __fastcall ~ClassPar();

float &operator[](int ii) {return values[ii];}

bool Same(ClassPar &p); // compares parameter data

void Change(ClassPar &p); // changes this values to p

virtual string UserName(void){return module + ' ' + param;};

friend class parser;

};

```

General.

Modules create all parameters by calling *declparam(...)* in their *.dec routine. Note that this is the only way parameters can be instantiated.

Data Members.

TVar varType.

```
// enum TVar {none, Int, Float, notused, Read, ReadI, ReadF};
```

A parameter is created as long or floating point as determined by **varType** being Int or Float.

TDim dimen.

```
// enum TDim {ONE=1, TWO, THREE, FOUR, FIVE, SIX, NHRU,  
NOBS, NLAY, NDEF};
```

A parameter is created to serve one of the following purposes:

1. One dimensional (NHRU) - simple module parameter.
2. One dimensional (NOBS) - used only for observation manipulation.
3. One dimensional (ONE) ... (SIX).
4. TWO dimensional (NLAY, NHRU) - used in multi - layer models.
5. TWO dimensional (1, NDEF) - used to indirectly assign model parameters.

The following variables are set as follows depending upon the value of TDim, dim and lay in the call to *declparam(...)*.

```
long dim; // primary array dimension.
```

```
long lay; // secondary array dimension.
```

```
// if(dim == NLAY) { lay = getdim(dimen); dim = getdim(NHRU);}
```

```
// else if(dimen == NDEF) { lay = defdim; dim = 1;}
```

```
// else if(dimen < NHRU) dim = getdim(dimen);
```

```
// else dim = getdim(NHRU);
```

N.B.

The following 4 pointers (values/layvalues and ivalue/ilayvalues) are always handled as pairs. Storage is allocated to layvalues or ilayvalues and values or ivalue is set equal to the first layer. The other pair are left as NULL. This is different from how they are handled for ClassVar.

float * values.

Pointer to allocated floating point array defined in layvalues[0]. NULL if long parameter.

long *ivalue;

Pointer to allocated long array defined in ilayvalues[0]. NULL if floating point parameter.

float ** layvalues;

Two dimensional pointer array to which storage is assigned when the parameter is floating point, otherwise is NULL. The first dimension is *lay* and the second is *dim*.

long **ilayvalues;

Two dimensional pointer array to which storage is assigned when the parameter is long, otherwise is NULL. The first dimension is *lay* and the second is *dim*.

Member Functions.

Constructors.

1. default

```
ClassPar(string module = "none", string param = "none") :  
    module(module), param(param), dim(0), values(NULL), ivalue(NULL),  
    layvalues(NULL), ilayvalues(NULL) {};
```

2. used

```
ClassPar(string module, string param, TDim dimen, string valstr, float  
minVal, float maxVal, string help, string units, TVar varType, int defdim);
```

Destructor

```
virtual __fastcall ~ClassPar();
```

Copy constructor

```
ClassPar(ClassPar &p);
```

Create copy allocating new storage for data members.

Operator []

```
float &operator[](int ii) {return values[ii];}
```

Compare Parameters

```
bool Same(ClassPar &p);
```

Returns false if any parameter value is different.

Change Parameter

```
void Change(ClassPar &p);
```

Assign parameter values to those given by &p

Return names string

```
virtual string UserName(void){return module + ' ' + param;};
```

Parse default string and assign values

```
friend class parser;
```

ClassData - observation file object.

```
class Classmacro;  
  
class parser;  
  
class __declspec(dllexport) ClassData {  
  
public:  
  
    string DataFileName; // filename from dialogue control  
  
    string Description; // not used  
  
    float **Data; // memory allocated to hold observations. [obs, Line]. If  
    // NULL after instantiating a ClassData object it indicates that an error has  
    // occurred.  
  
    double *Times; // holds sparse times  
  
    double Dt1; // file start time (days)  
  
    double Dt2; // file end time (days)  
  
    long Lines; // # of lines of data (or intervals) in file - calculated from (Dt2 -  
    // Dt1)*Freq  
  
    long DataCnt; // total # of data items/line.  
  
    long MacroCnt; // # of new observations created by filters.  
  
    double Interval; // calculated from (Dt2 - Dt1) (days)  
  
    long Freq; // calculated from 1/Interval  
  
    long IndxMin; // range of data available referenced to base file.  
  
    long IndxMax; // range of data available referenced to base file.
```

```

long ModN; // divisor for data less frequent than basic interval.

Classmacro *myMacro; // Classmacro object to process this files filters.

bool Simulation; // Set when filter 'Sim' is used to synthesize observations.

ClassData(string DataFileName) : DataFileName(DataFileName),
Data(NULL), myMacro(NULL), DataCnt(0), MacroCnt(0),
Simulation(false), ModN(0), Times(NULL) {DataReadFile();};

virtual __fastcall ~ClassData();

void DataReadFile(void);

bool Execute(void) {return (Data); }

};

```

Member Functions.

Constructor.

Creates a new object and calls *DataReadFile()* to actually read observation file.

Destructor

Deletes allocated storage for observation data and sparse times.

void DataReadFile(void);

1. Error is thrown if datafile cannot be opened.
2. The first line of the observation file is discarded. It is assumed to be a user comment.
3. The header lines are read and processed. Characters in column 1 and 2 determines the command.
 1. '#' indicates the end of the header. Ignore line and begin reading

values.

2. '\$\$' assume a comment and continue after ignoring the line.
 3. '\$' and next character not '\$' indicates a filter. Processing is done by [myMacro->addfilter](#).
 4. If not one of the above the line is assumed to be the definition of an observation variable. If any filters have been defined before all observation variables are defined an error is thrown as this is illegal. The variable is processed by [declread\(\)](#).
4. Time variables are next determined as follows:
- If data from a data file
 1. The current position in the data file is saved, i.e. the first line of data.
 2. The time field is read and saved as Dt1. If Global::DTstart is zero indicating the first observation file, it is set equal to Dt1.
 3. Read the time field of the next line and calculate the Interval and daily frequency (Freq).
 4. If Freq == 0, then the SparseFlag is set to TRUE and Freq = 1.
 5. Interval is recalculated as 1/Freq.
 6. IndxMin is set.
 7. Position to the beginning of the last line in the file.
 8. Read the time field and save as Dt2. If Global::DTend is zero indicating the first observation file, it is set equal to Dt2.
 9. If current observation file has more data than the first observation file set Dt2 = Global::DTend.
 10. Calculate the number of data lines to be read.
 11. IndxMax is set.
 12. If daily data increment number of data lines by one.
 13. Reposition to the beginning of the data using the position saved in 1.
 - If synthesizing data using the simulation filter, the values above will have already been set by the filter object.
5. Memory is allocated to hold the observation values
 6. A call is made to myMacro->fixup() to update pointer values.
 7. The cursor is changed to an hourglass.
 8. The following code is contained in a 'try __finally' to return the cursor to normal when finished and a loop to read *Lines* lines.

Only if data from a data file

1. Read the time field
2. If eof() break from loop after setting Lines to current iteration count.
3. If current time is less than that of the previous line, throw an error exception.
4. If current time is great than the last time display a warning and set the SparseFlag.
5. Read the observation data. Ignore extra data but throw an error if data is short.

Call myMacro->execute to execute filters which in the case of 'Sim' also generates observation data.

9. If not Sparse data delete the array Times as it is only required for soarse data.
10. Delete myMacro.
11. Close the data file.
12. Return.

bool Execute(void) {return (Data); }

Data is NULL if an error has occurred in instantiating a ClassData object. Errors include faulty time data, earlier dates occurring after later dates and insufficient number of data on a line. Missing times cause the file to be sparse data. Extra data on a line is ignored.

double *Times;

When the observation file is read by *DataReadFile(void)*, the date/time for every interval is saved in this array. This array is only retained if the file holds sparse data otherwise it is released.

Classmacro - Processes observation file filter calls creating Classfilter objects.

```
class __declspec(dllexport) Classmacro {  
public:  
    Classmacro(ClassData *File); // constructor.  
    ~Classmacro(); // destructor.  
    ClassData *File; // Current ClassData object.  
    TStringList *FilterList; // List to hold filters used by a ClassData object.  
    void addfilter(String Line); // Processes filter lines in observation file.  
    void fixup(void); // Memory is not allocated until after all the header lines  
    in observation file are read.  
    void execute(long Line); // Executed for every observation line read.  
};
```

Member Functions.

Constructor.

Creates a new FilterList to store observation filters used by current ClassData object.

Destructor

Deletes filters in FilterList and then the FilterList.

void addfilter(String Line);

Processes filter lines in observation file to determine filter type. Then

instantiates the filter class and adds the object to the FilterList. If there is an error generates a CHMException.

void fixup(void);

Memory is not allocated until after all header lines in observation file are read. After memory is allocated the FilterList is gone through calling all the filter *->fixup() routines which calculate the actual pointers for observation variables used by the filter.

void execute(long Line);

Executes after every observation line is read. Using the FilterList, it is able to call the doFunction() of every filter. Parameters include Line and using ObsCnt multiple calls are made to handle observations dimensioned greater than 1.

ClassFilter - observation file object.

```
class __declspec(dllexport) Classfilter {  
  
public:  
  
    Classfilter(ClassData *MyObs, String ToVar, String args, String argtypes =  
    ""); // constructor  
  
    virtual ~Classfilter(); // destructor  
  
    ClassData *MyObs; // parent observation file object class.  
  
    String ToVar; // variable name to store output from filter.  
  
    String argtypes; // initialise by filter constructor to combination of "V" and  
    // "C" defining input parameters.  
  
    String args;  
  
    long Vs; // number of filter variable inputs.  
  
    long Cs; // number of filter constant inputs.  
  
    bool Error; // set by Classfilter::error() on detection of error  
  
    long ObsCnt; // dimension of observation variable. Normally one.  
  
    long *DataIndx; // array of offsets of observation variables [# of Vs].  
  
    long *DataObsCnt; // maximum number of observations of this observation  
    // variable  
  
    ClassData **ObsFile; // array of pointers to parent observation files of  
    // variables used by filter [# of Vs].  
  
    float ***Data; // array of pointers to variable data areas [# of Vs].  
  
    double *Constants; // array of constants used by filter [# of Cs].
```

```
virtual void readargs(void); // interprets filter parameters.  
  
virtual void error(String Mess); // Calls exception handling.  
  
void fixup(void); // after data allocation, it is called by Classmacro::fixup()  
to  
  
virtual void doFunc(long Obs, long Line){Data [0] [Obs] [Line] = 0.0;} //  
Execute filter for every observation variable dimensioned.  
  
virtual void doFunctions(long Line); // Executes after every observation file  
line is read calling doFunc(...).  
  
};
```

Member Functions.

Constructor.

From filter *argtypes* allocates memory for constant and variable storage and processing then calls Classfilter::readargs() to process parameters.

Destructor

Deletes memory allocated for constant and variable storage and processing.

virtual void readargs(void);

Processes parameter list of filter. Then instantiates the filter class and adds the object to the FilterList. If there is an error generates a CHMException.

void fixup(void);

Memory is not allocated until after all header lines in observation file are read. The parent Classmacro object calls all the filter *->fixup() routines which calculate the actual pointers for observation variables used by the filter after allocation.

```
virtual void doFunc(long Obs, long Line){Data [0] [Obs] [Line] =  
0.0;}
```

Executes after every observation line is read. Using the FilterList it is able to call the doFunction() of every filter. Parameters include Line and using ObsCnt multiple calls are made to handle observations dimensioned greater than 1.

```
virtual void doFunctions(long Line);
```

The parent Classmacro object calls all the filter *->doFunctions(..) routines which calls its doFunc(...) for all the variables dimensions.

```
void error(String Mess);
```

Generates a CHMException stopping model execution and reports error.

Module Exception Class

This class is used to throw errors from CRHM modules. These exceptions can be thrown in module routines *decl*, *init* and *run* but not by *finish*.

```
enum TExcept {NONE, ERR, DECLERR, WARNING, USER, TERMINATE};  
  
class CHMException {  
  
public:  
  
    string Message; // error message  
  
    TExcept Kind; // type of error  
  
    CHMException() : Message(""), Kind(NONE) {} // default constructor  
  
    CHMException(string Message, TExcept Kind) : Message(Message),  
        Kind(Kind) {};  
  
};
```

Mathematical Errors

The mathematical errors:
"DOMAIN", "SING", "OVERFLOW", "UNDERFLOW", "TLOSS" are handled
by the C++ Exception class using

```
throw Exception(String("error description")) or throw  
Exception("error description")
```

Logging error messages.

The following two routines use Window messaging to transmit text to the CRHM log window. The routine logging the error message has to throw an exception to change program flow.

```
void __fastcall LogError(CHMException Except){
```

```
SendMessage(Global::crhmLog, WM_CRHM_LOG_EXCEPTION,
(unsigned int) &Except, 0);

}

void __fastcall LogError(String S, TExcept Kind){

SendMessage(Global::crhmLog, WM_CRHM_LOG_EXCEPTION1,
(unsigned int) &S, (unsigned int) &Kind);

}
```

declread - process observation variable defined in header of Observation file.

```
void declread(string module, string name, long cnt, long offset, ClassData *  
FileData, string Comment)
```

where

string module - hard coded as "obs" in ClassData::DataReadFile.

string name - name used in observation file header.

long cnt - size of variable, usually one.

long offset - order of definition starting at 0. Same as order in file data line.

[ClassData](#) * FileData // object handling file

string Comment // balance of line.

Operations

1. If cnt is greater than current Global::maxobs update NOBS to allow the observation variable to display properly.
2. Check if observation variable exists in Global::MapVars. If the variable already exists and has a varType >= Read then throw 'duplicate observation variable error' otherwise assign parameter values to this variable and return.
3. Otherwise create a new 'Read' variable with these qualities and insert it into Global::MapVars.

Notes

Observation variables are instantiated by either *declread* or by [ClassModule::declvar](#) depending upon whether the observation file or the model modules are loaded first.

Map Utilization.

The STL container classes **map** and **multimap** are used extensively in CHRM for assembling the CRHM model and keeping track of module names and object addresses. See [GlobalCommon.h](#) for the definitions of the **typedefs** used by the program.

When linking modules together a multimap is used to hold module and variable/parameter names. It is defined as follows.

```
typedef String KeyType2;  
  
typedef pair<KeyType2, KeyType2> Pairstr;  
  
typedef multimap<KeyType2, KeyType2> Mapstr;
```

1. The first item of the pair is the name of the module using the variable from another module.
2. The second item of the pair is the name of the source module concatenated with the variable name.

An example is pair<"pbsm", "obs hru_t">. Note that a single space is used as the separator in the second item.

When keeping track of module object addresses a map is used to hold the source module name concatenated with the variable name and the address of the object. It is defined as follows.

```
typedef string KeyType;  
  
typedef ClassVar *Var;  
  
typedef ClassPar *Par;  
  
typedef pair<KeyType, Var> PairVar;  
  
typedef pair<KeyType, Par> PairPar;
```

```
typedef map<KeyType, Var> MapVar;  
typedef map<KeyType, Par, Classless<KeyType>> MapPar;
```

1. The first item of the pair is the source module name concatenated with the variable name.
2. The second item of the pair is the address of the object.

An example is pair< "obs hru_t", hru_t>. Note that a single space is used as the separator in the first item.

When keeping track of dimensions a map is used to hold the dimension name and its size. It is defined as follows.

```
typedef long Dim;  
typedef pair<KeyType, Dim> PairDim;  
typedef map<KeyType, Dim> MapDim;
```

1. The first item of the pair is the name of the dimension.
2. The second item of the pair is the size of the dimension.

An example is pair< "NHRU", 5>.