

Building a Complete GEDAE Audio Application Containing an Extensive Graphical User Interface

Douglas Smith
Air Force Research Laboratory
AFRL/IFEC
Rome, NY 13441
(315) 330-3474

Douglas.Smith@afrl.af.mil

Daniel Curtacci
ACS Defense, Inc.
AFRL/IFEC
Rome, NY 13441
(315) 330-4026

Daniel.Curtacci@afrl.af.mil

ABSTRACT

GEDAE is primarily designed for the optimization of embedded systems development on real-time COTS hardware. Development of full-scale Graphical User Interfaces is a task typically accomplished using one of many available GUI development packages. The GEDAE distribution includes a basic framework for building applications using X widgets, but an extensive toolbox is not readily available. Development of an audio editing application was pursued using GEDAE. This was desirable to allow novel speech processing research algorithms to be easily integrated into the editor application for testing. Development of this application took four solid months, from design to first prototype, followed by one month of testing and modification. This paper will describe the design of the application, some of the challenges that needed to be overcome, and the issues associated with synchronizing parameters and processing challenges with the data flow. The presentation will also include a demonstration of the application.

1. DESIGN

Figure 1 shows a screenshot of the Audio Editing Application. The main canvas is a display of the time-domain waveform, complete with time and amplitude tick marks on the axes. Above the audio waveform is a small pane showing audio segments that

have been marked. Each of these segments has a corresponding label in the text box on the left. Push-button controls are provided for playing and stopping the audio. There are also controls to zoom in and out on the audio waveform. Pull-down menus at the top allow the user to perform different operations on the audio file. Access to four separate audio buffers is also provided

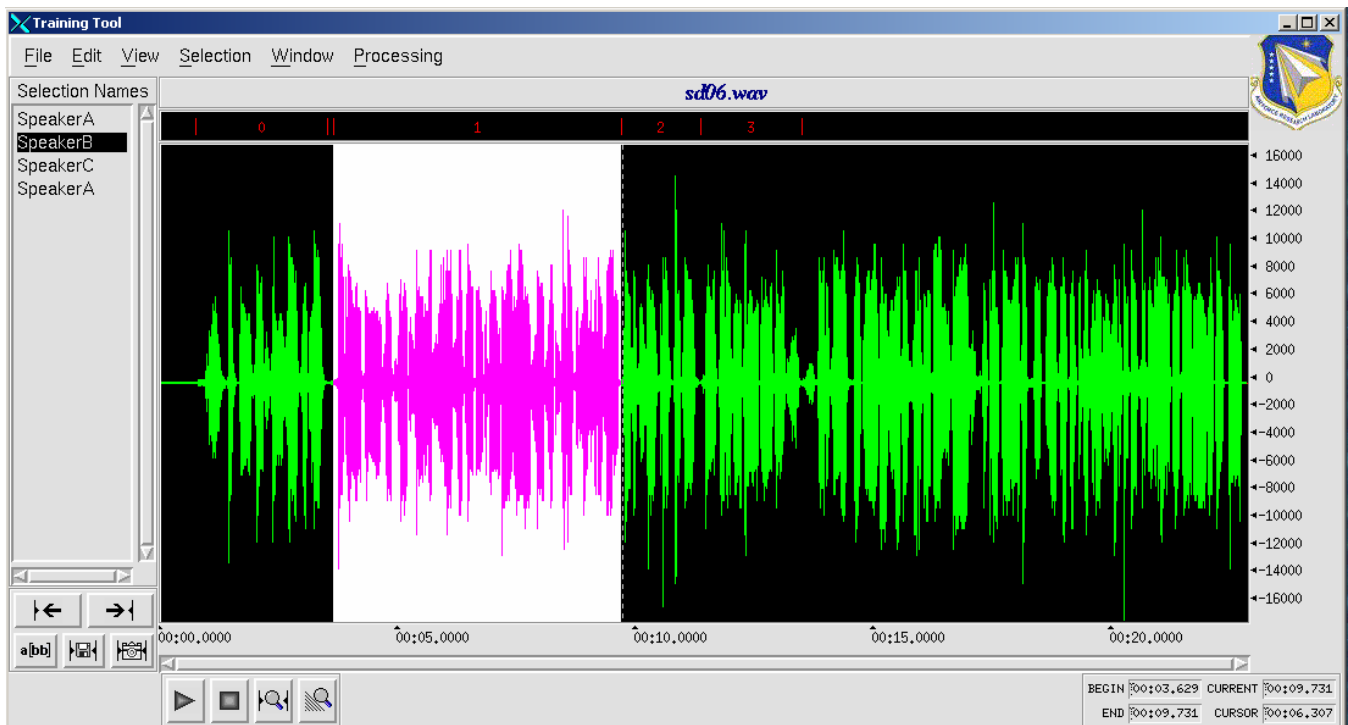


Figure 1: Complete Audio Editing Application

through these menus.

Figure 2 shows the main flowgraph for the audio editor. Looking at the number of box inputs, outputs, and connections between boxes demonstrates the large number of parameters that are necessary for the operation of the application. Also note that a significant number of these connections are in a feedback fashion. Viewing the flattened graph takes up multiple screen widths and heights and is extremely difficult to see anything but spaghetti connections. Each flowgraph within the top level graph is very specialized. There are 1,758 boxes in this graph, comprised of 183 subgraphs, 1,527 primitives, and 48 route boxes.

- *fgData* – Contains all of the “parameter” data for the entire application.
- *fgLayout* – Contains the design, or layout, of the graphical components of the application. Changes such as adding or removing a new widget, or rearranging the placement and order of widgets could be done quickly and easily without affecting the rest of the graph.
- *fgControls* – All user controls were kept here (menus, buttons, keyboard/mouse events). Each output from this box was a trigger telling the rest of the graph how and when to respond.
- *fgGraphics* – All of the graphics widgets are placed in this

graph. It also contains all the functions for drawing and updating each widget.

- *fgAudioBuffer* – This graph handles the buffers for the audio applications. These buffers contain the actual audio data. Functions for manipulation of audio data (copy, cut, paste, delete, audio playback) are located here.

2. CHALLENGES

During the development of this project, many problems arose. These challenges were a direct result of the sometimes overwhelming number of parameters involved in developing this application. The use of trigger boxes to shuffle around parameters adds a significant amount of clutter to graphs. Although most of the parameter clutter was hidden at the highest levels, it still makes for a difficult time building and debugging graphs. Synchronizing pushed parameters also was a significant challenge due to the large number of parameters and many application states possible.

3. SOLUTIONS

To overcome the large number of connections at the toplevel graph for the audio editor application, the use of typedefs was very helpful. Figure 2 shows the completed toplevel graph for the audio application. There are a considerable number of

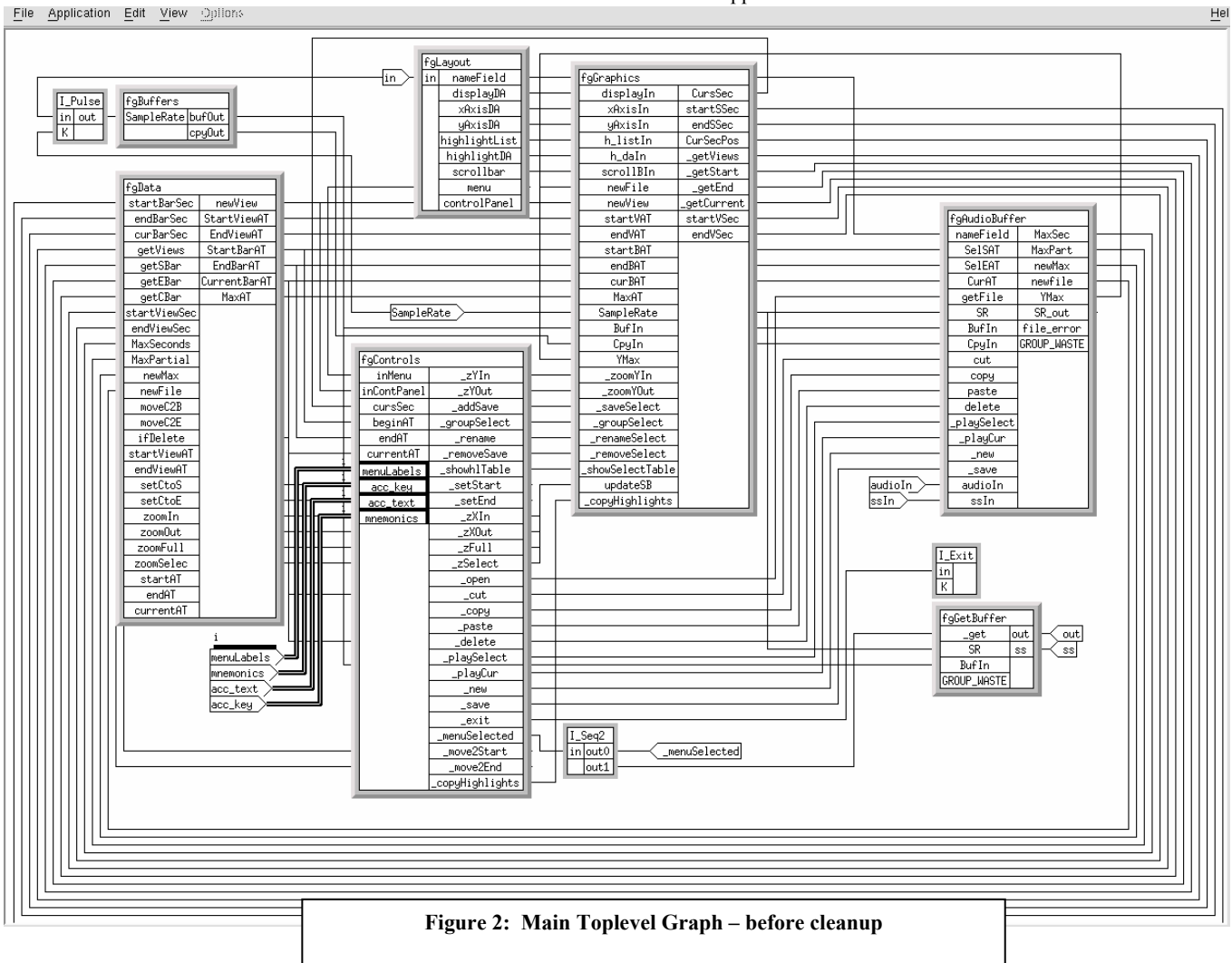


Figure 2: Main Toplevel Graph – before cleanup

connections that wrap around from one subgraph to another. Also, there are a few cases where the visible connections are obscured and/or ambiguous. After the application was mostly complete, typedef boxes were used to help simplify the graph. The clarity of the graph was greatly improved as seen in Figure 3. The number of connections was reduced significantly. Additionally, similar data that needed to move from and to the same subgraphs were grouped together into one typedef box.

The other significant challenge faced during this project was parameter synchronization. There were two issues that needed to be overcome: Firstly, deal with the sequencing of pushed parameters to guarantee tasks are completed in the correct order. Secondly, overcome the problems of parameters being overwritten when multiple parameters are pushed to the same box.

A pipeline concept was implemented to overcome the synchronization problem. All parameters are set on the top level graph, and the rest of the boxes are set sequentially down to the lowest level graph in a cascade fashion.

Along the way, order specific parameters perform two tasks: first they send their parameter, and then they trigger any remaining parameters. This process ensures dataflow progresses in the

correct order.

Use of this pipeline technique resulted in instances where a large volume of parameters are pushed down the same pipeline very quickly. If these parameters are not handled and used in time, they would be overwritten and lost. The solution to this problem was to develop a queuing system for parameters. Queue buffers were placed inline before any box where heavy traffic could occur, allowing parameters to accumulate when necessary and be handled in turn. With this construct, triggered parameters can be pushed without loss of data.

4. CONCLUSION

It is relatively easy to use GEDAE for the development of simple tools and utilities, as well as small graphical demos. After completing this project it is apparent that GEDAE can also be used for large scale GUI applications. At the time of this writing, the standard library boxes distributed with GEDAE cannot effectively handle a large scale application. Through the development of this effort at AFRL, these extensions to the trigger library have been implemented and can be fed back to improve the GEDAE product.

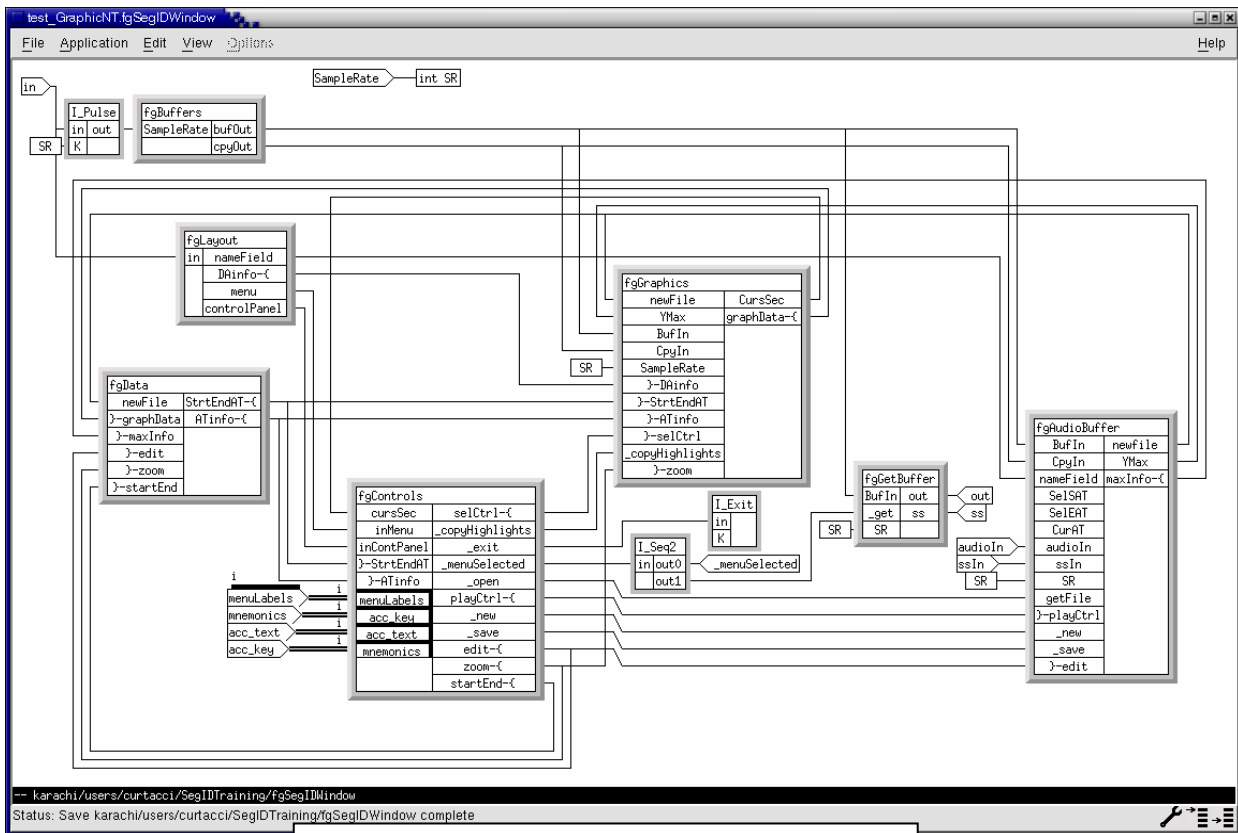


Figure 3: Main Toplevel Graph – using typedefs