Rosegarden Sequencer Design Document

A.J. Green

1. Introduction

The project outline for the sequencer was as follows:

"There is a recurring call for a music sequencer program running under UNIX. It should be based on a simple "piano-roll" format with the aim of generating MIDI files or MIDI real-time. An alternative output would be direct synthesis on either a SUN4 or SGI. The emphasis throughout should be on ease of use."

This document describes the design of the sequencer module of the Rosegarden music suite, developed by myself and Chris Cannam. The Rosegarden suite provides an integrated environment for the creation and manipulation of computer-based music. The suite consists primarily of two modules, the Sequencer, providing the ability to play, record and manipulate MIDI data, and the Editor, which is devoted to the production of traditional music scores, both interactively and from MIDI data captured by the sequencer. The tools are integrated via the INTERLOCK inter-process communication system, written by myself specifically for this project. A description of the INTERLOCK system is given in the document 'The INTERLOCK Tool Integration Framework', included as an appendix with this report.

1.1 The Structure of this Report

This report is intended not only as a description of the design of the sequencer, but also to serve as a reference source for anyone wishing to make use of the utility libraries developed as part of this project.

Section 2 of the report details the design of the user interface, and gives a brief outline of some of the problems encountered during its implementation.

Section 3 gives a brief overview of the internal representation used for MIDI data.

Section 4 discusses the problems inherent in attempting to implement MIDI recording and playback in a UNIX environment, with a description of the eventual solutions and a brief evaluation of how successful they were.

Section 5 is a reference guide for the MIDI library developed as part of this project. This is included to provide reference material for anyone wishing to make use of the library in their own applications.

Appendices A and B give some useful tables of MIDI information. Appendix C is the design document for the **INTERLOCK** framework, and again contains a reference guide suitable for anyone wishing to make use of this library.

1.2 Rosegarden Suite Design Philosophy

When myself and Chris first began sketching out ideas for our respective projects it became apparent that there were large areas of potential redundancy between the two projects, in terms of both code and desired functionality. Entering large pieces of music into the notation editor would be an onerous task without the ability to read MIDI data and transform to a score representation. Similarly, should the user wish to enter music directly into the sequencer (rather than recording from an external MIDI device) this task would be greatly simplified if the user could enter the music in traditional score notation, rather than forcing them to deal with MIDI events directly. It was also apparent that to produce applications of a usable quality would take more time than was available.

In an effort to reduce the amount of redundancy between the two applications, and to increase the utility of both, we decided to style the applications as modules in a combined tool suite, and use an integration medium to allow data to be transferred between the applications in a manner transparent to the end user. As a result of deciding on this design approach it then became much more important to keep a common look and feel between the user interfaces of the two applications.

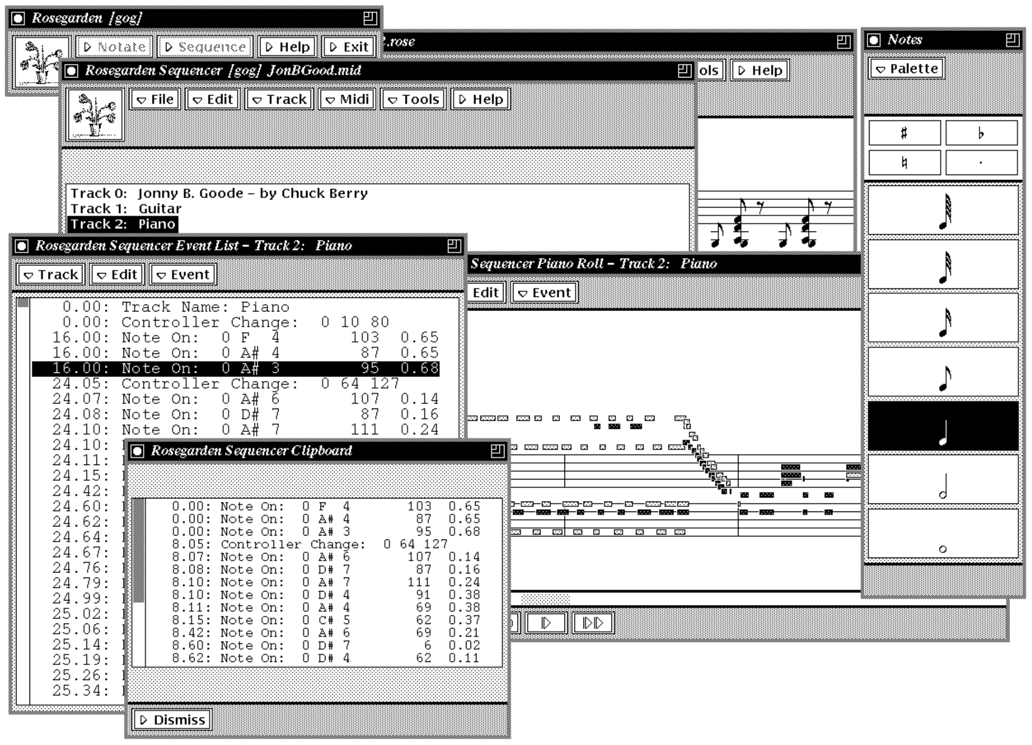
Having decided on this approach we apportioned work on the common areas in the follow manner,

1. I would implement the **INTERLOCK** system to allow inter-process communication.

2. Chris would provide a library of utility function that would enable easy creation of widgets (such as command buttons and menus) that would have then have a common style between the two applications.

3. To transfer data between the applications we would use the standard MIDI file format, writing the data to be transferred to a temporary file and passing the pathname of this file to the destination application. To this end I would make my MIDI file and event manipulation code available as a separate library. The MIDI track representation would make use of Chris' doubly-linked list library to prevent duplication of such generic code.

By taking this approach I believe that we have been able to produce two applications which are far more useful for the computer musician than would have been the case had they been developed separately, and also by avoiding unnecessary duplication of effort we have managed to make both our applications far more complete than they would otherwise have been given the limited timescale of the project.


Rosegarden [gog]
Notate
Sequence
Help
Exit
.rose
ols
Help
Rosegarden Sequencer [gog] JonBGood.mid
File
Edit
Track
Midi
Tools
Help
Notes
Palette
Rosegarden Sequencer Event List – Track 2: Piano
Track
Edit
Event
Sequencer Piano Roll – Track 2: Piano
Edit
Event
Rosegarden Sequencer Clipboard
Dismiss

```
Track 0:  Jonny B. Goode – by Chuck Berry
Track 1:  Guitar
Track 2:  Piano
```

```
 0.00: Track Name: Piano
 0.00: Controller Change:  0 10 80
16.00: Note On:  0 F  4       103  0.65
16.00: Note On:  0 A# 4        87  0.65
16.00: Note On:  0 A# 3        95  0.68
24.05: Controller Change:  0 64 127
24.07: Note On:  0 A# 6       107  0.14
24.08: Note On:  0 D# 7        87  0.16
24.10: Note On:  0 A# 7       111  0.24
24.10:
24.11:
24.15:
24.42:
24.60:
24.62:
24.64:
24.67:
24.76:
24.79:
24.99:
25.02:
25.06:
25.14:
25.19:
25.26:
25.34:
```

```
0.00: Note On:  0 F  4       103  0.65
0.00: Note On:  0 A# 4        87  0.65
0.00: Note On:  0 A# 3        95  0.68
8.05: Controller Change:  0 64 127
8.07: Note On:  0 A# 6       107  0.14
8.08: Note On:  0 D# 7        87  0.16
8.10: Note On:  0 A# 7       111  0.24
8.10: Note On:  0 D# 4        91  0.38
8.11: Note On:  0 A# 4        69  0.38
8.15: Note On:  0 C# 5        62  0.37
8.42: Note On:  0 A# 6        69  0.21
8.60: Note On:  0 D# 7         6  0.02
8.62: Note On:  0 D# 4        62  0.11
```

Figure 1 - The Rosegarden Suite in full flow, demonstrating the common 'look and feel' of its applications.

2. User Interface Design

The project specification stated that the application I was to write should provide music entry and sequencing capability. Entering music directly as MIDI events has always been an onerous task. Users are generally fairly averse to entering long sequences of hexadecimal numbers when they would rather being either playing the music live and then editing later, or entering the music on something resembling a traditional music score. To this end I designed the Rosegarden sequencer to be oriented more towards being a 'capture and edit' device than a music entry device in its own right. MIDI data can be captured either from live playing on an attached MIDI device, from data produced by the Rosegarden Notation Editor, or from MIDI files produced by other sequencers. The sequencer does, however, allow the user to create MIDI events directly if necessary.

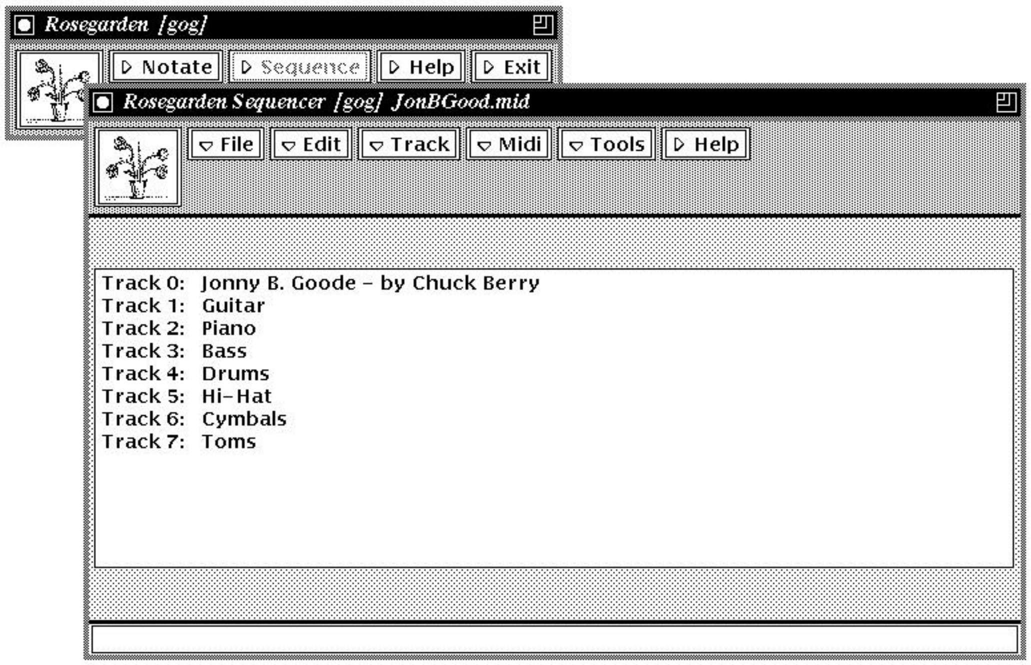
The principle features of the user interface are:

- A top-level window displaying a list of the tracks in the current MIDI file, providing menu option for manipulating whole tracks (and indeed the whole file), as well as configuration and communication options.
- Text-based 'Event List' windows, providing a detailed list of the events present in a MIDI track.
- 'Piano-Roll' windows that display the contents of a MIDI track in a graphical format.
- Full 'cut and paste' operation between tracks, including a clipboard-viewer window to display the current contents of the clipboard.
- An undo facility to allow the user to correct editing mistakes.
- A common selection paradigm across all windows.
- Dynamic updating of all windows after the completion of an editing operation. For instance, if the user edits a track from its event list window then any changes will be immediately echoed to the main window and associated piano-roll window if they are visible.
- Relevant editing options are available from all windows for ease of use.
- There is no restriction on the number of sub-window that may be present at any one time. Piano-roll and event list windows may be opened for all tracks concurrently.

2.1 The Main Window

At the top level the main window displays a list of the tracks in the current MIDI file, and provides operations to open and close MIDI files, manipulate whole tracks, alter the MIDI setup and communicate with other applications (principally the Notation Editor and the on-line help system). The title bar contains a field showing the name of the file currently loaded.

Tracks are presented as a list. Each list entry displays the number of the track and its name (if the track contains a Track Name meta-event). The instrument name is also displayed in brackets if the track contains an Instrument Name meta-event. If no file is loaded or the current file is empty then the track list is disabled and displays the message 'No Tracks'.



```
Track 0:  Jonny B. Goode - by Chuck Berry
Track 1:  Guitar
Track 2:  Piano
Track 3:  Bass
Track 4:  Drums
Track 5:  Hi-Hat
Track 6:  Cymbals
Track 7:  Toms
```

Figure 2 - The Rosegarden Sequencer Main Window

The **File** menu on the main window contains the usual options to load and save files, as well as options to display information about the currently loaded file and examine or change the timebase for the file.

The **Edit** menu contains the clipboard options. The clipboard is described in a later section.

From the **Track** menu on the main window the user can call up subwindows to display the contents of individual tracks. Tracks can be displayed in one of two formats. The first format displays the tracks as an 'Event List'. The MIDI events contained in the track are displayed (in chronological order) in a textual format This window allows for detailed examination and manipulation of the MIDI data. However it is not a particularly intuitive interface when attempting to see the shape of a piece of music, and does very little to help the user in locating particular notes or phrases. To this end a second display format is provided, styled after a 'piano-roll'. Notes are overlaid on a traditional musical staff, and are represented by boxes, the position, si▯e and colour of which denote the pitch, time and duration of the note.

The **Track** menu also contains all the track manipulation options available to the user. These can be applied to a track by selecting the track from the track list and then selecting the option. In the case of the **Track ▯ ▯ uantify** and **Track ▯ Transpose** options the user can choose the option when no tracks are selected from the track list. In this case the operation will be carried out on all the tracks in the track list. See section 2.4 for more information about the track manipulation functions.

The **Midi** option allows the user to configure the MIDI setup of the Rosegarden sequencer, and to issue a number of messages. The options are as follows:

- **MIDI Setup** - This menu option allows the user to configure the MIDI setup of the sequencer. The following dialogue is produced, prompting for information.

  The most important parameter is the name of the MIDI port, which must be set up by the user before attempting to perform MIDI input or output. The port name must correspond to the name of the UNIX device acting as a MIDI port.

  The two toggle-buttons at the top of the dialogue box control parameters for tweaking the playback strategy of the sequencer. The button labelled *'Synchronous'* controls the setup of the MIDI port. With this option enabled playback may be slightly improved on heavily loaded system. The button labelled *'Maintain Tempo'* determines whether the sequencer should attempt to maintain the tempo of the piece during playback. This option enables the speed of the piece to remain at an overall correct rate when playing back on a heavily loaded system.

Finally the user can configure the size of the event buffer used when reading in MIDI data in real-time over the MIDI link. The default value of 10,000 events should be more than adequate for all but the most complex pieces of music.

- **Set Initial Patches** - Allows the user to set the default instrument patches for each of the 16 MIDI channels. Program Change events for any channels with patches defined are broadcast immediately before playback, and may be overridden by subsequent Program Change events present in the MIDI tracks.

- **Reset** - Broadcasts 'All Notes Off' events to all channels, thus resetting the oscillators on every MIDI device attached to the sequencer.

- **System Reset** - Broadcasts the 'System Reset' message, indicating that all attached MIDI devices should immediately reset themselves to their default state.

- **System Exclusive** - This option has not yet been implemented. It will provide support for the handling of MIDI system exclusive messages.

The **Tools** menu contains menu options which communicate with other applications, or perform cosmetic configuration functions on the setup of the sequencer. The three available options are:

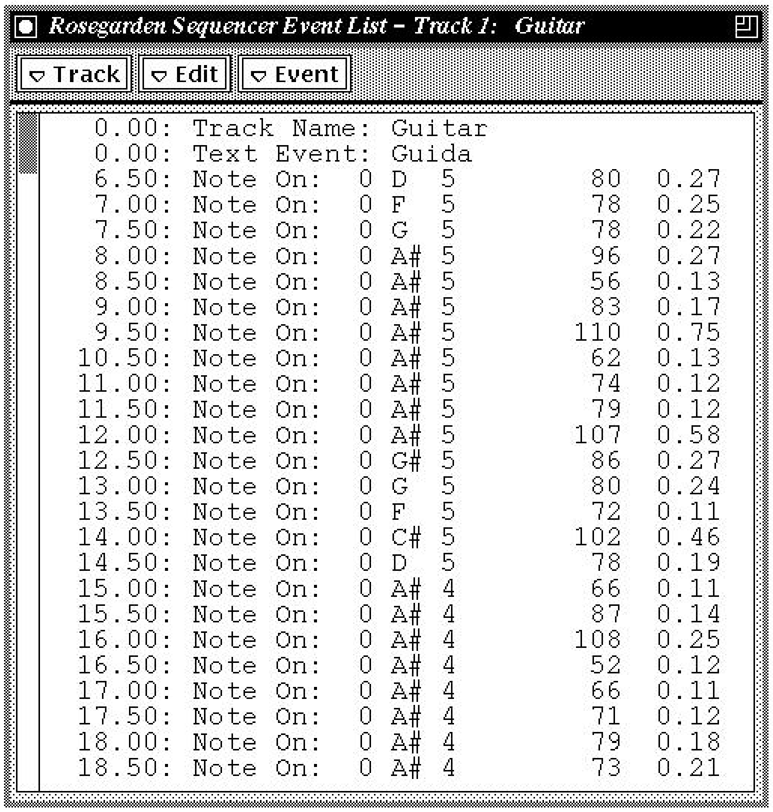
- **Notate[illegible]** - This option makes a request over the **INTERLOCK** messaging framework, asking that the Rosegarden Editor make a transcription of the MIDI data currently contained in the sequencer. The data to be transferred is written out to a temporary file in the system temporary directory, and the name of this filename is passed to the editor, which will then load the file and automatically transcribe it. This option is only available if the **INTERLOCK** framework is active (i.e. the sequencer was started from the Rosegarden Top Box).

- **Generate CSound** - This option has not yet been implemented. It is intended that it should generate CSound format orchestra and score files that will allow the contents of the MIDI file to be synthesised by the CSound system. See section 4.4 for a discussion of the proposed method.

- **Preferences** - This option will allow the user to configure various parts of the sequencer's user interface according to personal taste. Currently no areas of the user interface are configurable in this manner.

Finally the **Help** button provides a hot-link to the top-level entry in the on-line help. This service is only available via the **Interlock** framework. Therefore if the sequencer has not been started up from the Rosegarden Top Box this help button will be greyed out, and all other help buttons in the application will also be greyed out or removed.

2.2 The Event List Window



```
    0.00: Track Name: Guitar
    0.00: Text Event: Guida
    6.50: Note On:  0 D  5       80  0.27
    7.00: Note On:  0 F  5       78  0.25
    7.50: Note On:  0 G  5       78  0.22
    8.00: Note On:  0 A# 5       96  0.27
    8.50: Note On:  0 A# 5       56  0.13
    9.00: Note On:  0 A# 5       83  0.17
    9.50: Note On:  0 A# 5      110  0.75
   10.50: Note On:  0 A# 5       62  0.13
   11.00: Note On:  0 A# 5       74  0.12
   11.50: Note On:  0 A# 5       79  0.12
   12.00: Note On:  0 A# 5      107  0.58
   12.50: Note On:  0 G# 5       86  0.27
   13.00: Note On:  0 G  5       80  0.24
   13.50: Note On:  0 F  5       72  0.11
   14.00: Note On:  0 C# 5      102  0.46
   14.50: Note On:  0 D  5       78  0.19
   15.00: Note On:  0 A# 4       66  0.11
   15.50: Note On:  0 A# 4       87  0.14
   16.00: Note On:  0 A# 4      108  0.25
   16.50: Note On:  0 A# 4       52  0.12
   17.00: Note On:  0 A# 4       66  0.11
   17.50: Note On:  0 A# 4       71  0.12
   18.00: Note On:  0 A# 4       79  0.18
   18.50: Note On:  0 A# 4       73  0.21
```

Figure 3 - The Event List Window

In the event list sub-windows tracks are displayed as a list of events given in a textual format. The title bar of the event list window contains the number and name of the track it is displaying. Each entry in the list describes an event in the MIDI track. The format for each entry is,

```
Event Time        Event Name        Event Data
```

The event times are always given as the number of beats since the beginning of the track. The event data fields depend on the event type. The formats are as follows:

Event Type	Data Fields
Note Off Event	`Channel  Pitch      Octave  Velocity`
Note On Event	`Channel  Pitch      Octave  Velocity  Duration`
Controller Change	`Channel  Controller  Value`
Program Change	`Channel  Program    Value`
Channel Aftertouch	`Channel`
Polyphonic Aftertouch	`Channel  Pitch      Octave  Velocity`
Pitch Change	`Channel  Pitch-shift`
Text Meta-event	`Text String`
Set Tempo Meta-event	`New Tempo  (BPM)`
Time Signature Meta-event	`New Time Signature`
Key Signature Meta-event	`New Key Signature (Maj/Min)`
Unknown Event	`Hex-dump of event values.`

From the event list window the user has access to the **Track**, **Edit** and **Event** menus. The **Track** menu contains all the options available from the main window, with the exception of the **Clone**, **Merge** and **Event List...** options. Selecting the **Piano Roll...** option from the **Track** menu in an event list window calls up the piano roll window for the appropriate track.

The user can select events from the event list by clicking and dragging. The selected events can then be cut, copied and pasted by using the usual **Edit** options, or manipulated via the **Track** functions, which will then operate only on this subset of the track.

New events can be inserted via the options on the **Event** menu, which allow the user to create the full set of MIDI events and meta-events, with the exception of End of Track events (which are automatically placed at the end of the track in any case) and SMPTE events (which are not supported by the Rosegarden sequencer). If an event has been selected from the list then the event creation dialogue box will have the time field set to be the same as this event, although this value may still be altered by the user if desired. The event creation dialogue boxes are described in a later section.

One event list window can be called up for each track. Attempting to call up a second event list window for a track (by selecting **Track □ Event List...** from the main window) will simply bring the current event list window to the front of the display.

2.3 The Piano-Roll Window

The piano roll window provides a second means of viewing and editing a MIDI track. The main pane of the piano roll window displays a graphical representation of the track, with note events shown against a traditional musical stave. Events other than note events are not displayed on the piano-roll.

Notes are displayed as rectangular boxes, with the si□e of the box indicating the length of the note and its position denoting the pitch and time of the note. Note boxes are outlined in the background colour so as to make it clearer what is happening when notes overlap. Sharp or flat notes are displayed as light-grey boxes, natural notes as dark-grey boxes. Although this representation is not as intuitive to the user as prefixing sharp or flat signs might be the presence of such signs in a crowded piece of music would make the display too cluttered to be easily read.

Below the main pane lie a number of controls to aid the user in navigating the piece of music. These controls consist of four of repeating command buttons, allowing the user to **Fast Rewind**, **Rewind**, **Forward** and **Fast Forward** through the track. The **Forward** and **Rewind** buttons move the user forward and back through the track one bar at a time, while the **Fast Forward** and **Fast Rewind** buttons move a screenful at a time. A beat number indicator is placed centrally between the **Forward** and **Rewind** buttons so that the user can easily tell at what point in the music they are.

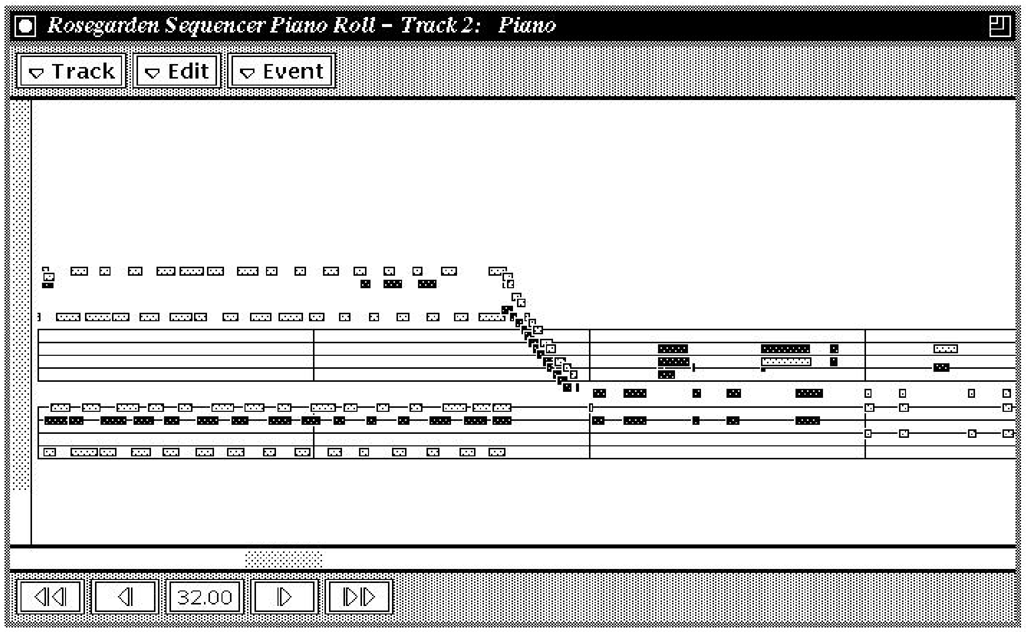


Figure 4 - The Piano Roll Window

In addition to using the navigation buttons the user can also scroll through the track by using the hori▯ontal scrollbar directly below the central pane. This scrollbar operates as for a standard Athena scrollbar widget, and as such also provides a visual indication of the proportion of the whole track represented by the notes in the window.

The piano roll window provides identical menus to those present on the event list window, with the exception of the **Piano Roll...** option on the **Track** menu, which is instead replaced by the **Event List...** option, and calls up the event list for the appropriate track.

Full cut and paste facilities are available from the piano roll window. The user can click and drag hori▯ontal areas on the graphic display to select events for manipulation in the same manner as they would click and drag to select events on the event list. Unlike for the event list, however, the user is unable to drag beyond the extent of the screen, and is thus slightly more restricted. While only note events are displayed on the piano roll click and drag selection will include any other events that lie within the appropriate time-frame.

On a monochrome display the selection area is represented by a rectangle that resi▯es as the mouse is dragged. However in order to avoid colour clash problems when running on a colour display the selection area is represented by a pair of ranging bars running hori▯ontally along the top and bottom of the graphic window, resi▯ing as the mouse is dragged.

As with the Event List windows, events can be inserted or modified via the **Event** menu. Outlining an event and then selecting the **Event ▯ Modify Event** option will call up the appropriate dialogue box for the selected event, containing the current parameters of the event which may be altered as required. The event creation menu options all call up dialogue boxes containing appropriate fields for the event type being created. The time fields of the dialogues are set up to be the point in the track where the user last clicked the mouse. Default values are provided for all event dialogue fields.

Only a single piano-roll window can be called up for each track. Attempts to call up a second piano-roll window for a particular track will cause the existing window for that track to be moved to the front of the display.

2.4 The Track Editing Facilities

The following track manipulation functions are available to the user from the **Track** menus, present on the main window and all event list and piano-roll sub-windows, with the exception of the **Clone** and **Merge** functions, which are only available from the main window. These functions provide the user with the basic tools to allow re-organisation and cleaning up of MIDI data.

2.4.1 Cloning

A track can be cloned, producing an exact copy of itself. This can be used in conjunction with the filtering functions in order to split up the data within an existing track. The cloned track is placed at the end of the track list, and will be identical as the selected track. To clone a track the user simply selects a track from the track list, followed by the **Track ▯ Clone** option from the main window. When no track is selected this option is disabled.

2.4.2 Merging

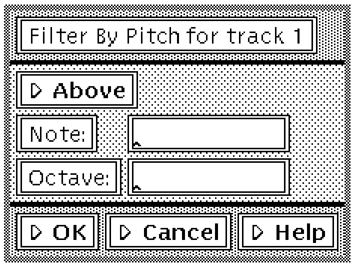
The user can merge two tracks together by selecting a track from the track list in the main window, and then selecting the **Track ▯ Merge** menu option. The user is then presented with a dialogue box prompting them to enter the number of the track to be merged with. After the user has hit OK the tracks are merged chronologically (i.e. events are interleaved depending upon their point in time).

2.4.3 Filtering By Channel

Tracks can be filtered to remove any events except for those on the desired channel. To filter by channel the user selects the **Track ▯ Filter By Channel** option. They are then presented with a dialogue box prompting them to enter the number of the channel for which events are to be retained. The user must then type a valid channel number (▯ - 15) into the text field. Selecting **OK** will carry out the filter operation on the selected track or tracks. Selecting **Cancel** will dismiss the dialogue box without invoking the operation. The dialogue box also contains a **Help** button which provides a hyper-link to the *'Track - Filter By Event'* page in the on-line help (this facility is only available if the sequencer was started from the Rosegarden Top Box).

2.4.4 Filtering By Pitch

To filter a channel with respect to pitch the user selects the **Track ▯ Filter By Pitch** option. They are then presented with the following dialogue box,



The dialogue box contains controls to specify both the pitch to filter from and the direction in which to filter. Clicking on the command button at the top of the central pane toggles the filter direction. The button is labelled either **Above** or **Below** to indicate the direction currently chosen. The pitch to filter from is entered as text in the two text fields below.

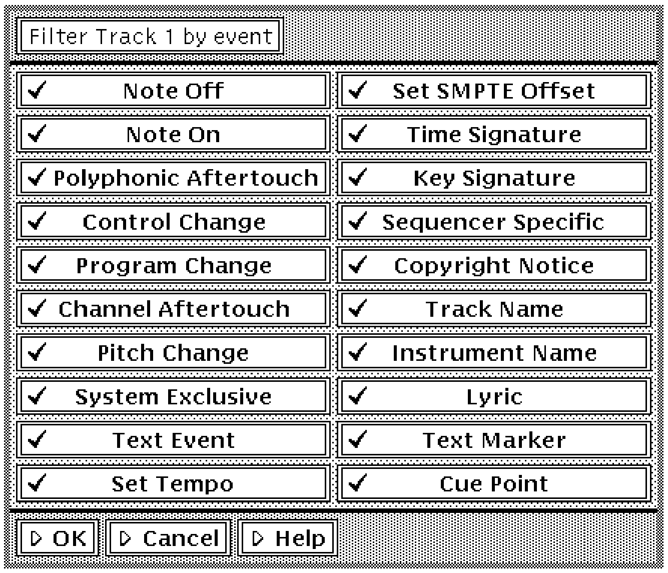
The **Note** field accepts a textual name of the note (in either upper or lower case). Notes must be specified as either naturals or sharps, (using the ▯ sign after the note). Flats are not recognised.

The **Octave** field accepts a valid MIDI octave number, between ▯ and 1▯. See Appendix B for a table of MIDI octave values.

Selecting **OK** will then filter the track retaining only those notes which are above or below the given pitch as specified by the toggle button. **Cancel** will dismiss the dialogue box without invoking the operation. The **Help** button provides a hyper-link to the *'Track - Filter By Pitch'* page in the on-line help, and is only present if the sequencer was started from the Rosegarden Top Box.

2.4.5 Filtering By Event

In addition to filtering by pitch or channel number the user can also filter tracks by event type by selecting the **Track □ Filter By Event** menu option. The user is then presented with the dialogue box shown below,



Filter Track 1 by event

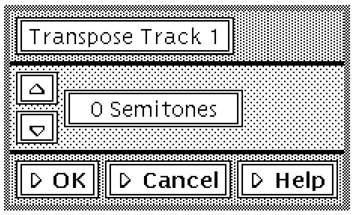
✓ Note Off	✓ Set SMPTE Offset
✓ Note On	✓ Time Signature
✓ Polyphonic Aftertouch	✓ Key Signature
✓ Control Change	✓ Sequencer Specific
✓ Program Change	✓ Copyright Notice
✓ Channel Aftertouch	✓ Track Name
✓ Pitch Change	✓ Instrument Name
✓ System Exclusive	✓ Lyric
✓ Text Event	✓ Text Marker
✓ Set Tempo	✓ Cue Point

The dialogue box contains an array of toggle buttons corresponding to the various event and meta-event types that may be present in a MIDI track. Any event types that are ticked will be retained in the track, any unticked types will be removed by the filter. The user can select any arbitrary combination of events for the filter.

Selecting **OK** will invoke the filter operation for the selected track or tracks. Selecting **Cancel** will dismiss the dialogue box without running the filter. The **Help** button provides a hyper-link to the *'Track - Filter By Event'* page of the on-line help manual, and is only present if the sequencer was started from the Rosegarden Top Box.

2.4.□ Transposition

The user can transpose a track or tracks by a number of semitones by selecting the **Track □ Transpose** option. They will then be presented with the following dialogue box,



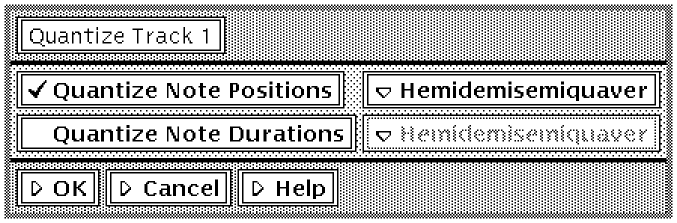
Transpose Track 1

0 Semitones

The user can alter the transposition amount by clicking on the arrow buttons on the left-hand side of the dialogue box. The buttons repeat when held. Selecting **OK** will transpose the selected track or tracks by the specified value. Selecting **Cancel** will dismiss the dialogue box without performing the transposition. The **Help** button provides a hyper-link to the *'Track - Transpose'* page of the on-line help manual, and is only present if the sequencer was started from the Top Box.

2.4.□ □ uanti□ation

The sequencer provides the ability to quanti□e MIDI data, forcing the notes to conform to a given resolution. Both the position and duration of notes can be quanti□ed. To quanti□e a track or tracks the user selects the **Track □ □ uanti□e** option, and is then presented with the dialogue box shown below.



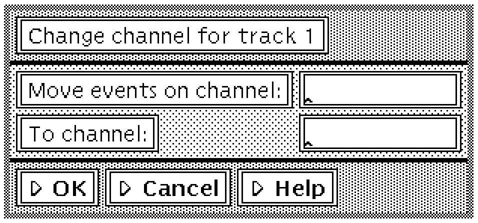
The two toggle buttons labelled **□ uanti□e Note Position** and **□ uanti□e Note Durations** allow the user to activate or deactivate quanti□ation with respect to position or duration respectively. Both forms of quanti□ation can be carried out in the same pass if desired.

When either of the toggle buttons is active a tick is displayed in the left hand corner of the button and the associated resolution menu becomes sensitive. The user can then select the resolution to quanti□e to from this menu. The menus allows the user to select semibreve, minim, crotchet, quaver, semiquaver, demisemiquaver or hemidemisemiquaver resolution for either form of quanti□ation.

Clicking **OK** will perform the quanti□e operation (although if neither forms of quanti□ation are selected the track will be left unaffected). Clicking **Cancel** will dismiss the dialogue box without performing the operation. The **Help** button provides a hyper-link to the *'Track - □ uanti□e'* page of the on-line help manual, and is only present if the sequencer was started from the Rosegarden Top Box.

2.4.□ Changing Channels

Events may be moved from one channel to another by selecting the **Track □ Change Channel** option. The user is then presented with the dialogue shown below.



The user must then enter the source and destination channel numbers into the text fields in the dialogue box. Both values must be valid channel number in the range □ to 15. Clicking on **OK** will move all events in the selected track or tracks that are on the source channel to the destination channel. **Cancel** dismisses the dialogue box without changing any events. **Help** provides a hyper-link to the *'Track - Change Channel'* page of the on-line manual, and is only available if the sequencer was started from the Rosegarden Top Box.

2.5 The Selection Paradigm

The selection paradigm is common throughout all the windows in the sequencer, and operates as follows□

If a selection has been made by clicking (or clicking and dragging) in a window then any subsequent operations in that window will operate only on the selected items. The selection remains valid until either the selection is cleared manually by the user or is invalidated by an operation.

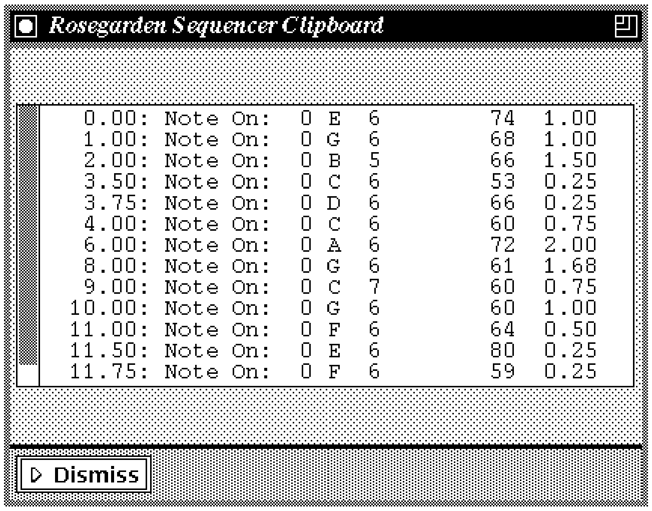
If an operation is invoked with no selection made for the window then the operation will be carried out for the entire contents of that window.

The upshot of this policy is that operations can be carried out from the main window affecting either a single track (if one is selected) or all the tracks in the file, and operations within the piano roll and event list subwindows can be carried out on either the whole track displayed in that window, or on a selection made within that track.

2. The Clipboard

The sequencer provides a clipboard mechanism, which is available at all the levels and provides a 'cut and paste' capability that can operate across tracks, or even files. Clipboard options are available from all the **Edit** menus in the sequencer. The clipboard provides **Delete**, **Cut**, **Copy** and **Paste** facilities, which function as follows,

- **Delete** - the selected events are removed and not placed in the clipboard. The time fields of any following events are not altered, effectively resulting in silence being inserted in place of any note events that have been deleted.
- **Cut** - removes the events selection specified by the user and places them in the clipboard. Events occurring after the removed selection are moved forward in time by the length of the selection.
- **Copy** - makes an identical copy of the selected events and places them in the clipboard.
- **Paste** - copies the contents of the clipboard to the insertion point specified by the user. The contents of the clipboard are left unaltered, to enable the user to paste the same events into a track at a number of points, if so desired.



```
 0.00: Note On:  0 E  6        74  1.00
 1.00: Note On:  0 G  6        68  1.00
 2.00: Note On:  0 B  5        66  1.50
 3.50: Note On:  0 C  6        53  0.25
 3.75: Note On:  0 D  6        66  0.25
 4.00: Note On:  0 C  6        60  0.75
 6.00: Note On:  0 A  6        72  2.00
 8.00: Note On:  0 G  6        61  1.68
 9.00: Note On:  0 C  7        60  0.75
10.00: Note On:  0 G  6        60  1.00
11.00: Note On:  0 F  6        64  0.50
11.50: Note On:  0 E  6        80  0.25
11.75: Note On:  0 F  6        59  0.25
```

The clipboard also provides a window that displays its contents in the form of an event list. This window is called up by the **Show Clipboard** option present on all the **Edit** menus in the sequencer and is automatically updated whenever the contents of the clipboard change. The clipboard window can be dismissed by clicking on the **Dismiss** button.

2. The Undo Facility

The sequencer allows the user to undo any operations which only affect a single track. To undo an option the user selects **Undo** from the **Edit** menu in any of the sequencer's active windows. All windows that are affected by the undo operation will automatically be refreshed.

For any operations which involve more than one track and cannot be undone a message box is displayed before the operation is invoked warning the user that the operation cannot be undone, and allowing them to cancel the operation if they wish.

2.□ User Interface Implementation Issues

The two major areas of difficulty that were encountered during the implementation of the sequencer were the limitations of the Athena list widget with regards to implementing the Event List sub-windows, and the need for a coherent yet reasonably efficient and general undo mechanism. These two problems are discussed below.

2.□.1 Implementation of Event List

As a first pass the event list in the event list sub-windows was implemented using the Athena list widget to hold the event information. This approach had to be discarded, however, due to two main limitations with the Athena list widget. Firstly the list widget only allows the selection of a single list element at a time, thus limiting the functionality of any cut and paste operations that might be provided by the event list window. Secondly and more importantly the Athena list widget creates its pane of list elements as a single pixmap, and is thus limited by X to having a maximum virtual height of 32□□□ pixels. As a medium si□e X font has characters of at least 1□ pixels high this would limit the total displayable length of a **MIDI** track to approximately 2□□□ entries, which is not long enough to contain all the event entries in a large or densely packed **MIDI** track.

The approach eventually used to overcome these difficulties was to implement the event list using the Athena text widget (which does not suffer from the same difficulties regarding its total height) and alter the standard selection behaviour so that only whole lines (corresponding to event entries) within the displayed text could be selected. This then yielded a simple implementation of a ‘click and drag’ editing facility. Finally some wrapper code was written to convert the text widget’s representation of a selection (in terms of character positions within the displayed text) to the points in the **MIDI** track corresponding to the start and end of the selection.

2.□.2 Implementation of Undo Facility

Implementing a general undo function for an application is always one of the more difficult areas of application design. Ideally users should be able to undo any operation they have made regardless of its affect on the data being manipulated. This is, however, usually quite costly on memory. As a compromise it is often acceptable to enable the user to undo operations which have only affected a subset of the data-set, and warn the user when an operation has too large a scope to be undone. This is the approach taken in the sequencer, which implements a scheme whereby operations that affect only a single track can be undone.

The undo heap consists of a single event list which represents a whole or partial **MIDI** track. To record the state of the track before an operation is carried out the track operation callbacks, call the function the function `Midi_UndoRecordOperation`, specifying that the pre-operation state be recorded in one of two ways. If the operation is localised to a contiguous area of the track, i.e. when cutting or deleting, the function is passed an event list containing the events that have been removed from the track, along with an insertion point at which the events should be replaced. If, however, the operation affects events over the whole length of a track and not necessarily in a contiguous formation, for instance when transposing or quanti□ing note events or applying a filter, the function is passed an image of the whole track in its pre-operation state, along with the number of the track’s entry in the track list. This second method is also used when cutting or deleting tracks from the track list in the main window.

When the user selects the **Edit □ Undo** option the function `Midi_UndoLastOperation` is called, which then either inserts tracks at the insertion point specified, or replaces the appropriate entry in the track list with the previous image of the track, depending on which method was used to record the operation.

This approach was taken primarily to make the use of cut and delete operations more memory efficient, as it was felt that these operations would be among those most commonly used.

3. Program Structure and Internal Representation of Datatypes

3.1. Representing MIDI events and Meta-events

The MIDI event stream is fundamentally a low-level mechanism, designed primarily for speed of transmission and interpretation by hardware. The transport layer is a simple □-bit datastream with no handshaking or error-correction. MIDI status bytes (i.e. the opcodes denoting MIDI instructions) are differentiated from data bytes by having the most significant bit set. Since some single byte MIDI messages (known as real-time system messages) must be transmitted against strict timing constraints they are allowed to occur interleaved with the data fields of other MIDI messages. MIDI data bytes can therefore only hold seven bits of data. However, as real-time messages are not recorded in MIDI files the MIDI file format can allow □-bit data fields, and indeed many of the MIDI file meta-events have □-bit fields.

3.2. Single-point vs. Dual-point Representation of Notes

In MIDI terms a note consists of two separate events. A NOTE ON event signifying the start of the note is emitted when a note is struck on a MIDI device, and a corresponding NOTE OFF event is generated when the note is released. MIDI is designed this way to enable MIDI devices to drive remote modules in real-time over the MIDI link.

For the purposes of implementing editing facilities in a sequencer, however, this representation presents certain difficulties. For instance, consider a simple operation to delete a number of notes in a MIDI track. The sequencer must scan forward from the NOTE ON event to find the next matching NOTE OFF event and delete this as well, updating the delta times accordingly. If a large number of notes are being removed then this can become very inefficient. The situation is exacerbated when trying to perform operations such as changing the length of a note. In this case the order of events in the MIDI track is likely to change, and so the sequencer must not only update the delta times but also re-order the track. It would also be very easy for NOTE OFF events to become stranded during cut and paste operations, leaving notes playing for arbitrary lengths of time.

To overcome this problem I have elected to use a single point representation for holding note events internally in the sequencer. For this purpose an extra pseudo-event is added to the MIDI event structure, with the data field containing the duration of the note in addition to the velocity and pitch held in an ordinary NOTE ON event. The problem then becomes one of being able to translate between the single and dual point representations.

3.2.1. Converting Dual-Point to Single-Point Representation

This function runs along a MIDI track held in dual-point format until it reaches a NOTE ON event. It then scans forward from this event to find the next matching NOTE OFF event[1]. Upon finding this event it calculates the time difference and stores this in the duration field of the NOTE ON event, removing the NOTE OFF event from the track. If no matching NOTE OFF event is found then the note is assumed to run until the end of the track and the time difference between the NOTE ON event and the END OF TRACK marker is used as the note's durations. This process is repeated for all NOTE ON events in the track.

In the Rosegarden Sequencer this operation is performed once for each track after it has initially been read in.

[1]A slight complication with this mechanism is the fact that the MIDI protocol allows NOTE OFF events to be alternately represented as NOTE ON events with velocity □ero. This representation is used in a number of commercial MIDI devices, including those manufactured by □ amaha.

3.2.2. Converting Single-Point to Dual-Point Representation

In order to recreate the dual-point format track when writing out the NOTE OFF events corresponding to each NOTE ON must be inserted at the correct point in the MIDI track. The sequencer does this by making use of binary-heap based priority queues.

When writing a MIDI track out to a MIDI file, the sequencer runs along the MIDI track writing each event in turn. When it encounters a NOTE ON event it examines the duration field of the event and creates a new NOTE OFF event with the correct time value to produce a note of this duration. This event is then placed on a binary heap which holds all the outstanding NOTE OFF events. The event at the root of this heap is checked against the next event in the MIDI track at each step to determine if it precedes it in time. If so then the NOTE OFF event is removed from the heap and written out to the file and the process is repeated until the root event of the NOTE OFF heap occurs after the next event in the MIDI track. The process is illustrated more clearly in the following pseudo-code fragment.

```
WriteTrack(EventList)
{
     Create empty NOTE_OFF_HEAP.

     while(EventList)
     {
          while (NOTE_OFF_HEAP is non-empty &&
                (Time(Root(NOTE_OFF_HEAP)) < Time(Head(EventList))))
          {
               write ExtractMin(NOTE_OFF_HEAP);
          }

          if (Head(EventList) == NOTE ON)
          {
               Create NOTE_OFF event;
               Insert(NOTE_OFF_HEAP, NOTE_OFF event);
          }

          write Head(EventList);

          EventList = Tail(EventList);
     }

     /* Deal with any straggling notes. */

     while(NOTE_OFF_HEAP is non-empty)
     {
          write ExtractMin(NOTE_OFF_HEAP);
     }
}
```

The method is also employed when pre-processing a track before live playback.

4. Recording and Playback Schema

Writing any software with real-time aspects under UNIX is never a particularly easy task. The pre-emptive nature of process scheduling and the lack of direct access to hardware interrupts means that any real-time application will be fighting a constant battle to maintain any sort of timing accuracy. On a high-powered UNIX platform these drawbacks are slightly alleviated by the amount of raw processing power, but the application is still inherently limited by the resolution of the UNIX timer mechanism, which is system dependent.

The MIDI transport layer is a medium speed □-bit data link running at approximately 32.5Kbit□s with no error correction or handshaking. As the average length of a MIDI real-time event is three bytes (see appendix A) this implies that any MIDI sequencing application must be able to transmit events at the rate of approximately 13□□ events per second. If running status is employed (where consecutive events with the same status byte are allowed to omit the status byte and transmit only the data bytes for second and subsequent events) this value could theoretically be marginally higher. This implies that to run a MIDI link at full speed with event resolution a system must be able to generate timer interrupts every □5□µsec. However, on the system the Rosegarden sequencer was developed on (an SGI Indigo) the default timer for applications is only capable of generating timing signals with ten millisecond resolution. These timers can be increased to millisecond resolution by use of the system utility **ftimer**, but only at the expense of a □-□□ reduction in total system performance, and in any case this facility can only be invoked by a process with super-user privileges.

It should be noted that these limitations affect only the timing resolution of the sequencer, and not the transmission speed. Thus when transmitting a chord of four notes, the note events will still be transmitted as a contiguous packet of bytes, but the timing of the chord could potentially be out by up to □ milliseconds.

In an attempt to place these timing constraints in context consider the case of a piece of music with a medium speed tempo of around 135bpm, such as for instance the opening movement of Beethoven's Sonata in C for piano (used as a test piece during the development of this software). The following table gives a correlation of note lengths and the average errors occurring when running the sequencer with a 1□ millisecond timing resolution.

Note □alue	**Duration**	**Average Error at 1□ms resolution**
Semibreve	1□□□ms	□.2□□
Minim	□□□ms	□.5□□
Crotchet	444ms	1.13□
□uaver	222ms	2.25□
Semiquaver	111ms	4.5□□
Demisemiquaver	55.5ms	□.1□□
Hemidemisemiquaver	2□.25ms	1□.□1□

In practice for medium tempo pieces these errors are not particularly discernible to the ear. Far more detrimental to playback is the effect of context switching when the host system is heavily loaded, leading to the sequencer being switched out for significantly more than 1□ milliseconds at a time. This leads to the sequencer's timing becoming far more irregular than on a lightly loaded system. If the sequencer then tries to maintain the tempo of the piece, attempting to catch up by outputting all the events that have effectively been delayed past their scheduled transmission time, then the playback of the piece becomes more 'bursty' in nature as flurries of short notes get compacted together.

This effect is not at all desirable and with a pre-emptive multi-user environment such as UNIX is difficult to overcome. The Rosegarden sequencer attempts to alleviate this problem by providing two playback modes, which may be selected by the user at run-time from the **Midi □ Midi Setup** menu option. The sequencer can either be configured to maintain the correct tempo for the piece as described above, or it can be configured to assume that each arbitrarily large increment in time it detects is equivalent to a single 1□ millisecond MIDI timeslice, which on a heavily loaded system will mean that the tempo of the piece will drop but playback will appear to be more regular to the ear, albeit still with noticeable timing irregularities.

Due to the timing constraints inherent in recording and playback the routines have to be designed to be as efficient as possible. The major impact of these constraints is that making use of the UNIX dynamic memory library functions during either recording or playback results in the routines becoming prohibitively slow.

4.1 Real-time Playback

The real-time playback routine is designed to perform the majority of the necessary processing required to make the MIDI track playable before playback begins. Firstly the track is filtered to remove all non-playable events, such as MIDI file meta-events (including End of Track events). At present the sequencer has no capability to transmit System Exclusive messages at run-time, so these are removed as well. Set Tempo meta-events are retained as they must be used by the sequencer to regulate the speed of playback. Finally the track is converted back to the two-point representation. The track is then placed in a binary heap which functions as a priority queuing mechanism during playback.

During playback events are taken from the top of the heap one at a time to be processed, transmitting each event as and when its transmission time expires.

When no events are to be transmitted within the current timeslot the routine checks the X event queue for the application and if any X events are outstanding it services one of these. This is to provide the user with the ability to stop playback partway through the piece if desired by clicking on the stop button that is displayed during playback. The callback attached to this button then modifies the `Playing` variable and the routine drops out of the playback loop, broadcasts the Stop All Notes message on all MIDI channels, and closes down the MIDI port. The loop is also exited when the MIDI event heap is empty (signifying that the end of the piece has been reached).

A pseudocode description of the playback routine is shown below.

```
Sequence()
{
     while (Playing)
     {
          CurrentTime = GetCurrentTime();

          if (CurrentTime > LastTime)
          {
               if (Maintaining Tempo)
               {
                    SequenceTime = (CurrentTime - LastTime) *
                                    TempoSetting;
               }
               else SequenceTime += TempoSetting;
          }

          while ((NextEvent = Root(EventHeap) and
                 Time(NextEvent) <= SequenceTime)
          {
               if (NextEvent == TempoEvent) Change TempoSetting;
               else Transmit NextEvent;

               ExtractRoot(EventHeap);

               if (Next(NextEvent))
                    Insert(EventHeap, Next(NextEvent);

          }

          if (NextEvent == NULL) Playing = False;
          if (X events queued) Process 1 X event;
     }

     Clear All Notes;
```

```
        Close MIDI port;
    }
```

Note that the increment of the current sequence time must take into account the number of clock ticks that have expired since the previous passage through the loop. This then allows the sequencer to 'catch up' when it has fallen behind either due to an excessive time spent in the inner loop transmitting MIDI events during a dense section of the music, or due to system exertions, and thus to maintain the overall tempo of the piece.

4.2 Real-time Recording of MIDI data

The recording routine is similarly constrained with respect to time. The routine must read in all the events coming in from the MIDI port and store them in memory with a time stamp. To speed up the routine events are stored in a raw format and the time stamp they are given is the number of 10 millisecond clicks since the start of recording. As meta-events are not transmitted over the MIDI link, and the sequencer provides no support for System Exclusive messages (see section 4.3) memory allocation for these raw events is greatly simplified by the fact that they are all of a fixed length (of either two or three bytes dependent on the event type).

The recording routine pre-allocates a storage buffer for the incoming raw MIDI events and then listens on the MIDI port filling in an event entry each time an event is received over the link. The algorithm is complicated by two features of the MIDI protocol:

- **Running Status** - This is where the status byte can be omitted for an event if it is the same status bytes as for the event immediately previous. To cope with this the recording routine must keep a record of the last event that was received over the MIDI link and then if the next byte read in is a data byte (i.e. does not have its MSB set) use the value of the previous status byte while constructing the new incoming event.

- **MIDI System Real-Time Messages** - The MIDI protocol defines a number of single byte messages known collectively as the system real-time messages. These message provide facilities such as clocking, tune up requests, etc., and may appear *at any time* in the MIDI byte stream, including being interleaved within the data bytes for a multi-byte event. Thus the recording routine must filter out these messages when reading in data from the MIDI link, and ensure that the correct number of actual data bytes is read in for each incoming event.

After recording has stopped (using a STOP button mechanism similar to that used in the playback routine described above) the raw MIDI event buffer is transformed into the linked list representation used internally by the sequencer, and is subsequently converted to use the sequencer's internal one-point note representation. The recorded data is then returned to the user as a new entry at the foot of the track list displayed in the main window.

In practice the recording of MIDI data turns out to be much more adversely affected by system loading than playback. The serial port on the target hardware is not buffered when configured as a MIDI port, and therefore events very easily get lost, especially on a heavily loaded system where the sequencer can be switched out for large fractions of a second. This event loss effect is exasperated by the two-point representation of notes in the MIDI protocol. If the Note Off event is missed during recording the note will continue to play until the next Note Off event for that note is received, or until the end of the track, when a synthetic Note Off event will be inserted by the sequencer as part of the transition to its single-point internal representation. This can easily result in very garbled input.

4.3 Limitations of Real-Time Playback and Recording

The recording and playback routines implement only the most commonly used subset of the MIDI real-time protocol. The system makes no attempt to produce or interpret any of the following system real-time messages:

- **MIDI Clock Signal** - These signals can be transmitted over the MIDI link to enable tight synchronisation of remote devices. Due to timing problems described above any attempts to implement transmission of these signals by the sequencer would be almost futile. Use of these signals is in any case not widespread and most MIDI devices ignore them.

- **SMPTE Time Signal** - These signals are similar to those described above, but provide a far more accurate timing mechanism. Again due to the timing constraints they would prove almost impossible to implement for the target environment. Their use is even less widespread than standard MIDI clock signals, and is largely confined to professional MIDI recording and audio-visual applications (such as production of film soundtracks) where accurate timing of sound events is required.

- **Active Sensing** - As part of the MIDI protocol it is defined that after receiving an Active Sensing message a MIDI device should expect to receive such messages at no greater than 300 millisecond intervals. If a signal is missed then the MIDI device should reset itself, turning off any notes that are currently playing. Due to the timing constraints this is not a sensible thing to implement for the Rosegarden sequencer, as on a heavily loaded system it is quite conceivable that the sequencer would get switched out for over 300 milliseconds, with the net effect that all attached devices would reset themselves mid-way through the piece.

- **System Exclusive** - The MIDI protocol provides the System Exclusive message as a means to send proprietary data over the MIDI link. System Exclusive messages are received by all the MIDI devices on the link, but are only interpreted by a device if the manufacture and product codes contained within them are recognised. System Exclusive message consist of an initial status byte (F0) followed by an arbitrary number of seven-bit data bytes, and terminated with the EOX status byte (F7). System Exclusive message may be transmitted during real-time playback. At present the Rosegarden sequencer does not support System Exclusive messages. It is possible, however, that the sequencer will be enhanced in the future to do so. The use of System Exclusive messages during real-time playback will probably not be supported, but the user will be able to transmit and receive the messages over the link as a separate option on the **Midi** menu.

The sequencer also provides no support for MIDI Thru. The MIDI standard defines three port types. MIDI In, which receives events, MIDI Out which transmits events, and MIDI Thru, which echoes all the events it receives to its output channel. The Rosegarden sequencer functions as a MIDI In port when recording and a MIDI Out port when playing. The lack of support for MIDI Thru means only that the sequencer must be the terminating device when daisy-chaining a number of MIDI devices on the same link.

4.4 Outputting Data for C-sound

5. MIDI Library Functions

The MIDI library written for use with the Rosegarden Sequencer and Editor may be used by anyone wishing to write MIDI software in a UNIX (or similar) environment. The library provides file-handling functions, event creation utility functions, track manipulation functions, and a generic implementation of binary heaps that is used internally by the library but is available to the programmer. The following sections provide a brief descriptions of the functions and macros provided by the MIDI library.

The MIDI library makes use of the linked-list library written by Chris Cannam, which must be linked in separately when compiling.

5.1 File Handling

A large number of functions are provided to enable the programmer to cope with the somewhat arcane MIDI file format. In this description the functions are split into those providing high-level facilities, such as opening file and reading or writing whole MIDI tracks, and those providing low-level facilities, such as reading or writing individual bytes. The programmer should in general use the high-level commands in preference to the low-level. Standard file-system functions such as `fread`, `fputc` or `fseek` should never be used on files opened as MIDI files, as the use of these functions could cause inconsistencies that might result in the file becoming corrupted. Appendix C gives a brief and informal overview of the MIDI file format.

5.1.1 High-Level Access

The high-level functions provide the programmer with the ability to transfer MIDI events and tracks to and from files without having to worry about how they are represented in the file format.

5.1.1.1 Opening a File

```
MIDIFileHandle Midi_FileOpen(char            *FileName,
                             MIDIHeaderChunk *HeaderBuffer,
                             MIDIFileMode     Mode)
```

This function opens a file in preparation for reading/writing. The function is passed the following parameters:

- **FileName** - the name of the file to be opened.
- **HeaderBuffer** - a pointer to a header information buffer. The header buffer field must point to a buffer where the header chunk information is to be stored if the file is being opened for reading, or to a buffer containing the header info to be written at the start of the file if opening for writing.
- **Mode** - the file access mode which may be `MIDI_READ`, `MIDI_WRITE` or `MIDI_READ_WRITE`. The function returns a MIDI file handle which is used to reference the opened file in subsequent actions.

The function returns a MIDI file handle, which should be used by the application in subsequent calls to MIDI file functions.

When opening a file for reading the function initially scans the first four bytes of the specified file, matching against the chunk identifier for a MIDI header chunk. If the file does not match or an error is found in the header information, then an error will be raised, the file will be closed and the file handle returned will be NULL.

After `Midi_FileOpen` returns the file pointer is pointing to the byte immediately following the end of the header chunk. When reading this should be the chunk identifier for the first track chunk. To move into this chunk the next operation performed on the file should be a call to `Midi_FileSkipToNextChunk` (see below).

5.1.1.2 Closing a file

```
void Midi_FileClose(MIDIFileHandle DoomedFile)
```

This function closes an open MIDI file and frees the file handle. DoomedFile is the file handle specifying the file that is to be closed. This function must be called for all files after operations have been completed on them. This is particularly important when a file has just been written as `Midi_FileClose` must then perform some tidying on the output file before it can be close and subsequently read in again. Neglecting to close a file after writing it out will cause the final track in the file to be corrupted.

5.1.1.3 Navigating File Chunks

```
int Midi_FileSkipToNextChunk(MIDIFileHandle MidiFile,
                             MIDIChunkType  ChunkIdentifier)
```

The MIDI file format is a chunked format, with header information and separate tracks stored in separate chunks of the file. Each chunk has an identifying string of four characters at its start - the chunk identifier.

This function skips to the next chunk in the MIDI file that matches the identifier passed in. The parameters are:

- **MidiFile** - The MIDI file handle for the file being navigated.
- **ChunkIdentifier** - The chunk identifier to match against. The function will skip over the chunks in the file until it finds a chunk matching the supplied identifier. The current MIDI chunk types are defined as `MIDI_FILE_HEADER` and `MIDI_TRACK_HEADER`. Other chunk identifiers may be used to store proprietary info in a MIDI file, but this is not recommended.

This function returns `EOF` upon reaching the end of a file without finding a satisfactory chunk. Programmers should always use this function to move between chunks in a MIDI file - the low-level file navigation functions will not move past chunk boundaries. Using the standard file-system navigation functions could cause the MIDI file to become corrupted in subsequent operations.

5.1.1.4 Reading an Event

```
MIDIEvent Midi_FileReadNextEvent(MIDIFileHandle MidiFile)
```

This function reads in the next event from the current chunk of a MIDI file. The event is allocated space on the heap, which must be freed as necessary by the calling application.

5.1.1.5 Reading a Track

```
EventList Midi_FileReadTrack(MIDIFileHandle MidiFile)
```

This function reads in the track contained in the current trunk and returns a pointer to a list of events. Space for the track is allocated on the heap. The track space must be freed explicitly by the calling application (via a call to `Midi_TrackDelete`) when no longer needed.

5.1.1.▯ Writing an Event

```
void Midi_FileWriteEvent(MIDIFileHandle MidiFile, MIDIEvent Event);
```

This function writes a single MIDI event to the specified file. The parameters are:

- **MidiFile** - The file handle for the MIDI file to write to.
- **Event** - The MIDIEvent that is to be written out.

5.1.1.▯ Writing a Track

```
void Midi_FileWriteTrack(MIDIFileHandle MidiFile, EventList Track);
```

This function writes a the specified MIDI track to the output file. The track is placed in a new chunk - the chunk identifier `MIDI_TRACK_HEADER` is written out the file immediately preceding the track data. The parameters are:

- **MidiFile** - The output file handle.
- **Track** - Pointer to the list of events to be written out as the next track in the file.

5.1.2 Low Level Access

The low-level file functions provide a set of primitives and utilities used when reading or writing a MIDI file. The functions should always be used in preference to their equivalents in the standard file library as they automatically update information in the MIDI file handle, and perform other useful tasks, such as checking against chunk boundary over-run. Use of the standard system file library on a file opened as a MIDI file could result in the file becoming corrupted.

5.1.2.1 Reading a Byte

```
int Midi_FileReadByte(MIDIFileHandle File)
```

Read a single byte from the input stream. This function will return `EOF` if an attempt is made to read past the end of a file, or `MIDI_FILE_ERR_END_OF_CHUNK` if an attempt is made to read past a chunk boundary.

5.1.2.2 Reading a Number of Bytes

```
int Midi_FileReadBytes(MIDIFileHandle  File,
                       byte           *Buffer,
                       int             NumBytes)
```

This function reads a number of bytes from the MIDI input stream and places them in the buffer provided. If the buffer is not large enough to accept the number of bytes read in then unpredictable results may occur.

5.1.2.3 Writing a Byte

```
void Midi_FileWriteByte(MIDIFileHandle MidiFile, byte Byte)
```

This function writes the byte **Byte** to the output file specified by **MidiFile**.

5.1.2.4 Writing a Number of Bytes

```
void Midi_FileWriteBytes(MIDIFileHandle  MidiFile,
                         byte           *Buffer,
                         int             NumBytes)
```

This function writes a number of bytes to the specified output stream. The parameters are as follows:

- **MidiFile** - the file handle specifying the output stream.
- **Buffer** - a pointer to the buffer containing the bytes to be written.
- **NumBytes** - the number of bytes to be written.

5.1.2.5 Rewinding

```
int Midi_FileRewindNBytes(MIDIFileHandle MidiFile, int Distance)
```

This function rewinds the MIDI file by the number of bytes specified in the **Distance** parameter. Attempts to rewind past a chunk boundary will result in the generation of a `MIDI_FILE_ERR_END_OF_CHUNK` error.

5.1.2.▯ Fast-Forwarding

```
int Midi_FileFfwdNBytes(MIDIFileHandle MidiFile, int Distance)
```

This function moves the file pointer forward in the MIDI file by the number of bytes specified in the **Distance** parameters. Attempts to fast-forward past a chunk boundary will result in the generation of a `MIDI_FILE_ERR_END_OF_CHUNK` error.

5.1.2.▯ Determining the Distance from the end of the Chunk

```
#define Midi_FileBytesLeftInCurrentChunk(X)
```

This macro returns the number of bytes remaining in the current chunk of the specified MIDI file. It is passed a MIDI file handle as its argument. The most common use for this macro is as a test to determine whether the file pointer is at the end of the chunk, as demonstrated in the code fragment below,

```
                        .
                        .

            if (!Midi_FileBytesLeftInCurrentChunk(fp))
            {
                  Midi_FileSkipToNextChunk(fp, MIDI_TRACK_HEADER);
                              .
                              .
            }
```

5.1.2.▯ Reading a ▯ariable Length Number

```
int Midi_FileReadVarNum(MIDIFileHandle MidiFile, long *ReturnValue)
```

This function reads in a MIDI format variable length number field from the file specified in **MidiFile**, converts it to a 32-bit integer and places it in the “long” buffer pointed to by **Return▯alue**.

5.1.2.▯ Writing a ▯ariable Length Number

```
void Midi_FileWriteVarNum(MIDIFileHandle MidiFile, long Value)
```

This function writes the value specified in the parameter **▯alue** to the MIDI output stream in the MIDI variable length number format.

5.1.2.1▯ Writing a Chunk Identifier

```
void Midi_FileWriteNewChunkHeader(MIDIFileHandle MidiFile,
                                  MIDIChunkType  ChunkIdentifier)
```

This function is used by `Midi_FileWriteTrack` to write the chunk identifier preceding each track in the MIDI file. It can be used by the programmer to write non-standard chunk identifier which can be used to store proprietary information in a standard MIDI file. These chunks will then be ignored by any applications that correctly implement the MIDI file standard, and will be recognised only by applications that know the chunk identifier associated with this type of proprietary data.

5.1.2.11 Integer▯Byte-stream conversion functions

```
#define ByteStreamTo32BitInt(PTR)
```

Utility function to convert a byte-stream representation of a 32 bit integer as read in from a MIDI file into an actual value. This is an attempt to make interpretation of MIDI file fixed-width integers platform independent. It takes a pointer to a 32 bit buffer containing the raw MIDI bytes stream as its argument.

```
#define ByteStreamTo16BitInt(PTR)
```

As above, but for 1□ bit integers.

5.2 Track Manipulation

The following functions all manipulate whole or partial MIDI tracks (i.e. EventLists). In general the functions are non-destructive to enable the simple implementation of facilities such as 'undo'. A new event list is created by the functions and a pointer to the start of this is returned. The exceptions to this are `Midi_TrackDelete`, for which it would obviously serve no useful purpose, and `Midi_TrackConvertToOnePointRepresentation` and `Midi_TrackAggregateDeltas` which are concerned with preprocessing tracks after they have been read in from a MIDI file. All the track functions apart from these three will only operate correctly on tracks which have been converted to the one-point representation and have had the event times aggregated.

5.2.1 Converting Two-Point to One-Point Representation

```
void     Midi_TrackConvertToOnePointRepresentation(EventList Track)
```

This functions takes a raw MIDI track as read from a MIDI file and transforms it from dual-point to single-point representation (see section 3.2.1). It scans for Note On - Note Off pairs and deletes the Note Off component, modifying the Note On event to contain an extra field indicating the duration of the note.

5.2.2 Converting One-Point to Two-Point Representation

```
EventList Midi_TrackConvertToTwoPointRepresentation(EventList Track)
```

This function takes a MIDI track that has been converted to single-point representation and transforms it back to a dual-point representation. A description of the implementation of this function along with a discussion of its use is given in section 3.2.2.

5.2.3 Aggregating Event Time Fields

```
void     Midi_TrackAggregateDeltas(EventList Track)
```

This function alters the DeltaTime field of each MIDI event in the track to contain the time the event occurs, represented as the number of MIDI clicks after the start of the track. This function should be called for each track after it has been read in and before any track operations are performed on it.

5.2.4 Deleting a Track

```
void     Midi_TrackDelete(EventList Track)
```

This function frees all the memory allocated for the specified track.

5.2.5 Cloning a Track

```
EventList Midi_TrackClone(EventList Track)
```

This function returns an identical copy of the specified track.

5.2.▯ Transposing a Track

```
void    Midi_TrackTranspose(EventList Track, int Delta)
```

This function transposes the given track, altering all notes by the value passed in. The **Delta** value indicates the number of semitones to transpose by. Positive values transpose upwards, negative values downwards.

5.2.▯ Filtering a Track by Channel

```
EventList Midi_TrackFilterByChannel(EventList Track, byte Channel)
```

This function filters the given MIDI track and returns a new track that contains only events for the MIDI channel specified in **Channel**, which must be a number between ▯ and 15. All file meta-events are also stripped by this function. The original unfiltered track is left unchanged, and can be re-used or freed by the application as desired.

5.2.▯ Filtering a Track by Event Type

```
EventList Midi_TrackFilterByEvent(EventList      Track,
                                  Midi_EventMask Mask)
```

This function filters a MIDI track by event type. The mask passed into the function in the **Mask** parameter determines which events are retained, and can take the following values:

```
MidiNoEventMask
MidiNoteOffEventMask
MidiNoteOnEventMask
MidiPolyAftertouchEventMask
MidiCtrlChangeEventMask
MidiProgChangeEventMask
MidiChnlAftertouchEventMask
MidiPitchBendEventMask
MidiSystemExEventMask
MidiTextEventMask
MidiSetTempoEventMask
MidiSmpteOffsetEventMask
MidiTimeSignatureEventMask
MidiKeySignatureEventMask
MidiSequencerSpecificEventMask
MidiCopyrightNoticeEventMask
MidiTrackNameEventMask
MidiInstrumentNameEventMask
MidiLyricEventMask
MidiTextMarkerEventMask
MidiCuePointEventMask
MidiSongPosPtrEventMask
MidiTuneRequestEventMask
MidiSequenceNumberEventMask
```

Combinations of event types can be filtered for by taking the logical OR of any combination of the above masks. The following combination masks are predefined:

```
MidiAllEventsMask
MidiSoundEventsMask
MidiTextEventsMask
MidiMetaEventsMask
```

For instance, to filter a track to contain only meta-events and note on events the programmer would use the call,

```
      Midi_TrackFilterByEvent(TrackPtr, MidiMetaEventsMask |
                                        MidiNoteOnEventsMask);
```

The 'end of track' meta-event is never removed from any track passed through the event filter. The original unfiltered track is left unchanged, and can be re-used or freed by the application as desired. Note that in one-point representation all notes are represented by a single Note On event.

5.2.□ Filtering a Track by Pitch

```
EventList Midi_TrackFilterByPitch(EventList Track,
                                  byte      Pitch,
                                  Boolean   RetainAbove)
```

This function allows the program to filter a track to only contain note events which lie above or below a specified pitch. The parameters are as follows:

- **Track** - a pointer to the start of the track to be filtered.
- **Pitch** - the pitch at which to start filtering. Filtering is done inclusive of this pitch. The pitch is specified as a raw MIDI value in the range □ to 12□, see the appendices for a table relating MIDI pitches to note value and octave.
- **RetainAbove** - a Boolean flag indicating in which direction to filter. If `True` is passed in then events above the specified pitch are retained, `False` will retain pitches below the specified pitch.

The original unfiltered track is left unchanged, and can be re-used or freed by the application as desired.

5.2.□ □ uanti□ing a Track

```
EventList Midi_TrackQuantize(EventList        Track,
                             MIDIHeaderChunk *Header,
                             Boolean          QuantizePos,
                             int              QuantizePosRes,
                             Boolean          QuantizeDur,
                             int              QuantizeDurRes)
```

This function quanti□es a track to force the notes it contains to conform to the specified granularity in terms of either their position in the piece or their duration. The parameters are:

- **Track** - a pointer to the start of the track to be quanti□ed.
- **Header** - a pointer to the MIDI file header chunk that the track was read from. This is required so that the function can determine the timebase for the track.
- **□ uanti□ePos** - a Boolean value specifying whether the positions of the notes in the track should be quanti□ed. `True` will result in the note positions being quanti□ed to fit the specified granularity, `False` will leave the notes positions unchanged.
- **□ uanti□ePosRes** - a value specifying the granularity that note positions should be quanti□ed to. This value should be one of the following,

  ```
  MIDI_SEMIBREVE
  MIDI_MINIM
  MIDI_CROTCHET
  MIDI_QUAVER
  MIDI_SEMIQUAVER
  MIDI_DEMISEMIQUAVER
  MIDI_HEMIDEMISEMIQUAVER
  ```
- **□ uanti□eDur** - this parameter is a Boolean value specifying whether note lengths should be quanti□ed. `True` will result in note lengths being forced to multiples of the specified granularity, `False` will leave the note lengths unchanged.

- **□uanti□eDurRes** - specifies the granularity note lengths should be quanti□ed to. The values should be as described above for **□uanti□ePosRes**.

The function alters the note events in a track in two ways. If quanti□ing by position then the notes are moved in time to the closest whole unit of granularity. If quanti□ing by duration then the length of the note is adjusted to be the closest multiple of the specified granularity, with the condition that notes will always have a length of greater than □ero. Both duration and position can be quanti□ed in the same pass.

The original unfiltered track is left unchanged, and can be re-used or freed by the application as desired.

5.2.1□ Changing Channels on a Track

```
EventList Midi_TrackChangeChannel(EventList Track,
                                  byte      ChangeFrom,
                                  byte      ChangeTo)
```

This function moves all the events on the channel specified in the **ChangeFrom** parameter to the channel specified in **ChangeTo**. Both **ChangeTo** and **ChangeFrom** must be in the range □ to 15.

5.2.11 Merging Two Tracks

```
EventList Midi_TrackMerge(EventList Track1, EventList Track2)
```

This function merges the specified MIDI tracks. The tracks are merge so that the resulting combined track still has all the events in chronological order.

5.3 Event Handling Functions

Some simple utility functions are provided for working with events. Utility functions are provided to simplify the creation of events, conversion of event structures to event lists, and functions to aid with interpreting MIDI values held in some events.

5.3.1 Creating a Text Event

```
MIDIEvent Midi_EventCreateTextEvt(byte  EventCode,
                                  long  DeltaTime,
                                  char *Text)
```

This function creates a text meta-event from the parameters provided. The parameters are:

- **EventCode** - specifies the type of text event. The possible event types are:

```
MIDI_TEXT_EVENT
MIDI_COPYRIGHT_NOTICE
MIDI_TRACK_NAME
MIDI_INSTRUMENT_NAME
MIDI_LYRIC
MIDI_TEXT_MARKER
MIDI_CUE_POINT
```

- **DeltaTime** - the time field for the event. If the event is to be inserted into an aggregated event list then this should be an absolute value.

- **Text** - the text string for the event.

5.3.2 Creating a Note Event

```
MIDIEvent Midi_EventCreateNote(long Time,
                               byte Channel,
                               byte Pitch,
                               byte Velocity,
                               long Duration)
```

This function creates a note event in one-point representation, and should therefore not be used with non-aggregated tracks. To create a Note On event for two-point representation tracks, use `Midi_EventCreateSoundEvt`, described below. The parameters for the function are:

- **Time** - the time field for the event, expressed as an absolute value from the start of the track.
- **Channel** - the channel for the event, in the range □ to 15.
- **Pitch** - the pitch of the note, expressed as a raw MIDI value in the range □ to 12□. See the appendices for a table relating MIDI pitch values to note and octave.
- **□elocity** - the volume of the note as a raw MIDI value in the range □ to 12□, with □ being silence and 12□ being full volume.
- **Duration** - the duration of the note.

5.3.3 Creating a Sound Event

```
MIDIEvent Midi_EventCreateSoundEvt(byte EventCode,
                                   long DeltaTime,
                                   byte Channel,
                                   byte Param1,
                                   byte Param2)
```

This is a generic function for producing MIDI sound events. The function can be used to create Note On, Note Off, Aftertouch, Control Change, Channel Change and Pitch Bend events.

The parameters for the function are:

- **EventCode** - the type of event being created. This should be one of the following values:

```
MIDI_NOTE_OFF
MIDI_NOTE_ON
MIDI_POLY_AFTERTOUCH
MIDI_CTRL_CHANGE
MIDI_PROG_CHANGE
MIDI_CHNL_AFTERTOUCH
MIDI_PITCH_BEND
```

- **DeltaTime** - the time field for the event. If the event is to be inserted into an aggregated event list then this should be an absolute value.
- **Channel** - the channel for the event, in the range □ to 15.
- **Param1** - the first parameter for the event. The meaning of this field is dependent on the event being created, but should always be in the range □ to 12□.
- **Param2** - the second parameter for the event. As for **Param1** the meaning is dependent on the type of event, but should always be in the range □ to 12□. If the event being created only has one parameter then this argument is ignored.

5.3.4 Creating a Tempo Event

```
MIDIEvent Midi_EventCreateTempoEvt(long DeltaTime, long BPM)
```

This function creates a Set Tempo meta-event. The parameters are:

- **DeltaTime** - the time field for the event. If the event is to be inserted into an aggregated event list then this should be an absolute value.
- **BPM** - The tempo to be set in Beats per Minute.

The BPM value is converted to MIDI's internal representation of tempo. The function `Midi_EventConvertTempoToBPM` should be used when subsequently reading the tempo value from the event.

5.3.5 Creating a Time Signature Meta-Event

```
MIDIEvent Midi_EventCreateTimeSigEvt(long DeltaTime,
                                     byte Numerator,
                                     byte Denominator)
```

This function creates a Set Time Signature meta-event. The parameters specify the time of the event, and the numerator and denominator of the time signature respectively. The numerator and denominator are both specified as literal numeric values.

Note that the time signature meta-event does not affect MIDI playback in any way and is simply a notational convenience.

5.3.6 Creating a Key Signature Meta-Event

```
MIDIEvent Midi_EventCreateKeySigEvt(long DeltaTime,
                                    byte Sf,
                                    byte MaMi)
```

This function creates a Set Key Signature meta-event in sharps/flats notation. The parameters are:

- **DeltaTime** - the time field for the event. If the event is to be inserted into an aggregated event list then this should be an absolute value.
- **Sf** - the number of sharps or flats in the key signature. This is expressed as a numeric value on a scale from -7 (seven flats) to 7 (seven sharps). Zero represents the key of C major / A minor.
- **MaMi** - a numeric value specifying whether the key is a major key (0) or a minor key (1).

Note that the time signature meta-event does not affect MIDI playback in any way and is simply a notational convenience.

5.3.7 A General Ordering Function on MIDI Events

```
int     Midi_EventTimeLessp(void *Ev1, void *Ev2)
```

The function `Midi_EventTimeLessp` provides a general ordering function on two events with respect to the time domain, returning 1 if the event **Ev1** precedes event **Ev2**, 0 otherwise. This function works equally well for both aggregated and non-aggregated tracks. However when using with non-aggregated tracks care must be taken to ensure that the relative time of the two events is being measured from the same place.

5.3.8 Converting a MIDI Event to an Event List.

```
EventList Midi_EventCreateList(MIDIEvent NewEvent,
                               Boolean   RetainEventStruct)
```

This function creates an EventList from a MIDIEvent. The function should be applied to MIDI event structures before they are inserted into a track event list. The parameters specify the event structure to be converted, and a Boolean value, **RetainEventStruct**, indicating whether the event structure should be freed after conversion. Setting **RetainEventStruct** to `True` retains the original event structure in memory, `False` discards it.

5.3.□ Reading the □alue of a Tempo Event

```
long    Midi_EventConvertTempoToBPM(MIDIEvent TempoEvent)
```

This utility function returns the tempo in Beats Per Minute from the value as held in the MIDI Set Tempo event supplied. If the event passed to this function is not a Set Tempo event the function will return □ero.

□. Conclusions and Further Work

I believe that most of the aims of the project have been successfully fulfilled by the Rosegarden sequencer. The major area of difficulty with the system lies in the timing constraints placed upon it by the UNIX multitasking environment. These difficulties do not have too great an impact on the end user, providing that the system the sequencer is running on is fast enough and the load is light enough. In practical terms this means that real-time recording and playback are not particularly viable on a machine such as `gog` (the Maths Undergraduate machine used for development), except if it can be used as a single-user machine.

The level of integration achieved with the music notation editor written by Chris Cannam in parallel with this project has, in my opinion, led to both applications having far greater general utility than might otherwise have been the case given the limited development timescale. By using an inter-process messaging framework, however, we have also managed to retain the independence of the two applications. Both the editor and the sequencer function perfectly well in a stand-alone configuration.

The project has resulted in the production of two major libraries, both of which are designed to be generic and re-usable by anybody attempting a similar project at a later date. The **Interlock** framework is entirely independent of the Rosegarden Suite, and may be used for any application where a service-based inter-process messaging system is required. The framework is document fully in Appendix D. The MIDI file and track handling functions are also supplied as a general-purpose library, and may be re-used if desired by anyone embarking on a similar MIDI-oriented project in the future.

As it stands the Rosegarden sequencer provides a reasonably complete implementation of the most commonly used areas of a MIDI sequencer. There is, as always, plenty of scope for enhancement, however. The quanti□ation facility, although easy to use, is quite limited in the operations it can perform on the MIDI data. It cannot, for instance, correctly quanti□e triplet phrases in its present form. Also, quanti□ation is currently only available with regard to timing, whereas ideally one would be able to quanti□e with respect to velocity as well. Similarly support for widespread modification of specific event types (such as making all the notes in a piece quieter or louder) is lacking. However, I believe that as it stands it represents a workable system when running on a machine in a single-user situation.

Appendix A - MIDI Status Bytes

(adapted from “MIDI by the Numbers” by D. □ alenti, Elec Musician mag 2□□□)

STATUS B□ TE	Function	DATA B□ TES	
1□□□nnnn (□□-□F)	Note Off (Channel nnnn)	Note □ alue (□-12□) see Appendix B	Note □ elocity (□-12□)
1□□1nnnn (□□-□F)	Note On (Channel nnnn)	Note □ alue (□-12□) see Appendix B	Note □ elocity (□-12□)
1□1□nnnn (A□-AF)	Polyphonic Aftertouch (Channel nnnn)	Note □ alue (□-12□) see Appendix B	Aftertouch Pressure (□ - 12□)
1□11nnnn (B□-BF)	Control□Mode Change (Channel nnnn)	See Appendix C	See Appendix C
11□□nnnn (C□-CF)	Program Change (Channel nnnn)	Program Number	N□A
11□1nnnn (D□-DF)	Channel Aftertouch (Channel nnnn)	Aftertouch Pressure	N□A
111□nnnn (E□-EF)	Pitch Wheel Change (Channel nnnn)	MSB (□-12□)	LSB (□-12□)
1111□□□□ (F□)	System Exclusive	□□	□□
1111□□□1 (F1)	System Common - undefined	□	□
1111□□1□ (F2)	System Common - Song Position Pointer	LSB	MSB
1111□□11 (F3)	System Common - Song Select	Song Number (□-12□)	N□A
1111□1□□ (F4)	System Common - undefined	□	□
1111□1□1 (F5)	System Common - undefined	□	□
1111□11□ (F□)	System Common - Tune Request	N□A	N□A
1111□111 (F□)	System Common - End of System Exclusive	N□A	N□A
11111□□□ (F□)	System Real Time - Timing Clock	N□A	N□A
11111□□1 (F□)	System Real Time - undefined	N□A	N□A
11111□1□ (FA)	System Real Time - Start	N□A	N□A
11111□11 (FB)	System Real Time - Continue	N□A	N□A
111111□□ (FC)	System Real Time - Stop	N□A	N□A
111111□1 (FD)	System Real Time - undefined	N□A	N□A
1111111□ (FE)	System Real Time - Active Sensing	N□A	N□A
11111111 (FF)	System Real Time - Reset	N□A	N□A

□□ Note: System Exclusive (data dump) 2nd byte □ □ endor ID followed by more data bytes and ending with EOX (F□).

Appendix B - MIDI Pitch Value Table

Summary of MIDI Note Numbers for Different Octaves
(adapted from “MIDI by the Numbers” by D. Valenti - Electronic Musician 2[illegible])

Octave Number	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127	-	-	-	-

(Middle C is C in octave 5).

Appendix C - A Brief Description of the MIDI File Format

The MIDI file format is a standard defined by the MIDI association for transfer of MIDI data between devices. The file follows a ‘chunked’ format, with each chunk consisting of a four byte chunk identifier, that identifies the contents of the next chunk, a four byte field giving the length of the chunk in bytes (excluding the identifier and length field), and then the chunk data itself.

Currently the MIDI file format supports two chunk types. These are:

- **MHdr** - This specifies the header chunk of a standard MIDI file. This chunk must be the first chunk in a MIDI file, and contains formatting and timing data for the whole file. There may be only one header chunk in any MIDI file.

- **MTrk** - This specifies a track chunk. Track chunks contain the events and meta-events that make up a MIDI track. A MIDI file may contain an arbitrary number of tracks. However most commercial sequencer are limited to handling at most □4 tracks, and this is probably more than is necessary for almost all applications.

The MIDI file format specifies that chunks with unknown identifiers should be skipped over when reading in MIDI data from a MIDI file. This allows for the possibility of encapsulating proprietary data within a MIDI file by using non-standard chunk identifiers. Figure C-1 gives a diagrammatic representation of a MIDI file.

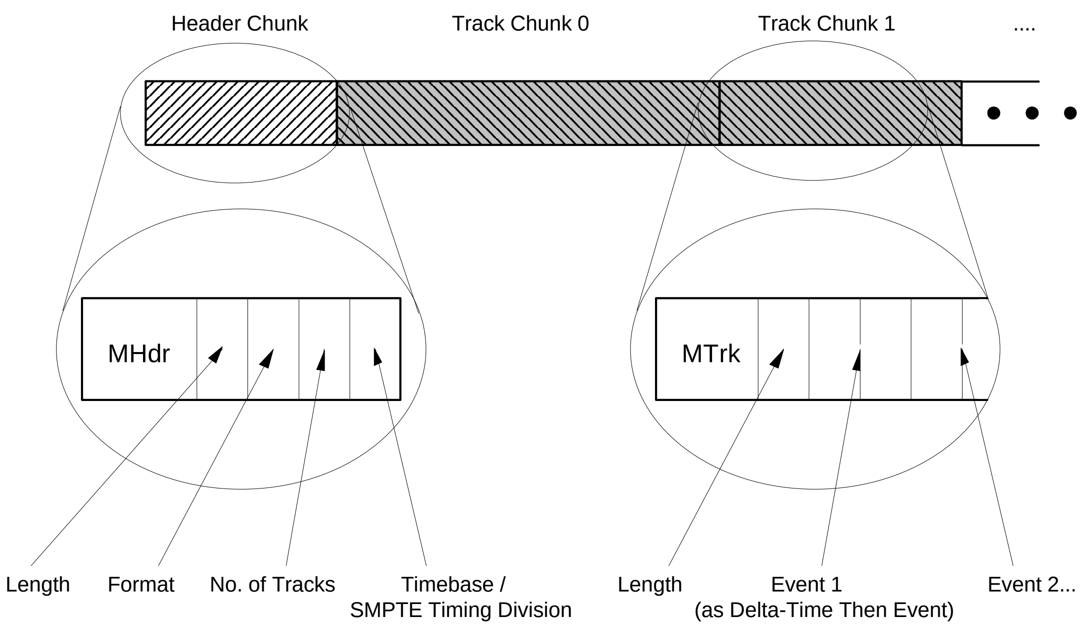


Figure C-1 The MIDI File Format

The header chunk contains the following fields:

- **Chunk Identifier** - containing the string ‘MHdr’.

- **Length Field** - always □ bytes.

- **Format** - two byte field indicating the format style of the MIDI file. This defines the relationship between the MIDI tracks in the file. The possible formats are:

  1. Single Track File.

2. Simultaneous Multi-Track File.
3. Sequential Multi-Track File.

- **No. Of Tracks** - The number of tracks in the file (a 16-bit value).
- **Timebase** - The timebase for the track, indicating the division of a crotchet that a unit value time in the MIDI file would represent (typically $^{1}/_{4\square\square}$ of a crotchet). If this value is negative then it instead represents an SMPTE timing resolution and time values in the file are stored in SMPTE format.

Track chunks contain an arbitrary number of events. The format of a track chunk is as follows:

- **Chunk Identifier** - containing the string 'MTrk'.
- **Length Field** - the number of bytes in the chunk, excluding the identifier and length field.
- A stream of events, each of which is preceded by a delta time value indicating the time difference between that event and the previous event on the track. All standard MIDI events are allowed, except for the system real-time messages.

The file format uses the status byte FF_{hex} as an escape code to allow a number of file meta-events to be included to enhance the protocol. Details of these meta-events along with a full explanation of the file format can be found in the document 'MIDI File Format Specification 1.0' published by the MIDI Manufacturers Association.