Software Design Document (SDD)

Team O

Event Driven DIS PDU Logger (EDDIS system)

Revision: 3.0

Date: 2002/10/28 01:33:16

Miles Izzo Jeremy Harrap Patrick Donelan Bing Chen Nadia Davidson

November 6, 2002

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Scope

This Software Design Description (SDD) describes the detailed structure of the components of the Event Driven DIS PDU Logger and the precise implementation details required to satisfy the requirements as specificted in the Software Requirements Specification (SRS). It is assumed that the reader has read the SRS, since this document also defines the implementation details of the desired behaviour given the requirements within it. This document will build heavily on the SADD and so knowledge of the general system architecture is recommended prior to commencing this document.

1.2 Aim

The design description defined in this document serves multiple purposes:

- To describe the functional structure, data and algorithms to be implemented.
- To identify required system resources.
- To be used to assess the impact of requirement changes.
- To assist in producing test cases.
- To be used to verify complaince with requirements.
- To aid in maintanance activities.

1.3 Intended Audience

In contrast to the Software Requirements Specification (that is written for the client and user), most of this Software Design Description is written for knowledgeable software professionals and designers. Thus the Client will not be within the intended audience for this document, which is:

- Team

- Supervisor
- Auditors and Reviewers

1.4 People

1.4.1 Client

The Client for this project is:

Name:	Dr. Lucien Zalcman
Company:	Defence Department/ Defence Science and Technology Organisation (DSTO)
Address:	Air Operations Division 506 Lorimer Street Fishermans Bend Port Melbourne, 3207
Phone Number:	9626 7704
Email:	`lucien.zalcman@dsto.defence.gov.au`

1.4.2 Team

The team for this project is Team O, consisting of:

Team Member	login ID	Phone Number
Nadia Davidson	nadiad	0412 913 044
Miles Izzo	mgi	0403 071 632
Patrick Donelan	pdonelan	0421 089 007
Jeremy Harrap	jpharrap	0402 211 578
Partick Chen	bingc	0403 284 718

1.4.3 Supervisor

The supervisor for this project is Mei Chi Tam, contactable at the following email addresses:

`mtam@students.cs.mu.oz.au`
`mctam@unimelb.edu.au`

Mei Chi is also contable on 0411 389 980.

1.4.4 External Reviewers/Auditors Details

Reviewer	login ID	Phone Number
Gregory Feely	gafeel	9347 4403
Ko-Chen Wo	kocw	0425 806 206
Aaron Iles	arronli	9340 0503
Chi Zhang	czh	0411 626 856

1.4.5 Course Coordinator

Coordinator	login ID
Ed Kazmierczak	ed

1.5 Definitions and Acronyms

For a complete list of the definitions and acronyms used in the remainder of this document, refer to the Glossary.

Chapter 2

Module Design

The SDD module design of the EDDIS System contains a detailed description of the objects defined in the SADD. It will attempt to define methods, properties, accessors and, to a certain extent, provide algorithms or ways of approaching the coding process.

2.1 Class diagram

Figure 2.1 shows the class diagram for the system. This diagram uses UML to represent the classes. Consult a UML text for a data dictionary (recommended **??**).

2.2 Class Template – filename.cs

2.2.1 Function Template

Each function will be detailed in the format below:

Purpose: Describes the routine.

Prototype: Shows how the method is called, giving a basic prototype (not necessarily with types). For example:

```
void myName( X, Y )
```

Inputs: Description of input parameters (ie `X`).

Outputs: Description of return value or `ref` variables.

Restrictions: Description of any restrictions to be placed on input.

Called by: Routines that call this routine.

Calls: Routines that this routine calls.

Algorithm: Description of algorithm or procedure.

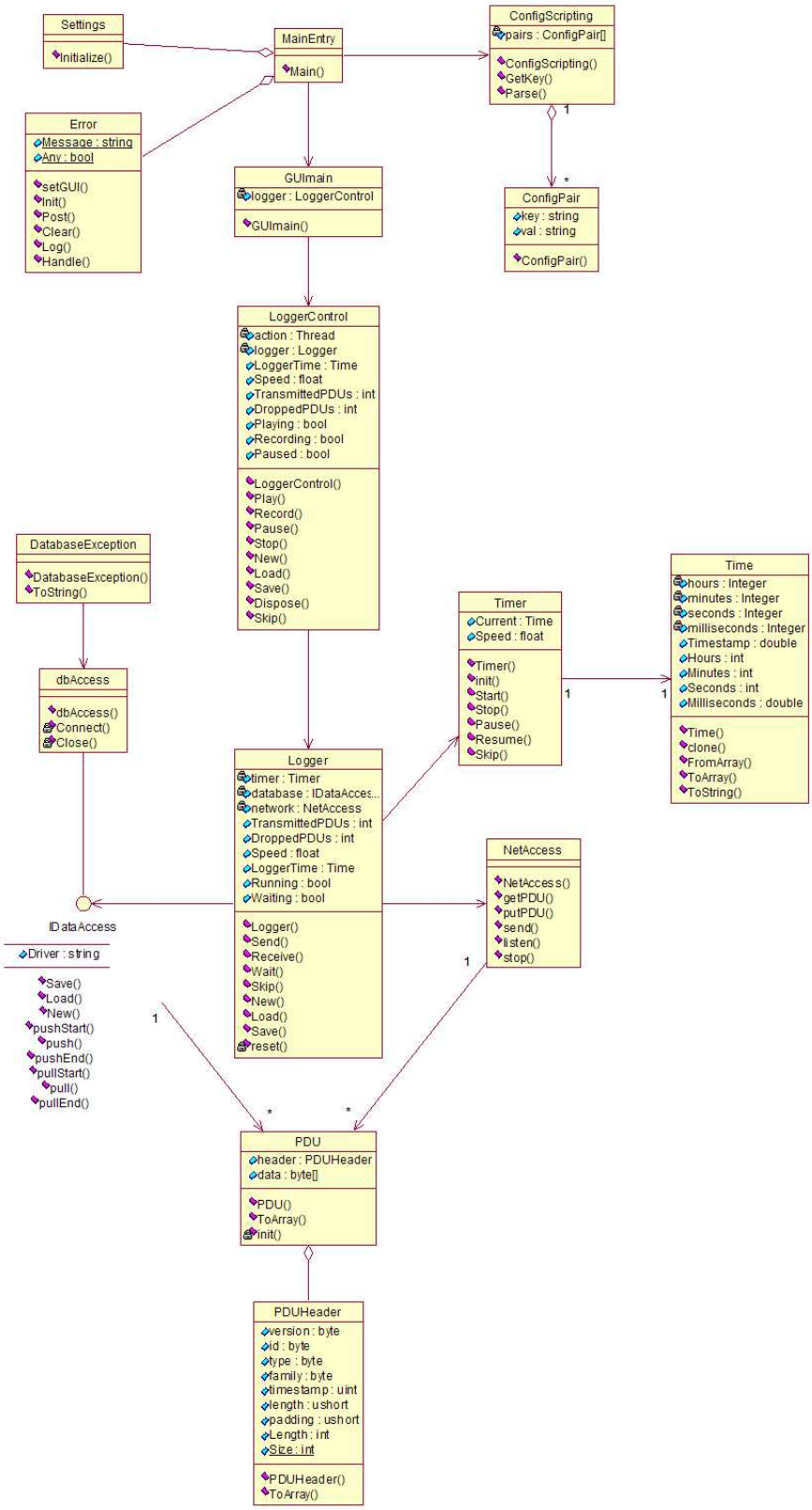

Figure 2.1: Class diagram of system

2.3 The GUImain class – GUImain.cs

This class is called by the `MainEntry` class (the top-level of the program). It shall contain methods to handle events particular to the user interface, provide the user with functionality, and present to the user a graphical view of the data being transmitted.

Note that User Interface events, such as buttons clicked, menu items selected, have been ommitted from this document (with the exception of the `OnTick` event), as they can be easily deduced from the methods contained within this section.

2.3.1 GUImain

Purpose: Constructor for the GUImain form. Initializes any variables or controls required.

Prototype: `public GUImain()`

Inputs: None.

Outputs: None.

Called by: `MainEntry.Main()` method.

Calls: –

- `InitializeComponent()`
- `resetUI()`

Algorithm: –

- Saves the default window title.
- Instantiates the `LoggerControl` class.
- Sets the minimum size of the form.
- Resets the User Interface.
- Starts the User Interface timer.

2.3.2 InitializeComponent

Purpose: Initializes all the controls configured in the Windows Form Designer.

Prototype: `private void InitializeComponent()`

Inputs: None.

Outputs: None.

Restrictions: Any restrictions imposed by the Windows Form Designer.

Called by: `GUImain()`

Calls: Any methods required by the Windows Form Designer.

Algorithm: Automatically generated by the Windows Form Designer. Any code modifications will be on an as-needed basis.

2.3.3 resetUI

Purpose: Reset components related to the User Interface.

Prototype: `private void updateUI()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `GUImain()`, and various button click events (such as the toolbar Record button event).

Calls: –

- `updateToolbar()`
- `updateWindow()`

Algorithm: –

- Reset the currently zoomed portion of the graph to the selected portion from `Settings`.
- Reset the scrollbar.
- Reset the values in the playspeed control.
- Update the window (`updateToolbar()` and `updateWindow()`).

2.3.4 updateWindow

Purpose: Resizes all controls in relation to the form size and any other element that has a variable size.

Prototype: `private void updateWindow()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `GUImain`, the 'Moved' or 'Resize' event of any GUI elements that move or change in size.

Calls: Calls to GUI elements' resize methods/accessors.

Algorithm: –

- Stop the interface from updating its contents (suspend).
- Move `currentTime` so it is docked on the right of the form.
- Resize `eventFrame` (contains Event List and Graph)
- Resize `graphDisplay` according to the new size of `eventFrame` and the position of the splitter.

- Update the Event List size.
- Update the `timeScaleImage` control.
- Resize the graph scrollbars.
- Update the window's title bar.
- Update the time start/end controls.
- Allow the interface to update its contents (resume).

2.3.5 updateToolbar

Purpose: Updates all the Toolbar controls contained in the User Interface.

Prototype: `private void updateToolbar()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: Various button click event methods (such as the toolbar Record click event method), as well as `OnTick()` if the interface is identified as being out-of-date.

Calls: Nothing.

Algorithm: Update each control in the toolbar according to the current state of the logger. Following is a snipped of psuedocode that could be used:

```
toolBarBack.Enabled     = playing;

toolBarRecord.Enabled   = ((!playing && !recording) ||
                          (recording && paused)) &&
                          file_is_empty;

toolBarPlay.Enabled     = (!playing && !recording &&
                          !file_is_empty) ||
                          (playing && paused);

toolBarPause.Enabled    = playing && !paused;
toolBarStop.Enabled     = playing || recording;
toolBarForward.Enabled  = playing;
```

2.3.6 Dispose

Purpose: To safely destroy any components allocated by GUImain.

Prototype: `protected override void Dispose( disposing )`

Inputs: `disposing` – Boolean. True if components are to be disposed of.

Outputs: None.

Restrictions: None.

Called by: The C# garbage collection facility.

Calls: Windows Form Designer dispose code, Base class dispose routine.

Algorithm: –

- Execute Windows Form Designer code.
- Execute custom code to destroy data.

2.3.7 drawScale

Purpose: To handle the drawing of the time scale onto the graph display.

Prototype: `private void drawScale( Graphics g, long width, long height )`

Inputs: `Graphics` object to draw to, and the width and height of the destination object.

Outputs: None.

Restrictions: `Graphics` object must exist. Width and height variables must be valid otherwise the displayed timescale is undefined.

Called by: The timeScaleImage Paint event.

Calls: None.

Algorithm: –

- Confirm that the user wants to display the time scale.
- Calculate what scale the 'ticks' will be.
- Set up the drawing objects.
- Draw all the ticks.

2.3.8 drawNodes

Purpose: To handle the drawing of the nodes (events) onto the graph display.

Prototype: `private int drawNodes( Graphics g, TreeNodeCollection root, int initial )`

Inputs: `Graphics` object to draw to, the root node of the tree to traverse, and the height at which to start drawing on the graph.

Outputs: The y-position (height) of the final node in this (sub)tree.

Restrictions: Inputs `g` and `root` must be non-null objects.

Called by: The graphDisplay Paint event.

Calls: `drawNodes()`.

Algorithm: –

- Recursively loop through the list and draw all events that are:
  1. visible
  2. enabled

Extra info: This method could use the `PDUGraphing` class to draw the event/time graph. This class is described in the Architectural Design (SADD), but not in this document.

2.3.9 drawSelection

Purpose: To draw the current (playback) start and end time lines onto the graph display.

Prototype: `private void drawSelection( Graphics g )`

Inputs: `Graphics` object to draw to.

Outputs: None.

Restrictions: Input `g` must be a non-null object.

Called by: The `graphDisplay` Paint event.

Calls: `Time2Cart()`

Algorithm: –

- Convert the start time value to cartesian coordinates (on the graph).
- Draw the start time line.
- Convert the end time value to cartesian coordinates (on the graph).
- Draw the end time line.

2.3.10 setStartTime

Purpose: To set the start time for playback.

Prototype: `private void setStartTime( int x )`

Inputs: The x-position on the graph that the start time should be set to.

Outputs: None.

Restrictions: Input `x` must be a valid coordinate, otherwise behaviour is undefined.

Called by: The `graphDisplay` Mouse click/move event (left-click).

Calls: –

- `Cart2Time()`

- `updateWindow()`

Algorithm: –

- Check that the x-position is within the bounds of the control.
- Convert the coordinate to a time value using `Cart2Time()`.
- Set the start time only if it's less than the end time.
- Update the window.
- Send a repaint event to the `graphDisplay` component.

2.3.11 setEndTime

Purpose: To set the end time for playback.

Prototype: `private void setEndTime( int x )`

Inputs: The x-position on the graph that the end time should be set to.

Outputs: None.

Restrictions: Input `x` must be a valid coordinate, otherwise behaviour is undefined.

Called by: The `graphDisplay` Mouse click/move event (right-click).

Calls: –

- `Cart2Time()`
- `updateWindow()`

Algorithm: –

- Check that the x-position is within the bounds of the control.
- Convert the coordinate to a time value using `Cart2Time()`.
- Set the start time only if it's greater than the start time.
- Update the window.
- Send a repaint event to the `graphDisplay` component.

2.3.12 Cart2Time

Purpose: Converts a cartesian x-coordinate (on the `graphDisplay`) to its corresponding time value.

Prototype: `public void Cart2Time( long x, ref Time obj )`

Inputs: The input cartesion coordinate `x`, and a reference to the current `Time` object being set.

Outputs: None.

Restrictions: Input `obj` must be a non-null object.

Called by: –

- `drawScale()`
- `setStartTime()`
- `setEndTime()`

Calls: None.

Algorithm: –

- Get the width of the `graphDisplay` component.
- Calculate the difference between the end time and the start time (timestamps).
- Calculate the time by taking the ratio of `x` over the ratio of the width of the component and the difference in time, adding the start time to the final value.
- Using fixed-point integers is highly recommended (probably shift-left by 16).

2.3.13 Time2Cart

Purpose: Converts a time value to its corresponding cartesian x-coordinate (on the `graphDisplay`).

Prototype: `public void Time2Cart( Time obj, ref long x )`

Inputs: The input time value to be converted `obj`, and a reference to cartesion x-coordinate being set.

Outputs: Input `obj` must be a non-null object.

Restrictions: None.

Called by: `drawSelection()`

Calls: None.

Algorithm: –

- Get the width of the `graphDisplay` component.
- Calculate the difference between the end time and the start time (timestamps)
- Calculate the x-coordinate by taking the ratio of the width over the difference in time, and multiply by the difference between `obj.Timestamp` and the start timestamp.
- Using fixed-point integers is highly recommended (probably shift-left by 16).

2.3.14 OnTick

Purpose: To keep the User Interface updated while logging (recording or playback) is taking place.

Prototype: `private void onTick( object sender, EventArgs myEventArgs )`

Inputs: The sender `object`, and any event arguments to be passed to this event.

Outputs: None.

Restrictions: None.

Called by: The C# event handler. Created by `GUImain()`.

Calls: –

- `updateToolBar()`
- `updateWindow()`

Algorithm: –

- If we're playing or recording, update the 'time' indicator on the interface (should be increasing). Also, show the number of PDUs per second we've transmitted.
- If we're recording, show how many PDUs we've received and how many we've dropped.
- If we're playing, show how many PDUs we've sent.
- If the GUI thinks we're playing or recording but the logger (`LoggerControl`) says otherwise, update the toolbar and window to reflect any change in state.

2.4 The MainEntry class – MainEntry.cs

The `MainEntry` class is the main entry point for the application. It contains a Main method which initializes settings and begins such things as the log file. It also checks the keys in the configuration file and initializes the UI before running the applications.

2.4.1 Main

Purpose: The main entry point for the application.

Prototype: `static void Main()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: Nothing.

Calls: –

- `Settings.Initialize()`
- `Error.Init()`
- `Error.Log()`
- `Error.setGUI()`
- `ConfigScripting.Parse()`
- `ConfigScripting.GetKey()`
- `GUImain.GUImain()`

Algorithm: –

- Initialize the default settings.
- Initialize the error reporting system.
- Log (into the error file) that the application has started, and note the time and date.
- Parse the configuration file.
- Load any settings defined in the configuration file (using `GetKey()`).
- Initialize the User Interface (`GUImain()`).
- Link the User Interface to the error reporting system (using `setGUI()`).
- Run the application.
- Log (into the error file) that the application has ended, and note the time and date.

Extra info: This method should be declared `[MTAThread]` as it is invoking a multithreaded application.

2.5 The LoggerControl class – LoggerControl.cs

Ther `LoggerControl` class provides a link between the user interface and the back end (ie the logger), allowing the logging process to be controlled. It defines the following public accessor properties:

TransmittedPDUs The number of PDUs transmitted (sent/received). Of type `Int`. Read-only.

DroppedPDUs The number of PDUs that could not be sent or received due to some error. Of type `int`. Read-only.

LoggerTime The current time in the logger playback/recording. Of type `Time`. Read-only.

Playing Whether or not the logger is currently playing. Of type `bool`. Read-only.

Recording Whether or not the logger is currently recording. Of type `bool`. Read-only.

Paused Whether or not the logger is currently paused. Of type `bool`. Read-only.

Speed The speed of the logger playback (1.0 is realtime).

2.5.1 LoggerControl

Purpose: Constructor for the LoggerControl class. Initializes any variables or controls required.

Prototype: `public LoggerControl()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `GUImain.GUImain()`

Calls: `Logger.Logger()`

Algorithm: –

- Set up all variables.
- Instantiate `Logger` object.

2.5.2 Play

Purpose: To create a new thread for playback of data.

Prototype: `public bool Play()`

Inputs: None.

Outputs: `true` if this method was successful and a new thread could be created to playback data.

Restrictions: None.

Called by: Play button event in `GUImain`.

Calls: `Logger.Send()`

Algorithm: –

- Check if the logger is busy, if so return `false`.
- Create a new thread which runs `Logger.Send()`.
- Return `true`.

2.5.3 Pause

Purpose: To pause a recording or playback thread.

Prototype: `public bool Pause()`

Inputs: None.

Outputs: `true` if this method was successful and the thread could be paused.

Restrictions: None.

Called by: Pause button event in `GUImain`.

Calls: `Logger.Wait()`

Algorithm: –

- Check if currently recording or playing, and if not, return `false`.
- Tell the logger thread to pause its current operation.
- Return `true`.

2.5.4 Stop

Purpose: To stop (abort) a recording or playback thread.

Prototype: `public bool Stop()`

Inputs: None

Outputs: `true` if this method was successful and the thread could be stopped.

Restrictions: None.

Called by: Stop button event in `GUImain`.

Calls: Windows threading methods (detailed below).

Algorithm: –

- Check if currently recording or playing, and if not, return `false`.
- Stop the logger through a call to `Thread.Abort()`.
- Wait a set amount of time (default 10 seconds) for the thread to stop using `Thread.Join()`.
- Return `true`.

2.5.5 Record

Purpose: To create a new thread for starting a recording.

Prototype: `public bool Record()`

Inputs: None.

Outputs: `true` if this method was successful and a new thread could be created to record data.

Restrictions: None.

Called by: Record button event in `GUImain`.

Calls: `Logger.Receive()`

Algorithm: –

- Check if currently recording or playing, if so, return `false`.
- Create a new thread which runs `Logger.Receive()`.
- Return `true`.

2.5.6 Skip

Purpose: To skip the currently playing logger forward or backward by 5%.

Prototype: `public void Skip( SkipEnum dir )`

Inputs: A `SkipEnum` enumeration specifying the direction in which to skip. See section 2.8.10 for a definition of this enumeration.

Outputs: None.

Restrictions: Restricted to the enumeration parameters defined in `SkipEnum`.

Called by: Skip button events in `GUImain` (back and forward).

Calls: `Logger.Skip()`

Algorithm: –

- This method simply forwards its input to `Logger.Skip()`.

2.5.7 Dispose

Purpose: To dispose safely of the logger and end any currently running logger thread.

Prototype: `public void Dispose()`

Inputs: None.

Output: None.

Restrictions: None.

Called by: Exit event in `GUImain`.

Calls: None.

Algorithm: –

- Aborts the logger thread using `Thread.Abort()`.

2.5.8 New, Load, and Save

These methods simply forward their input to the `Logger` methods of the same name. Their prototypes shall be:

```
public bool New()
public bool Load( string fullname )
public bool Save( string fullname )
```

2.6 The Settings class – Settings.cs

The `Settings` class stores all settings and options required for the EDDIS system. All items in the `Settings` class should be declared `static`, as all settings contained within are project-wide. The individual settings need not be defined in this document, as they can be added as needed.

2.6.1 Initialize

Purpose: To initialize all settings and options to their default values.

Prototype: `public static void Initialize()`

Inputs: None.

Outputs: None.

Called by: `MainEntry.Main()`

Calls: Nothing.

Algorithm: –

- Set all values' defaults.

2.7 The Validate class – Validate.cs

The Validate class has been ommitted from the SDD, as it is no longer used in the project. There is an architectural description of the Validate class in the SADD.

2.8 The Logger class – Logger.cs

The Logger class handles operations directly related to logging and playing back network (UDP DIS) data. It defines the following public accessor properties:

TransmittedPDUs The number of PDUs transmitted (sent/received). Of type `Int`. Read-only.

DroppedPDUs The number of PDUs that could not be sent or received due to some error. Of type `int`. Read-only.

LoggerTime The current time in the logger playback/recording. Of type `Time`. Read-only.

Running Whether or not the logger is currently running (playing or recording). Of type `bool`. Read-only.

Waiting Whether or not the logger is currently paused. Of type `bool`. Read-only.

Speed The speed of the logger playback (1.0 is realtime).

2.8.1 Logger

Purpose: Constructor. To initialize a new Logger object.

Prototype: `public Logger()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `LoggerControl.LoggerControl()`

Calls: `reset()`

Algorithm: –

- Call the `reset()` method to reset the logger.
- Set the logger speed to 1.0 (realtime).

2.8.2 reset

Purpose: To perform any initialization necessary to reset the logger. Overloaded.

Prototype: –

- `private void reset()`
- `private void reset( Time time )`

Inputs: A `Time` object containing the start time of the logger.

Outputs: None.

Restrictions: `Time` object must be non-null.

Called by: –

- `Logger()`
- `Send()`
- `Receive()`

Calls: –

- `Timer.Timer()`
- `NetAccess.NetAccess()`
- `IDataAccess` implemented constructor

Algorithm: –

- Overloaded `reset` with no parameters calls `reset` with the currently set start time (most likely from `Settings`).
- Instantiate `Timer` object.
- Instantiate `NetAccess` object.
- Instantiate `IDataAccess` implemented object.

2.8.3 Send

Purpose: To continuously retrieve data from the database and send the data over the network.

Prototype: `public void Send()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `LoggerControl.Play()`

Calls: –

- `reset()`
- `IDataAccess.pullStart()`
- `NetAccess.send()`
- `Timer.Start()`
- `IDataAccess.pull()`
- `NetAccess.putPDU()`
- `Timer.Stop()`
- `NetAccess.stop()`
- `IDataAccess.pullEnd()`

Algorithm: –

- Reset the logger.
- Prepare the database for selecting.
- Prepare the network for sending.
- Get the first PDU from the database, if there is one. If not, return.
- Start the timer.
- Loop while running, transmitting over the network and retrieving from the database.
- Stop the timer.
- Shutdown the network and database connections.

2.8.4 Receive

Purpose: To continuously retrieve data from the network and store it in the database.

Prototype: `public void Receive()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `LoggerControl.Record()`

Calls: –

- `reset()`
- `IDataAccess.pushStart()`
- `NetAccess.listen()`
- `Timer.Start()`
- `NetAccess.getPDU()`
- `IDataAccess.push()`
- `Timer.Stop()`
- `NetAccess.stop()`
- `IDataAccess.pushEnd()`

Algorithm: –

- Reset the logger.
- Prepare the database for inserting.
- Prepare the network for receiving/listening.
- Start the timer.
- Loop while running, receiving from the network and inserting into the database.
- Stop the timer.
- Shutdown the network and database connections.

2.8.5 Wait

Purpose: To pause the current operation (send or receive).

Prototype: `public void Wait( bool wait )`

Inputs: `true` to pause the current object's operation, `false` to resume.

Outputs: None.

Restrictions: None.

Called by: `LoggerControl.Pause()`

Calls: –

- `Timer.Pause()`
- `Timer.Resume()`

Algorithm: –

- If we want to pause, and we weren't already paused, then pause the timer.
- If we want to resume, and we were currently paused, then resume the timer.

2.8.6 Skip

Purpose: To skip the logger's current time forward or backward by 5% of the total length.

Prototype: `public void Skip( SkipEnum dir )`

Inputs: A `SkipEnum` enumeration specifying the direction in which to skip. See section 2.8.10 for a definition of this enumeration.

Outputs: None.

Restrictions: Restricted to the enumeration parameters defined in `SkipEnum`.

Called by: `LoggerControl.Skip()`

Calls: –

- `Wait()`
- `Timer.Skip()`
- `IDataAccess.pullEnd()`
- `IDataAccess.pullStart()`
- `IDataAccess.pull()`

Algorithm: –

- Calculate what 5% actually is.
- Check if skipping back/forward 5% will go outside the range of playback, if so return without doing anything.
- Pause the logger (using `Wait()`).
- Skip the timer forward/backward by the 5% value.
- Shutdown the database.
- Start the database (`pullStart()`) at the new time value reported by `Timer().Current`.
- If there aren't any items in the database after this point, stop the logger from running.
- Pull the first item out of the database.
- Resume the logger (using `Wait()`).

2.8.7 New

Purpose: To provide a hook to the database object belonging to this logger, and provide the 'New database' functionality.

Prototype: `public bool New()`

Inputs: None

Outputs: Result of the `IDataAccess.New()` operation

Restrictions: None

Called by: `LoggerControl.New()`

Calls: `IDataAccess.New()`

Algorithm: –

- Should just call the database `New` method and return the result

2.8.8 Load

Purpose: To provide a hook to the database object belonging to this logger, and provide the 'Load database' functionality.

Prototype: `public bool Load( string fullname )`

Inputs: Filename to be passed to the database `Load` method

Outputs: Result of the `IDataAccess.Load()` operation

Restrictions: Inherited from database

Called by: `LoggerControl.Load()`

Calls: `IDataAccess.Load()`

Algorithm: –

- Should just call the database `Load` method and return the result

2.8.9 Save

Purpose: To provide a hook to the database object belonging to this logger, and provide the 'Save database' functionality.

Prototype: `public bool Save( string fullname )`

Inputs: Filename to be passed to the database `Save` method

Outputs: Result of the `IDataAccess.Save()` operation

Restrictions: Inherited from database

Called by: `LoggerControl.Save()`

Calls: `IDataAccess.Save()`

Algorithm: –

- Should just call the database `Save` method and return the result

2.8.10 SkipEnum

This enumeration is used to specify which direction to skip the logger during playback. It shall be defined:

```
public enum SkipEnum { Forward, Backward };
```

2.9 The Time class – Time.cs

The `Time` storage class is used to store time and convert between different time formats (ie. timestamp to hours/minutes/seconds to string).

Since this object is a storage class, the *Called by* field in the method/property description will be "Many methods in this system". It is not necessary to know what calls these Time methods/properties, as it is independent from the system.

2.9.1 Time

Purpose: An overloaded method to perform any initialization necessary to create a `Time` object.

Prototype: –

- `public Time()`
- `public Time( double stamp )`

Inputs: A timestamp to initialize this object to.

Outputs: None.

Restrictions: None.

Called by: Many methods in the system.

Calls: `Timestamp`

Algorithm: –

- Set the timestamp to `stamp`.
- If no parameters are given, set the timestamp to `0`.

2.9.2 Hours

Purpose: Gets or sets the current hours stored in this object.

Prototype: `public long Hours`

Inputs: A `long` value for hours.

Outputs: The current hours stored within this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Get: Returns the hours stored in this object.
- Set: Sets the hours stored in this object.

2.9.3 Minutes

Purpose: Gets or sets the current minutes stored in this object (0-59).

Prototype: `public long Minutes`

Inputs: A `long` value for minutes.

Outputs: The current minutes stored within this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Set the internal minutes property using:

  `minutes = value % 60`
- Set the hours if the value specified is > 59:

  `Hours = value / 60`

2.9.4 Seconds

Purpose: Gets or sets the current seconds stored in this object (0-59).

Prototype: `public long Seconds`

Inputs: A `long` value for seconds.

Outputs: The current seconds stored within this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Set the internal seconds property using:

  `seconds = value % 60`

- Set the minutes if the value specified is > 59:

  `Minutes = value / 60`

2.9.5 Milliseconds

Purpose: Gets or sets the current milliseconds stored in this object (0-1).

Prototype: `public double Milliseconds`

Inputs: A `double` value for milliseconds.

Outputs: The current milliseconds stored within this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Set the internal milliseconds property using:

  `milliseconds = value % 1.0`

- Set the seconds if the value specified is ≥ 1.0:

  `Seconds = (int)value`

  Note that the `(int)` typecast is assumed to truncate – not round – the value.

2.9.6 clone

Purpose: Creates a copy of this `Time` object.

Prototype: `public Time clone()`

Inputs: None.

Outputs: A copy of this object (`Time`).

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: Creates a new `Time` object. A couple of ways to do this:

1. Create a new `Time` object and assign this object's timestamp to it.
2. Use the C# method, `MemberwiseClone()` to create a 'shallow' copy of this object. Shallow means copying "the non-static fields of the original object" [13].

2.9.7 ToString

Purpose: Overloaded. Creates a string representation of the `Time` object.

Prototype: –

1. `public override string ToString()`
2. `public string ToString( long multiply )`
3. `public string ToString( string dec_separate )`

Inputs: –

1. None.
2. A value to multiply the timestamp by.
3. The string to use instead of a decimal point. If an empty string is given, no decimal places (for milliseconds) should be returned (only the time in hh:mm:ss form).

Outputs: The `string` representation of this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: For each overloaded method:

1. Return `this.ToString( "" )`.
2. Return `Timestamp*multiply`, converted to a string using `ToString` on the value.
3. Return a string of the form hh:mm:ss with optional decimal point.

2.10 The NetAccess class – NetAccess.cs

The Network Access class is used to send and read UDP packets over a broadcast network.

2.10.1 NetAccess

Purpose: Constructor for the NetAccess class. Initializes network access by creating network sockets.

Prototype: `public NetAccess( int netPort, IPAddress netAddress )`

Inputs: A port number and an IP address to use for transferring data.

Outputs: None.

Restrictions: None.

Called by: `Logger.Logger()`

Calls: None.

2.10.2 listen

Purpose: Creates and binds the network socket to a port, and begins buffering network packets.

Prototype: `public void listen()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `Logger.Receive()`

Calls: None.

Algorithm: –

- Create a socket for receiving data.
- Initialize the IP endpoint.
- Handle any errors that may occur elegantly.

2.10.3 send

Purpose: Initializes the send socket using high-level abstraction: UdpClient.

Prototype: `public void send()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `Logger.Send()`

Calls: None.

Algorithm: –

- Create the `UdpClient` object.
- Initialize the IP endpoint.
- Handle any errors that may occur elegantly.

2.10.4 getPDU

Purpose: Grabs a network packet from the receive buffer if one is waiting.

Prototype: `public bool getPDU( ref PDU pdu )`

Inputs: Reference to a `PDU` object to receive the network data into.

Outputs: Returns `true` if there was a packet ready, and was received successfully.

Restrictions: None.

Called by: `Logger.Receive()`

Calls: None.

Algorithm: –

- Create a new buffer to store the packet data.
- Fill the buffer.
- Check if the received data was at least as big as a PDU header.
- Fill out a `PDUHeader` object.
- Check that the size reported in the header matches the size of the packet.
- Fill out a `PDU` object.
- Return `true`.
- If any of these checks above fail, return `false`.

2.10.5 PutPDU

Purpose: Puts a PDU out onto the network.

Prototype: `public void PutPDU( ref PDU pdu )`

Inputs: A single `PDU` object to send over the network.

Outputs: None.

Restrictions: None.

Called by: `Logger.Send()`

Calls: None.

Algorithm: –

- Convert the PDU to a byte array.
- Send the PDU over the network using our `UdpClient` object.

2.10.6 stop

Purpose: Closes and shuts down sockets (this empties the network buffer which may contain packets waiting to be received).

Prototype: `public void stop()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Receive()`

Calls: None.

Algorithm: –

- If sending, reset any flags and return.
- If receiving, shutdown and close the socket.
- Throw an exception if the socket still remains open.

2.11 The Database Access Interface - IDataAccess.cs

Controls all access to the database used within the EDDIS system. Can be implemented using almost any database package (including ODBC, OleDb, and SQL).

2.11.1 pushStart

Purpose: Create a new empty access database and initialize it (for example, create an empty table).

Prototype: `public bool pushStart()`

Inputs: None.

Outputs: Returns `true` if the method succeeded.

Restrictions: None.

Called by: `Logger.Receive()`

Calls: None.

Algorithm: –

- Initialize the database, ready for insertion.
- Algorithm is specific to the database package used.

2.11.2 push

Purpose: To insert a PDU and the current timestamp into the proper position in the database.

Prototype: `public void push( Time time, ref PDU pdu)`

Inputs: Time object specifying the timestamp for the data, and a reference to a `PDU` object to insert.

Outputs: None.

Restrictions: Assumes the PDU object is initialized correctly.

Called by: `Logger.Receive()`

Calls: None.

Algorithm: –

- Inserts the data into the database.
- Algorithm is specific to the database package used.

2.11.3 pushEnd

Purpose: Close the connection to the database once insertion is complete.

Prototype: `public void pushEnd()`

Inputs: None.

Outputs: None.

Restriction: None.

Called by: `Logger.Receive()`

Calls: None.

Algorithm: –

- Shuts down the database after insertion is complete.
- Algorithm is specific to the database package used.

2.11.4 pullStart

Purpose: Connect to an existing database and prepare to read data from it.

Prototype: `public bool pullStart( Time time )`

Inputs: The timestamp at which to start reading data from.

Outputs: Returns `true` if the method succeeded (and there was data to read).

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Skip()`

Calls: None.

Algorithm: –

- Initialize the database, ready for selection.
- Read the first entry from the database (if there is one).
- Return `true` if there was an entry to read, and the method succeeded. Otherwise return `false`.
- Algorithm is specific to the database package used.

2.11.5 pull

Purpose: Read an entry from the database and convert it to a PDU.

Prototype: `public bool pull( ref Time time, out PDU pdu)`

Inputs: Timestamp of PDU, and PDU filled with data.

Outputs: Returns `true` if successful and there is another entry.

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Skip()`

Calls: None.

Algorithm: –

- Read the data from the database, assuming there is some.
- Fill the `Time` and `PDU` objects given as parameters.
- Return `true` if the method succeeded and this was not the last entry. Otherwise return `false`.
- Algorithm is specific to the database package used.

2.11.6 pullEnd

Purpose: Shut down the database once the selections are done.

Prototype: `public void pullEnd()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Skip()`

Calls: None.

Algorithm: –

- Shut down the database, closing any connections necessary.
- Algorithm is specific to the database package used.

2.11.7 New

Purpose: To initialize a new database.

Prototype: `public bool New()`

Inputs: None.

Outputs: `true` if clearing the database was successful, `false` otherwise.

Restrictions: None.

Called by: `Logger.New()`

Calls: None.

Algorithm: –

- Restore default values of settings relating to file and database.
- Algorithm is not necessarily database specific.

2.11.8 Save

Purpose: To save a database with a given name.

Prototype: `public bool Save( string fullname )`

Inputs: Full path name of file to save this database as.

Outputs: `true` if successful, `false` otherwise. Also outputs (to the filesystem) the file to be saved.

Restrictions: Filename should be a valid Windows path and file name combination.

Called by: `Logger.Save()`.

Calls: None.

Algorithm: A suggested algorithm is: –

- Copy current database to new database name.
- Delete old database if it was previously 'temporary' (that is, if the user didn't previously create it and was created by EDDIS in a temporary directory with a temporary name).

2.11.9 Load

Purpose: To load an existing database and ready it for replay.

Prototype: `public bool Load( string fullname )`

Inputs: Full path name of file (database) to load.

Outputs: `true` if successful, `false` otherwise.

Restrictions: File must exist.

Called by: `Logger.Load()`.

Calls: –

- `Connect()`
- `Close()`

Algorithm: –

- Find file.
- Connect to database file.
- Find the length of the exercise (last row's timestamp).
- Close the database file.
- Set up any settings relating to file and database.

2.11.10 DatabaseException class

This class is used as an Exception in the database. For example, if a method generates an error, it can throw a DatabaseException object.

2.11.10.1 DatabaseException

Purpose: Overloaded. To contain information about an exception thrown in the database. Constructor for class.

Prototype: –

- `public DatabaseException()`
- `public DatabaseException( string msg )`

Inputs: A string message to save in the exception (can be read out later).

Outputs: None.

Restrictions: None.

Called by: Any `IDataAccess` method.

Calls: None.

Algorithm: Simply sets an internal message `string` to either the string specified, or a generic string such as:

```
"EDDISsys.DatabaseException"
```

2.11.10.2 ToString

Purpose: To return a string representation of the exception.

Prototype: `public override string ToString()`

Inputs: None.

Outputs: Returns the string message generated from the constructor.

Restrictions: None.

Called by: Any error handling routine.

Calls: None.

Algorithm: Should return the string created by the constructor.

2.12 The Timer class – Timer.cs

The Timer class is used to keep track of time in the logger. It reports the time since starting a playback/recording and takes into account when a playback pauses or skips.

Timer defines the following public accessor properties:

Speed Sets the speed of the timer object (1.0 for realtime). Write-only.

Current Gets the current time of the timer object, taking into account the fact that the timer may have been paused or skipped. Read-only.

2.12.1 Timer

Purpose: Overloaded. Initializes any variables or objects used by the Timer class. Constructor for class.

Prototype: –

- `public Timer()`
- `public Timer( Time offset )`

Inputs: An offset `Time` object to add to the final time reported.

Outputs: None.

Restrictions: Assumes the time object given is non-null.

Called by: `Logger.reset()`

Calls: `init()`

Algorithm: Constructor with no parameters calls `init()` with a `new Timer()` object, otherwise passes `offset` to `init()`.

2.12.2 init

Purpose: Initialize any variables or object used by the class.

Prototype: `public void init( Time offset )`

Inputs: An offset `Time` object to add to the final time reported.

Outputs: None.

Restrictions: Assumes the time object given is non-null.

Called by:

`Timer()`

Calls: None.

Algorithm: Set the current internal time to the offset given.

2.12.3 Start

Purpose: Overloaded. Starts the timer.

Prototype: –

- `public void Start()`
- `public void Start( float spd )`

Inputs: The speed at which the timer will be running.

Outputs: None.

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Receive()`

Calls: `Speed`.

Algorithm: –

- Given no parameters, should call itself with `1.0f` as a parameter.
- Sets the `Speed` accessor to the speed given.

2.12.4 Stop

Purpose: To stop the timer.

Prototype: `public void Stop()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: –

- `Logger.Send()`
- `Logger.Receive()`

Calls: None.

2.12.5 Pause

Purpose: To pause the timer.

Prototype: `public void Pause()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `Logger.Wait()`

Calls: None.

2.12.6 Resume

Purpose: To resume a paused timer.

Prototype: `public void Resume()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `Logger.Wait()`

Calls: None.

2.12.7 Skip

Purpose: To skip the timer forward or backward in time.

Prototype: `public void Skip( double amount )`

Inputs: The amount of time (in seconds or fractions of a second) to skip the timer by. Negative values means skip bacwards, positive means forwards.

Outputs: None.

Restrictions: None.

Called by: `Logger.Skip()`

Calls: None.

2.13 The Error class – Error.cs

The Error class is used to report and log all errors that occur in the EDDIS system. All methods and properties of the Error class should be declared static, so they can be accessed anywhere within the system.

The Error class defines the following public accessor properties:

Message Gets the last message generated within the Error class. Resets the internal message string to empty. Read-only.

Any Gets if there are any messages currently waiting to be handled. Returns `true` if there are, `false` otherwise. Read-only.

2.13.1 Init

Purpose: Overloaded. To initialize the error reporting class.

Prototype: –

1. `public static void Init( Form main )`
2. `public static void Init( Form main, string log )`
3. `public static void Init( Form main, string log, bool debug )`

Inputs: –

1. Takes a Windows form for displaying error dialogs onto. This should usually be the `GUImain` object instantiated in `MainEntry`.
2. As above, but also takes a filename to save log information to.
3. As above, but also takes `true` if debugging information should be shown (extra data about errors).

Outputs: None.

Restrictions: Log-file must be a valid filename.

Called by: `MainEntry.Main()`

Calls: `setGUI()`

Algorithm: –

- First two definitions calls the third definition with default values (log-file and debugging info should be set to settings stored in `Settings`).
- Sets all internal variables, and sets the destination form to `main`.

2.13.2 setGUI

Purpose: To set the User Interface that the Error will display messages to.

Prototype: `public static void setGUI( Form main )`

Inputs: A Windows form to display error dialogs to.

Outputs: None.

Restrictions: None.

Called by: –

- `MainEntry.Main()`
- `Error.Init()`

Calls: None.

2.13.3 Post

Purpose: Overloaded. To post a message to the Error class.

Prototype: –

1. `public static void Post( string m, Exception ex )`
2. `public static void Post( string m, string debug )`
3. `public static void Post( string m )`

Inputs: –

1. A string message and the exception that caused this post to occur.
2. A string message and a string debugging message.
3. A string message.

Outputs: None.

Restrictions: None.

Called by: Any method in the system that wishes to report an error.

Calls: `Log()`

Algorithm: –

- First two definitions call the third definition with the string representation of the exception or the debug string concatenated to the message, if debugging info is on. Otherwise calls the third definition with just the message.
- Saves the message and logs it to the log-file.

2.13.4 Clear

Purpose: To clear the internal message string.

Prototype: `public static void Clear()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `Handle()`

Calls: None.

2.13.5 Log

Purpose: Overloaded. To log a message to the log-file.

Prototype: –

1. `public static void Log( string message )`
2. `public static void Log( string message, bool only_debug )`

Inputs: String message and optional variable specifying whether the message should only be logged if debugging is on.

Outputs: None.

Restrictions: None.

Called by: `Post` and any method in the system that wishes to log an error.

Calls: None.

Algorithm: –

- First definition should call second with 'false' as the second parameter.
- Opens the log-file and appends the message onto the end of it in the form:

  `hh:mm:ss : message`

  where `hh:mm:ss` is the time at which the log entry was saved, and `message` is the log message.

2.13.6 Handle

Purpose: To handle the error (if there is one) and display a dialog to the User Interface defined in the error class.

Prototype: `public static bool Handle()`

Inputs: None.

Outputs: Returns `true` if there were no errors to handle.

Restrictions: None.

Called by: Any method in the system that is ready to display an error dialog to the interface.

Calls: None.

Algorithm: –

- Executes the `MessageBox` method into the form stored in the class.

2.14 The ConfigScripting class – ConfigScripting.cs

This class is used to parse and obtain 'key = value' pairs (or 'key value' pairs) from a script file. The script file is line-based; that is, each line is read anew and assumed to be a new definition. Because of this, the normal tokenizer approach is not recommended, as it would complicate the parsing process too much.

A line in the script file has the following layout:

```
key value  # comment
key=value  # comment
# comment
```

That is, a key and value pair can be separated by either a single equals sign, `=`, or a single whitespace. A comment is anything after and including the pounds (`#`) sign.

The only data type which would have a different representation to that of the standard (ie `int`s and `double`s are obvious, simply use `ToString`) is the `bool` type. A key which has a `bool` value should use one of the following definitions:

	Def 1	Def 2
true	yes	on
false	no	off

Key value pairs should be stored in an `ArrayList` of `ConfigPair` objects.

2.14.1 ConfigScripting

Purpose: To initialize the object. Constructor for class.

Prototype: `public ConfigScripting()`

Inputs: None.

Outputs: None.

Restrictions: None.

Called by: `MainEntry.Main()`

Calls: None.

Algorithm: Simply clear the list of key value pairs in the class.

2.14.2 GetKey

Purpose: Overloaded. Given a key string, gets a value associated with it.

Prototype: –

1. `public bool GetKey( string key, ref string val )`
2. `public bool GetKey( string key, ref int val )`
3. `public bool GetKey( string key, ref long val )`
4. `public bool GetKey( string key, ref float val )`
5. `public bool GetKey( string key, ref double val )`
6. `public bool GetKey( string key, ref bool val )`
7. An option `int num` parameter is also available for each method.

Inputs: A string key to search for, a reference to a variable to store the value in (overloaded to accept many different types), and an option parameter specifying the entry number to search for (in cases where multiple definitions are defined).

Outputs: Returns `true` if the key was found.

Restrictions: None.

Called by: `MainEntry.Main()`

Calls: None.

Algorithm: Each method should follow the rules identified in section 2.14 to parse the value.

2.14.3 Parse

Purpose: To load/parse a configuration script and load the data contained into `ConfigPair` objects.

Prototype: `public bool Parse( string filename )`

Inputs: A filename to parse. An error will be logged if the file does not exist, or an error occurs during parsing.

Outputs: Returns `true` if successful (no errors).

Restrictions: None.

Called by: `MainEntry.Main()`

Calls: None.

Algorithm: –

- Open the file using a `StreamReader` object.
- Parse each line individually, splitting the line at a comment character (`#`).
- Split the line again at a whitespace or `=` character.
- Instantiate a new `ConfigPair` object with the two strings obtained, and add it to the `ArrayList`.
- Continue doing this until EOF.
- Close the file and return `true`

2.15 The ConfigPair class – ConfigPair.cs

The ConfigPair class is used to store key/value information parsed from a configuration script in the `ConfigScripting` class.

The class has two public properties: `key` and `val`. Their type is `string`.

2.15.1 ConfigPair

Purpose: Overloaded. To initialize a configuration key/value pair. Constructor for class.

Prototype: –

- `public ConfigPair()`
- `public ConfigPair( string k, string v )`

Inputs: The key and value strings to store.

Outputs: None.

Restrictions: None.

Called by: `ConfigScripting.Parse()`

Calls: None.

Algorithm: –

- Constructor with no parameters sets the values to empty strings.
- Otherwise constructor sets the values to those given as parameters.

2.16 The PDUHeader class – PDUHeader.cs

This class is used to store data contained in a DIS PDU header. The following public properties are defined:

`byte version`	Specifies version of the protocol used in PDU
`byte id`	Specifies the exercise to which PDU pertains
`byte type`	Type of PDU
`byte family`	Family of protocols to which PDU type belongs
`uint timestamp`	Time that PDU was generated
`ushort length`	Length of PDU in byte (including header)
`ushort padding`	Unused

For more detailed information about a PDU header, refer to section 3.

There are also the following public accessors defined in PDUHeader:

Size Static `int` that returns the size of a DIS PDU header. Always should be 12 (bytes). Read-only.

Length Gets the size of the PDU data, not including the PDU header size. Should be `length - Size`. Read-only.

Since this object is a storage class, the *Called by* field in the method/property description will be "Many methods in this system". It is not necessary to know what calls these PDUHeader methods/properties, as it is independent from the system.

2.16.1 PDUHeader

Purpose: Overloaded. Initializes PDUHeader object.

Prototype: –

- `public PDUHeader()`
- `public PDUHeader( byte[] data )`

Inputs: An array of bytes. Maps the first `Size` bytes to the PDU header properties.

Outputs: None.

Restrictions: None.

Called by: Usually called by methods in `PDU`, but can be called by other methods which need to create a header, for example `NetAccess.getPDU()`.

Calls: None.

2.16.2 ToArray

Purpose: To convert the header into a byte array.

Prototype: `public byte[] ToArray()`

Inputs: None.

Outputs: A byte array representation of this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Basically performs the opposite function to the constructor. 'Unmaps' the object back into a byte array.

2.17 The PDU class – PDU.cs

This class is used to store data contained in a DIS PDU. It includes the PDU header, which is stored in a `PDUHeader` object. The following public properties are defined:

`PDUHeader header`	The object containing the PDU header information
`byte[] data`	A byte array containing the PDU data (not including the PDU header)

Since this object is a storage class, the *Called by* field in the method/property description will be "Many methods in this system". It is not necessary to know what calls these PDU methods/properties, as it is independent from the system.

2.17.1 PDU

Purpose: Overloaded. To initialize a new PDU object.

Prototype: –

1. `public PDU( PDUHeader hdr, byte[] rest )`
2. `public PDU( byte[] input )`

Inputs: –

1. Takes a `PDUHeader` object and a byte array containing the PDU data (not including PDU header).
2. Takes a byte array containing the PDU data and PDU header.

Outputs: None.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Initialize the byte data by calling `PDUHeader` constructor, or simply assigning `data` and `header`.

2.17.2 ToArray_nohdr

Purpose: To convert the PDU to a byte array, not including the PDU header.

Prototype: `public byte[] ToArray_nohdr()`

Inputs: None.

Outputs: A byte array representation of this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Copy the `data` byte array to a new byte array and return it.

2.17.3 ToArray

Purpose: To convert the PDU to a byte array, including the PDU header.

Prototype: `public byte[] ToArray()`

Inputs: None.

Outputs: A byte array representation of this object.

Restrictions: None.

Called by: Many methods in the system.

Calls: None.

Algorithm: –

- Convert the PDU header to a byte array.
- Create a new byte array with the PDU header and `data` concatenated together.
- Return this new byte array.

Chapter 3

Database Design

This section describes the basic design of the database that will store the event data derived from the PDUs, which is retreived from the network.

3.1 Table Design

Table 3.1 lists the structure of an individual entry (row) in the database. It shows the field name of each column, the MS Access datatype, and the equivalent C# type.

Field	MS Access datatype	C# Type
`eddisID`	Double	`double`
`eddisTimestamp`	Double	`double`
`pduVersion`	Byte	`byte`
`pduID`	Byte	`byte`
`pduType`	Byte	`byte`
`pduFamily`	Byte	`byte`
`pduTimeStamp`	Double	`Int32` / `uint`
`pduLength`	Long	`Int16` / `ushort`
`pduPadding`	Long	`Int16` / `ushort`
`pudData`	Binary	`byte[]`

Table 3.1: Database Row structure

3.1.1 Database Schema

The database schema is:

```
table(
    eddisID,
    eddisTimestamp,
    pduVersion,
    pduID,
    pduType,
```

```
    pduFamily,
    pduTimeStamp,
    pduLength,
    pduPadding,
    pduData
)
```

`eddisID` is the primary key for the table.

3.2 Field Description

Following is a description of each field in the database:

EDDIS ID (primary): A unique number for each row (entry) in the database. It shall be the primary key, and the order defined as the order the PDU was received in.

EDDIS timestamp: The internal EDDIS time that the PDU was received.

Protocol Version: Specifies the DIS version that the PDU originated from.

Exercise ID: An independent value determining the current DIS exercise being performed.

PDU Type: The PDU type. See section 3.3 for a list of PDU types.

Protocol family: The family the PDU type belongs to. See section 3.3 for a list of protocol families.

Timestamp: Specifies the time at which the PDU data is valid.

Length: The number of bytes occupied by the PDU, including the header.

Reserved: An unused padding field.

Data: Data for the received PDU. See IEEE 1278.1 and 1278.1a for more information.

3.3 PDU Families and Types

There are 67 PDU types which are organized into 12 protocol families. This is an excerpt from IEEE 1278.1.

1. Entity Information/Interaction
   (a) Entity State PDU
   (b) Collision PDU
   (c) Collision-Elastic PDU
   (d) Entity State Update PDU
2. Warfare

   (a) Fire PDU
   (b) Detonation PDU

3. Logistics

   (a) Service Request PDU
   (b) Resupply Offer PDU
   (c) Resupply Received PDU
   (d) Resupply Cancel PDU
   (e) Repair Complete PDU
   (f) Repair Response PDU

4. Simulation Management

   (a) Start/Resume PDU
   (b) Stop/Freeze PDU
   (c) Acknowledge PDU
   (d) Action Request PDU
   (e) Action Response PDU
   (f) Data Query PDU
   (g) Set Data PDU
   (h) Data PDU
   (i) Event Report PDU
   (j) Comment PDU
   (k) Create Entity PDU
   (l) Remove Entity PDU

5. Distributed Emission Regeneration

   (a) Electromagnetic Emission PDU
   (b) Designator PDU
   (c) Underwater Acoustic(UA) PDU
   (d) IFF/ATC/NAVAIDS PDU
   (e) Supplemental Emission/Entity State PDU

6. Radio Communications

   (a) Transmitter PDU
   (b) Signal PDU
   (c) Receiver PDU
   (d) Intercom Signal PDU
   (e) Intercom Control PDU

7. Entity Management

   (a) Aggregate State PDU
   (b) IsGroupOf PDU
   (c) Transfer Control Request PDU
   (d) IsPartOf PDU

8. Minefield

   (a) Minefield State PDU
   (b) Minefield Query PDU
   (c) Minefield Data PDU
   (d) Minefield Response Negative Acknowledgment PDU

9. Synthetic Environment

   (a) Environment Process PDU
   (b) Gridded Data PDU
   (c) Point Object State PDU
   (d) Linear Object State PDU
   (e) Areal Object State PDU

10. Simulation Management with Reliability

   (a) Create-Entity-R PDU
   (b) Remove-Entity-R PDU
   (c) Start/Resume-R PDU
   (d) Stop/Freeze-R PDU
   (e) Acknowledgment-R PDU
   (f) Action Request-R PDU
   (g) Action Response-R PDU
   (h) Data Query-R PDU
   (i) Set Data-R PDU
   (j) Data-R PDU
   (k) Event Report-R PDU
   (l) Comment-R Message PDU
   (m) Record Query-R PDU
   (n) Set Record-R PDU
   (o) Record-R PDU

11. Live Entity

   (a) Time Space Position Information PDU
   (b) Appearance PDU
   (c) Articulated Parts PDU
   (d) LE Fire PDU
   (e) LE Detonation PDU

Chapter 4

User Interface

The User Interface is a crucial aspect of the system in terms of both what the client wants and needs. For this reason there is an overview of the User Interface in the Software Requirements Specification (SRS). This section will detail all aspects of the UI and its design, and thus will be more extensive than the SRS section. These documents also have different target audiences and aims and thus different User Interface sections are presented.

The Graphical User Interface (GUI), for the purpose of this description, has been broken up into three main sections. These are:

- The Menus
- The Toolbar
- The Event List and Display

4.1 The Menus

This section of the GUI is simply the menu system our program will be using. There are a number of menu items, as detailed below, which the user will have access to. For simplicity, we have grouped similar items under the one menu in a similar way to many other Windows applications. This will increase the intuitiveness of the GUI and allow the the user to find the desired item quickly and easily.

A detailed description of the menu items and their functions follows:

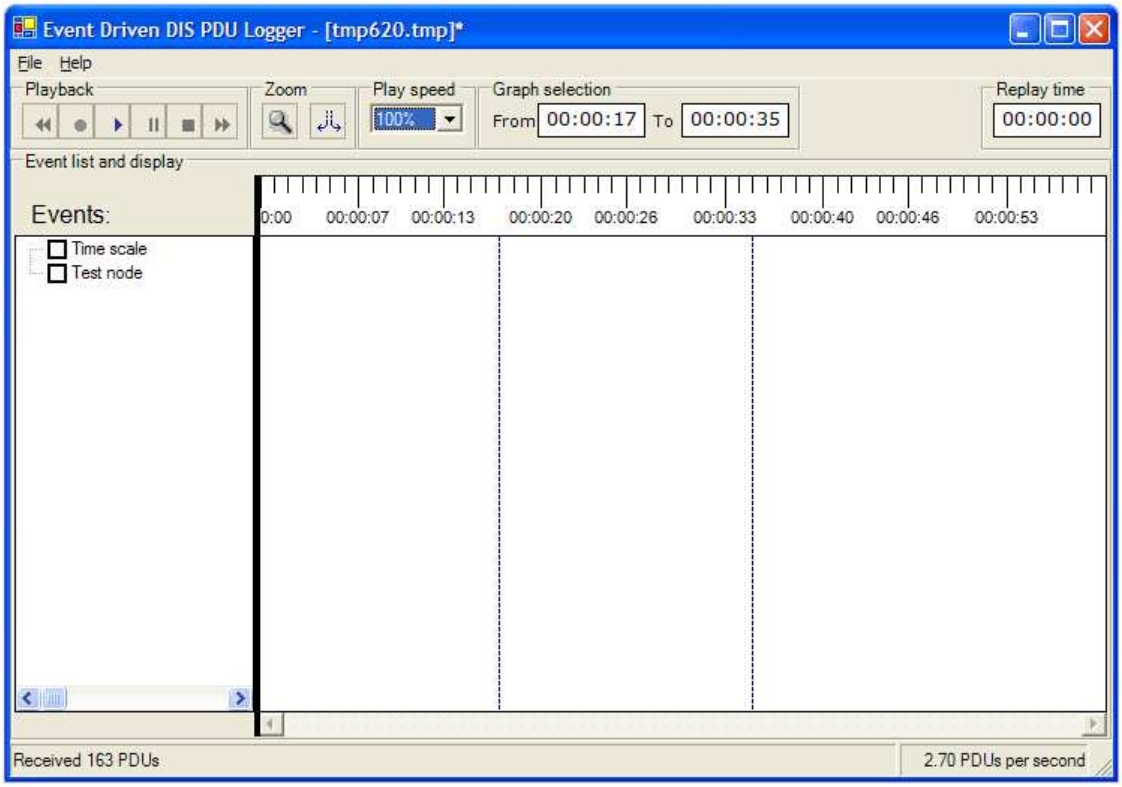

Figure 4.1: The User Interface of the EDDIS System

Menu Item	Description
File-New	Clears all database-specific data. Prompts the user if data has not been saved.
File-Open	Opens a standard Windows dialog box to enable the user to open a saved session. A file type filter (EDDIS Data File *.edf) will assist the user in finding the required file quickly.
File-Save	Saves the current session. Opens a standard Windows dialog box to enable the user to save the current session.
File-Exit	Opens a dialog box prompting the user to save the current session and confirm the exit request.
Help-User Manual	Opens the User Manual (HTML file)
Help-About	Displays a dialog box detailing the author's information and version number.

4.2 The Toolbar

This section of the GUI contains various buttons, combo boxes and text displays which enable control of the program and give the user information.

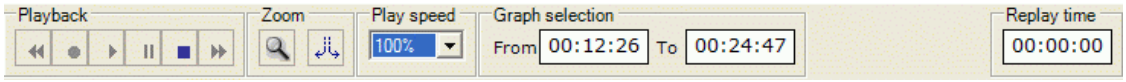

Figure 4.2: The Toolbar

4.2.1 Playback Controls

The playback and recording controls use standard VCR-like icons that are easy to identify. Whilst the program is idle (not recording or playing back a session), the *Back*, *Pause*, *Stop* and *Forward* buttons will be disabled. Once playback or recording commences, the *Record* and *Play* buttons will become disabled. A summary of the different control's functions is given below:

Item	Description
Back	Skips back through the playback of the simulation
Pause	Pauses the playback of the simulation
Play	Begins playback of the simulation
Forward	Skips forward through the playback of the simulation
Stop	Stops the playback/recording of the simulation
Record	Begins recording of the simulation

4.2.2 Zoom Controls

The zoom controls are used to to change the Graph Display as the user desires. After the user selects the *From* and *To* times (as described in below), the *Zoom Region* button is pressed and the display will zoom to the desired level. To get back to the default view, the user can press the *Zoom Full* button.

To select a specific area of the graph for zooming purposes, the user will follow these steps:

- *Left Click* on the desired start time, which will be displayed in the *From* display on the Toolbar.
- *Right Click* on the desired end time, which will be displayed in the *To* display on the Toolbar.
- Click the *Zoom Region* button on the toolbar, which will cause the Graph Display to zoom as appropriate.
- The user can then either select a new region to zoom, or can restore the default view by clicking the *Zoom Full* button on the Toolbar.

Item	Description
Zoom Region	Zooms to the desired level using the *From* and *To* times.
Zoom Full	Restores the Graph Display to the original default level.

4.2.3 Play Speed

This control is a drop down box which can be used to set the desired Playback speed. The user can choose one of the predefined rates as shown in the drop down box. Please note that the Client has

requested that the original option of entering the desired rate manually (up to a maximum rate of 10.00) be removed. Selectable speeds are read in from a configuration script, but the default are:

- 25%
- 50%
- 100%
- 200%

4.2.4 Graph Selection and Replay Time Displays

The remainder of the Toolbar controls are non-interactive displays which present different times to the user:

Item	Description
From Time	Displays the current *From* time as selected by the user.
To Time	Displays the current *To* time as selected by the user.
Replay Time	Displays the current *Replay* time, relative to the beginning of the playback or recording.

4.3 The Event List and Display

This section of the GUI displays the Events on a scrolling graph and allows the user to restrict which Events are to be displayed.

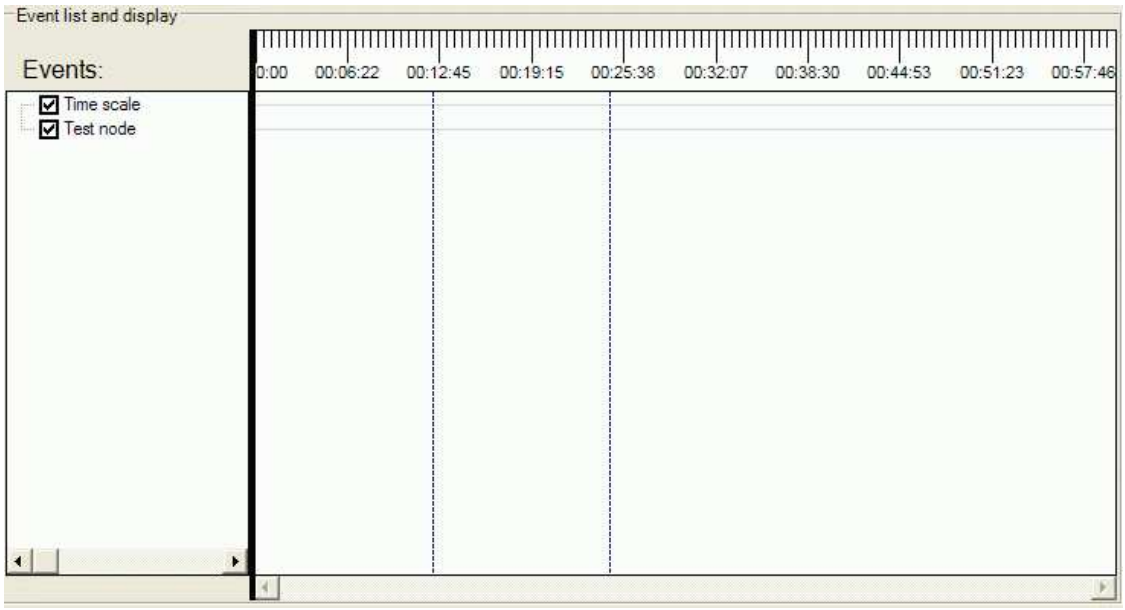

Figure 4.3: The Event List and Graph Display

4.3.1 The Event List

All possible events for display (as defined in the appropriate *config* file) are listed here in a CheckBox list. The list is horizontally opposite the Graph Display, as shown in the diagram below. The vertical position of a specific Event corresponds to the vertical position of the Event on the Graph Display. The user can select which events to display by checking the appropriate CheckBoxes.

4.3.2 The Graph Display

This is where the simulation is graphically represented after recording or loading. The x-axis is a time scale (as shown across the top of the display). The y-axis is the Events, as described in the previous section. To select a specific area of the graph for zooming purposes, the user will follow these steps:

- *Left Click* on the desired start time, which will be displayed in the *From* display on the Toolbar.
- *Right Click* on the desired end time, which will be displayed in the *To* display on the Toolbar.
- Click the *Zoom Region* button on the toolbar, which will cause the Graph Display to zoom as appropriate.
- The user can then either select a new region to zoom, or can restore the default view by clicking the *Zoom Full* button on the Toolbar.

4.3.3 Status Bar

A *Status Bar* below the Graph Display will give the user information on the current operation and notify of any errors or warnings.

Bibliography

[1] 433-340 Software Engineering Project Manual, 2002

[2] Schach, Stephen R, "Classical And Object-Oriented Software Engineering with UML and Java. 4th Edition", 1999, WCB/McGraw-Hill

[3] Vliet, Hans Van, "Software Engineering Principles and Practice", 2001, Wiley

[4] Team A New Hope Software Architectural Design Document (SADD), 2001

[5] Team B Software Architectural Design Document (SADD), 2001

[6] Team O Software Quality Assurance Plan (SQAP), 2002

[7] Team O Software Requirements Specification (SRS), 2002

[8] Sparx Systems, "UML Tutorial", *http://www.sparxsystems.com.au/UML_Tutorial.htm*, 2002

[9] Kobryn, Cris, "Introduction to UML: Structural Modelling and Use Cases", *http://www.cs.mu.oz.au/341/uml-intro.pdf*, 2000

[10] *http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/6*

[11] Interfaced Systems International, "User Interface Design and Usability", *http://www.isii.com/ui_design.html*, 1999

[12] "IEEE Standard for Distributed Interactive Simulation – Application Protocols", 1995

[13] Microsoft, ".NET Framework Class Library, Object.MemberwiseClone Method", *ms-help://MS.VSCC/MS.MSDNVS/cpref/html/ frlrfSystemObjectClassMemberwiseCloneTopic.htm*, 2002.

[14] Priestley, Mark, "Practical Object-Oriented Design with UML", 2000, McGraw-Hill.

Glossary

DIS *Distributed Interactive Simulation* The protocol by which a simulation transmits data. DIS uses the IEEE 1278.1 standard.

DSTO *Department of Science and Technology* The DSTO is the Client Organisation.

event A change within the simulation, eg an Enemy Hit. Usually signified by a DIS PDU being sent.

Exercise The overall 'war-game', usually played by many countries and forces around the world, involving hundreds of simulators and sometimes thousands of PDUs per second.

GUI *Graphical User Interface* The interface through which the user interacts with all aspects of the program.

IEEE *Institute of Electrical and Electronics Engineers* Professional organization whose activities include the development of communications and network standards.

ISO *International Organization for Standardization* International organization that is responsible for a wide range of standards, including those relevant to networking.

ITU-T *International Telecommunication Union Telecommunication Standardization Sector* International body that develops worldwide standards for telecommunications technologies.

OSI *Open System Interconnection* International standardization program created by ISO and ITU-T to develop standards for data networking that facilitate multivendor equipment interoperability.

PDU *Protocol Data Unit* The OSI term for a network packet.

RFC *Request For Comments* Document series used as the primary means for communicating information about the Internet.

UDP *User Datagram Protocol* UDP is a simple protocol that exchanges datagrams without acknowledgments or guaranteed delivery, requiring that error processing and retransmission be handled by other protocols. UDP is defined in RFC 768. UDP is limited to a packet size of 65507 bytes.