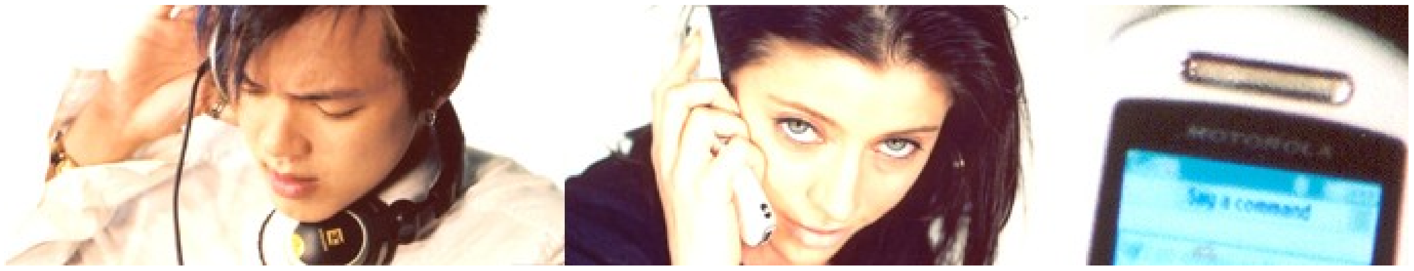

AudioINSIDE

from
Sonic Network, Inc.

JET

Programming Manual

Revision: 0.2
Date: 30-Jan-07

Revision History

Rev	Date	By	Notes
0.1	03-Nov-06	dls	Draft specification
0.2	12-Jan-07	dls	Updates based on implementation
0.3	30-Jan-07	dls	Added JCOP and JAPP to file format

Abstract

JET is an interactive music library implemented as a middleware layer on top of the EAS audio library. JET handles the complexity of managing multiple MIDI streams and sound libraries, providing a simple API for controlling audio in an application that requires a high degree of interactivity. JET allows content authors to develop content using standard MIDI software tools. A simple post-processing tool combines the content into JET compatible content files for use in the JET environment.

Nomenclature

It is important to use a common set of terms to minimize confusion. Since JET uses MIDI in a unique way, normal industry terms may not always suffice. Here are the definition of terms as they are used in this document:

Channel: MIDI data associated with a specific instrument. Standard MIDI allows for 16 channels of MIDI data each of which are typically associated with a specific instrument.

Controller: A MIDI event consisting of a channel number, controller number, and a controller value. The MIDI spec associates many controller numbers with specific functions, such as volume, expression, sustain pedal, etc. JET also uses controller events as a means of embedding special control information in a MIDI sequence to provide for audio synchronization.

Segment: A musical section such as a chorus or verse that is a component of the overall composition. In JET, a segment can be an entire MIDI file or a derived from a portion of a MIDI file.

SMF-0: Standard MIDI File Type 0, a MIDI file that contains a single track, but may be made up of multiple channels of MIDI data.

SMF-1: Standard MIDI File Type 1, a MIDI file that contains a one more tracks, and each track may in turn be made up of one or more channels of MIDI data. By convention, each channel is stored on a separate track in an SMF-1 file. However, it is possible to have multiple MIDI channels on a single track, or multiple tracks that contain data for the same MIDI channel.

Track: A track is a timed sequence of MIDI events consisting of one or more channels of MIDI data.

JET Operation

JET supports a flexible music format that can be used to create extended musical sequences with a minimal amount of data. A musical composition is broken up into segments that can be sequenced to create a longer piece. The sequencing can be fixed at the time the music file is authored, or it can be created dynamically under program control.

Linear Music Example

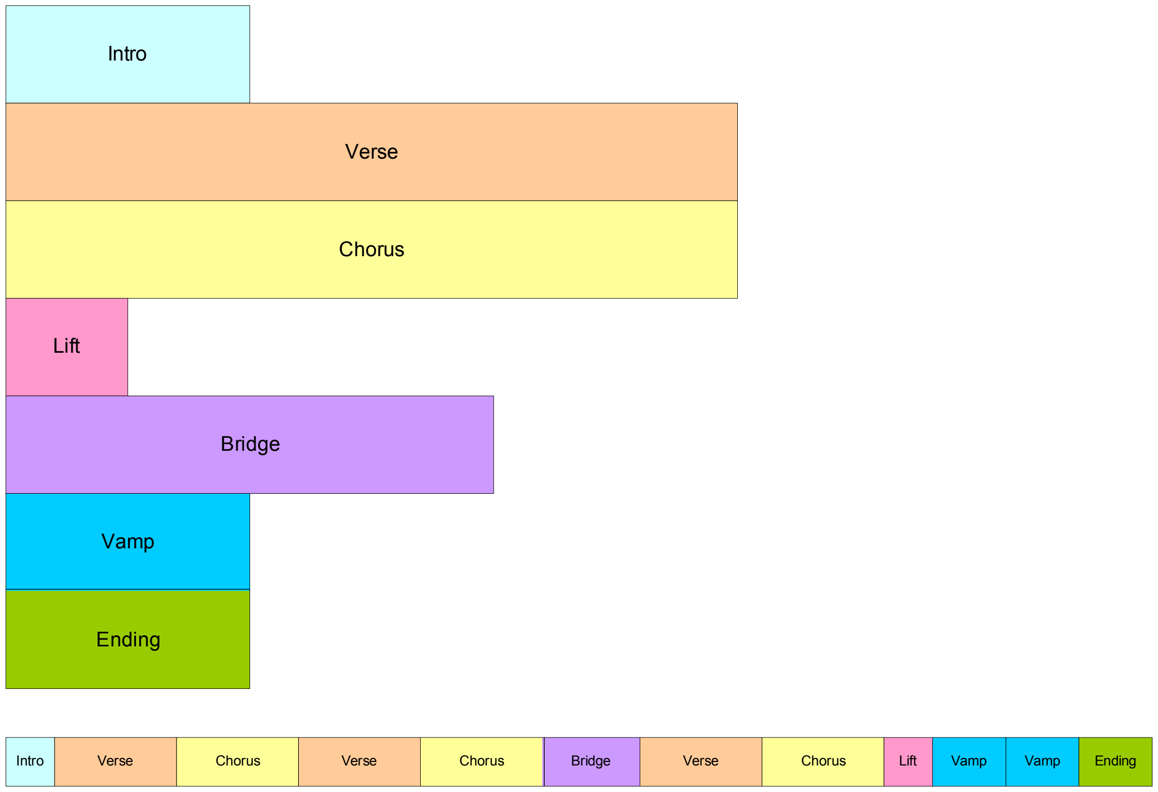

Figure 1: Linear Music Piece

This diagram shows how musical segments are stored. Each segment is authored as a separate MIDI file. A post-processing tool combines the files into a single container file. Each segment can contain alternate music tracks that can be muted or un-muted to create additional interest. An example might be a brass accent in the chorus that is played only the last time through. Also, segments can be transposed up or down.

The bottom part of the diagram shows how the musical segments can be recombined to create a linear music piece. In this example, the bridge might end with a half-step key modulation and the remaining segments could be transposed up a half-step to match.

Non-linear Music Example

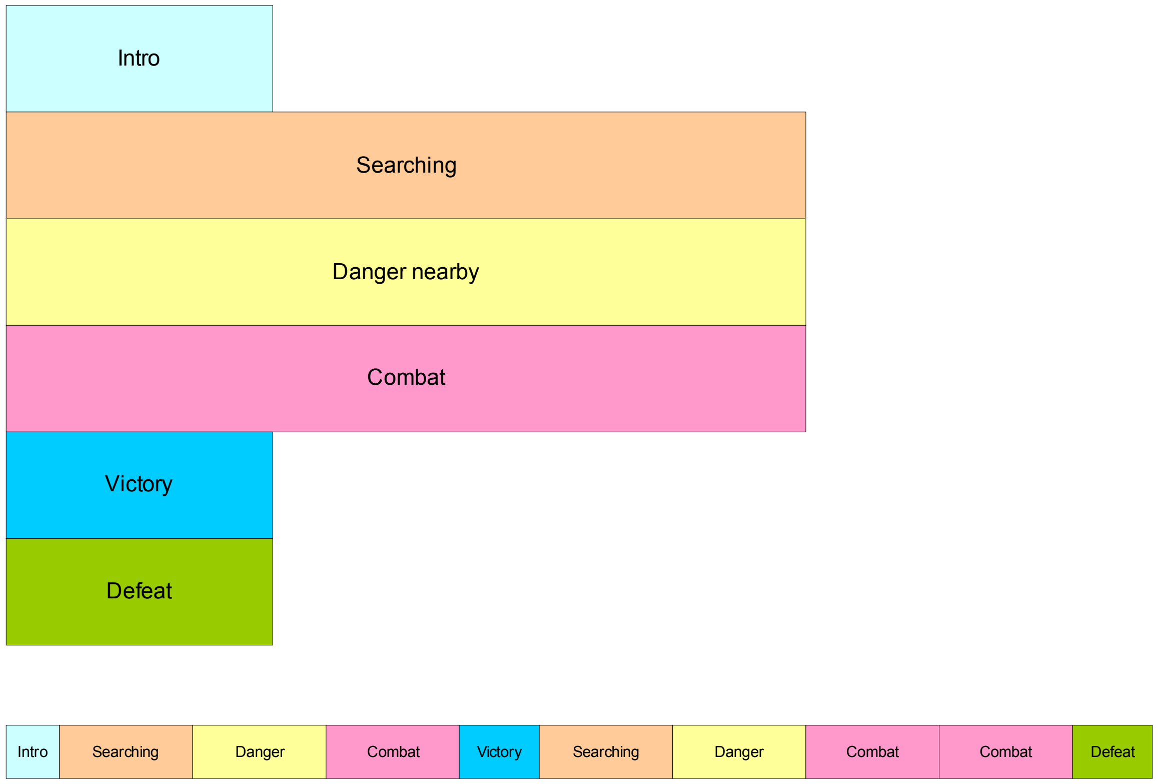

Figure 2: Non-linear music piece

In this diagram, we see a non-linear music piece. The scenario is a first-person-shooter (FPS) and JET is providing the background music. The intro plays as the level is loading and then transitions under program control to the Searching segment. This segment is repeated indefinitely, perhaps with small variations (using the mute/un-mute feature) until activity in the game dictates a change.

As the player nears a monster lair, the program starts a synchronized transition to the Danger segment, increasing the tension level in the audio. As the player draws closer to the lair, additional tracks are un-muted to increase the tension.

As the player enters into combat with the monster, the program starts a synchronized transition to the Combat segment. The segment repeats indefinitely as the combat continues. A Bonus Hit temporarily un-mutes a decorative track that notifies the player of a successful attack, and similarly, another track is temporarily un-muted to signify when the player receives Special Damage.

At the end of combat, the music transitions to a victory or defeat segment based on the outcome of battle.

Mute/Un-mute Synchronization

JET can also synchronize the muting and un-muting of tracks to events in the music. For example, in the FPS game, it would probably be desirable to place the musical events relating to bonuses and damage as close to the actual game event as possible. However, simply un-muting a track at the moment the game event occurs might result in a music clip starting in the middle. Alternatively, a clip could be started from the beginning, but then it wouldn't be synchronized with the other music tracks.

However, with the JET sync engine, a clip can be started at the next opportune moment and maintain synchronization. This can be accomplished by placing a number of short music clips on a decorative track. A MIDI event in the stream signifies the start of a clip and a second event signifies the end of a clip. When the application calls the JET clip function, the next clip in the track is allowed to play fully synchronized to the music. Optionally, the track can be automatically muted by a second MIDI event.

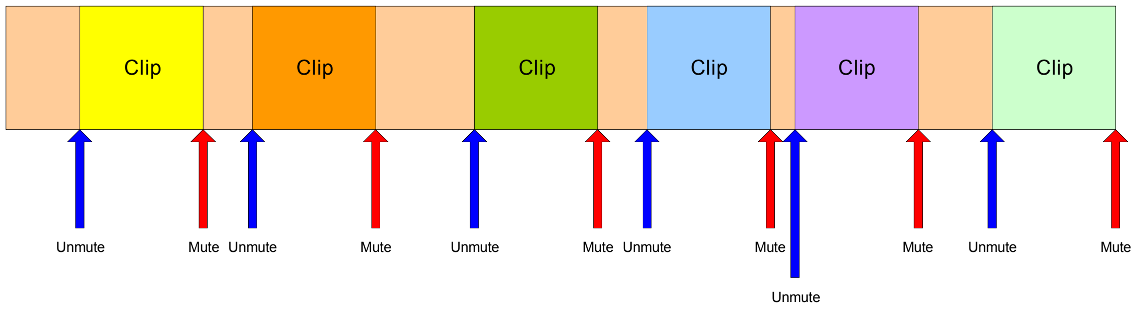

Figure 3: Synchronized Mute/Unmute

Audio Synchronization

JET provides an audio synchronization API that allows game play to be synchronized to events in the audio. The mechanism relies on data embedded in the MIDI file at the time the content is authored. When the JET engine senses an event during playback it generates a callback into the application program. The timing of the callback can be adjusted to compensate for any latency in the audio playback system so that audio and video can be synchronized. The diagram below shows an example of a simple music game that involves pressing the left and right arrows in time with the music.

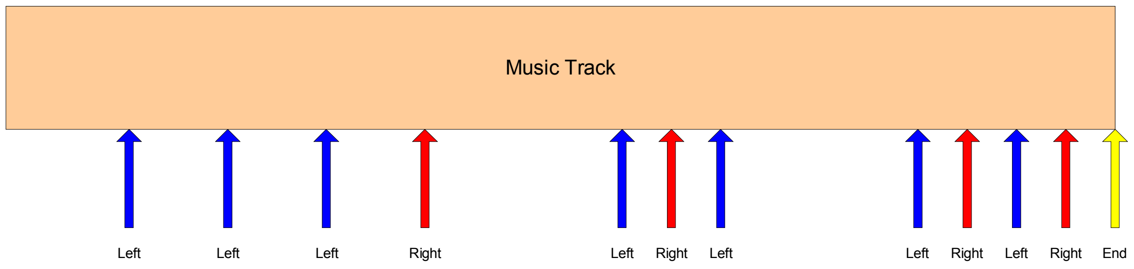

Figure 4: Music Game with Synchronization

The arrows represent events in the music sequence where game events need to be synchronized. In this case, the blue arrow represents a time where the player is supposed to press the left button, and the red arrow is for the right button. The yellow arrow tells the game engine that the sequence is complete. The player is allowed a certain time window before and after the event to press the appropriate key.

If an event is received and the player has not pressed a button, a timer is set to half the length of the window. If the player presses the button before the timer expires, the game registers a success, and if not, the game registers a failure.

If the player presses the button before the event is received, a timer is set to half the length of the window. If an event is received before the timer expires, the game registers a success, and if not, the game registers a failure. Game play might also include bonuses for getting close to the timing of the actual event.

Operational Details

JET uses the standard EAS library calls to manage multiple MIDI streams that are synchronized to sound like a seamless audio track. JET requires the use of the dynamic memory model, i.e. support for malloc() and free() memory allocation functions or their equivalent. JET also requires the DLS parser and synthesizer module to support custom instruments in JET content files.

JET uses standard MIDI events for audio synchronization. This simplifies the authoring process by allowing content authors to use their favorite tools for developing content. After the content has been developed, a simple post-processing tool pulls the content together into a JET compatible content file.

Synchronization Events

JET uses MIDI controller events to synchronize audio. The controllers used by JET are among those not defined for specific use by the MIDI specification. The specific controller definitions are as follows:

Controllers 80-83	Reserved for use by application
Controller 102	JET event marker
Controller 103	JET clip marker
Controllers 104-119	Reserved for future use

Controllers 80-83 – Application Controllers

The application may use controllers in this range for its own purposes. When a controller in this range is encountered, the event is entered into an event queue that can be queried by the application. Some possible uses include synchronizing video events with audio and marking a point in a MIDI segment to queue up the next segment. The range of controllers monitored by the application can be modified by the application during initialization.

Controller 102 – JET Event Marker

Controller 102 is reserved for marking events in the MIDI streams that are specific to JET functionality. Currently, the only defined value is 0, which marks the end of a segment for timing purposes.

Normally, JET starts playback of the next segment (or repeats the current segment) when the MIDI end-of-track meta-event is encountered. Some MIDI authoring tools make it difficult to place the end-of-track marker accurately, resulting in synchronization problems when segments are joined together.

To avoid this problem, the author can place a JET end-of-segment marker (controller=102, value=0) at the point where the segment is to be looped. When the end-of-segment marker is encountered, the next segment will be triggered, or if the current segment is looped, playback will resume at the start of the segment.

The end-of-segment marker can also be used to allow for completion of a musical figure beyond the end of measure that marks the start of the next segment. For example, the content author might create a 4-bar segment with a drum fill that ends on beat 1 of the 5th bar – a bar beyond the natural end of the segment. By placing an end-of-segment marker at the end of the 4th bar, the next segment will be triggered, but the drum fill will continue in parallel with the next segment providing musical continuity.

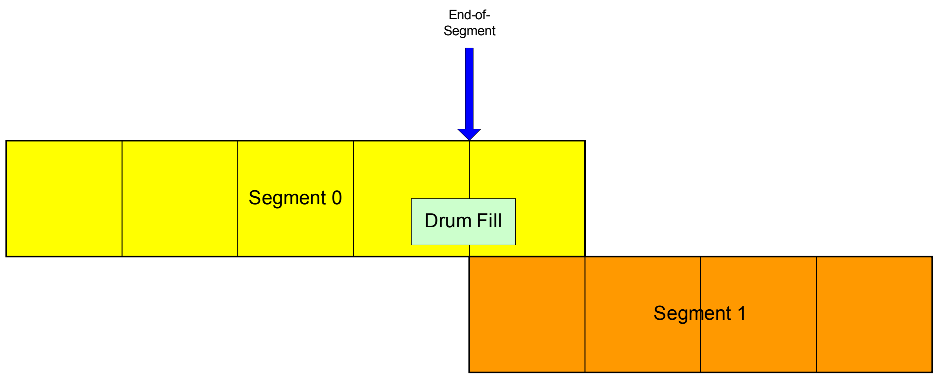

Figure 5: End-of-segment Marker

Controller 103 – JET Clip Marker

Controller 103 is reserved for marking clips in a MIDI track that can be triggered by the JET_TriggerClip API call. The clip ID is encoded in the low 6 bits of the controller value. Bit 6 is set to one to indicate the start of a clip, and set to zero to indicate the end of a clip.

For example, to identify a clip with a clip ID of 1, the author inserts a MIDI controller event with controller=103 and value=65 at the start of the clip and another event with controller=103 and value=1 at the end of the clip. When the JET_TriggerClip() function is called with a clip ID of 1, the track will be un-muted when the controller value 65 is encountered and muted again when the controller value 1 is encountered.

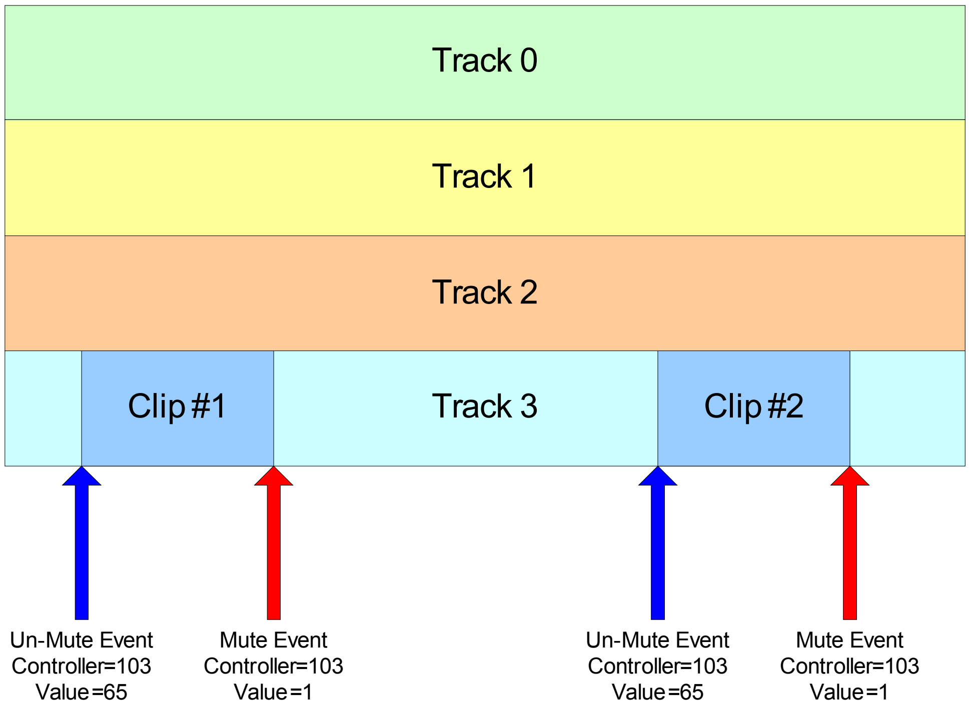

Figure 6: Synchronized Clip

In the figure above, if the JET_TriggerClip() function is called prior to the first controller event, Track 3 will be un-muted when the first controller event occurs, the first clip will play, and the track will be muted when the second controller event occurs. If the JET_TriggerClip() function is called after the first controller event has occurred, Track 3 will be un-muted when the third controller event occurs, the second clip will play, and the track will be muted again when the fourth controller event occurs.

Note: Normally, the track containing the clip is muted by the application when the segment is initially queued by the call to JET_QueueSegment(). If it is not muted, the clip will always play until Jet_TriggerClip() has been called with the clip ID.

Creating JET Content

JET uses standard MIDI files and DLS files that can be created with commercially available content tools as its source material. These source files are then bundled into a package file suitable for JET using a Python script called “jetfile.py”.

Creating MIDI Files

JET can use both type 0 (SMF-0) and type 1 (SMF-1) MIDI files as source material. However, SMF-0 files are typically larger than the same file in SMF-1 format and it is not possible to take advantage of clip automation (synchronized muting and un-muting of tracks) using SMF-0 files. Therefore, it is generally advisable to use SMF-1 files.

Jetfile.py

The jetfile.py tool uses a simple text based configuration file for describing the content in the JET file. To create a JET content file from source material, simply include the name of the configuration file on the jetfile.py command line:

```
     jetfile.py my_tune.jcfg
```

Here is a sample configuration file. It uses a fairly standard format with sections denoted in brackets, e.g. “[section] “ followed by elements that belong to that section. Comment lines begin with a semi-colon and are ignored by the file processor.

```
;this is a comment line

;Define output file
[output]
filename=my_tune.jet
chase_controllers=true
delete_empty_tracks=false
copyright=(c) Copyright 2007, Sonic Network, Inc.
app_data=my_app_data.bin

;Intro segment
[segment0]
filename=my_tune.mid
start=0:0:0
length=4:0:0
quantize=3

;Verse segment
[segment1]
filename=my_tune.mid
start=4:0:0
length=8:0:0
quantize=3

;Chorus segment
[segment2]
filename=my_tune.mid
start=12:0:0
length=8:0:0
quantize=3
clip0=0,6,0,12:0:0,14:0:0
clip1=0,6,0,16:0:0,18:0:0
```

```
;Ending segment
[segment3]
filename=my_tune.mid
start=20:0:0
length=4:0:0
quantize=3

;DLS libraries
[libraries]
lib0=my_sounds.dls
```

This configuration file pulls source material from an SMF-1 file called “my_tune.mid” and writes it to a file called “my_tune.jet”. The content consists of 4 segments denoted in the comments as “Intro, “Verse”, “Chorus”, and “Ending”.

Time Format and Length Format

Times and segment lengths in the configuration file are specified in the format measures:beats:ticks where 0:0:0 is the start of the file. There are 4 beats per measure (jetfile.py currently does not respond to the SMF meter meta-event). The number of ticks per beat is specified in the MIDI file itself (sometimes referred to as “parts per quarter note” or PPQN) and can typically be set in the authoring tool.

For example, an 8-bar segment that starts on the 5th bar has a start time of 4:0:0, and end time of 12:0:0, and a length of 8:0:0.

Quantization

Nearly all MIDI compositions that are created through performance (e.g. played in via a MIDI controller) have some variability in the timing of notes. By convention, beats always fall on tick 0, but a performance may have notes falling slightly ahead of or behind the beat.

For example, if a given source file has a PPQN of 120, a note that is supposed to fall on the first beat of the 8th measure (i.e. 8:0:0) may actually fall slightly ahead of the beat, at 7:3:119 for example. If the file is segmented at 8:0:0, the note at 7:3:119 will be placed in the previous segment. For a piece that is linear, this may not be an issue, but if the segments are linked in non-linear fashion, or if repeats are used, this may cause a discontinuity in the music.

To assist with these kinds of marginal timing problems, a quantization feature has been included. A quantization window can be specified where any notes that fall within the window on a segment boundary will be moved to the following section. In the example, a quantization window of 1 or more would result in moving the note at 7:3:119 to 8:0:0 so that the note falls in the proper segment. Similarly, if the same quantization window is specified for the previous segment, that segment would not include the note.

[output]

The [output] section is a required element that describes the output file, where the final packaged JET content is written. The supported entries in this section are:

```
filename=<file-spec>
chase_controllers=<true/false>
omit_empty_tracks=<true/false>
copyright=<copyright string>
app_data=<file-spec>
```

The *filename* element is a required entry and sets the path to the output file. The output file will be created if it does not exist and will be overwritten if it does exist. This is where the file that will be opened by the JET_OpenFile() function.

The *chase_controllers* entry is optional and if omitted defaults to true. If true, the MIDI file processor will "chase" the value of program changes, supported continuous controllers, RPN's, and channel pressure from the start of the file to the point where MIDI data is extracted for a segment or at the start of a clip. When the output file is written, MIDI events are inserted at the beginning of a segment or clip to bring the state of all these values to the same point it would be if the MIDI file were played sequentially to that point.

The *omit_empty_tracks* element is optional and if omitted, defaults to false. If true, any tracks that do not contain MIDI notes, program changes, or controller events will be omitted from the final output file (the source material is untouched). Note that if a track is omitted, any tracks that follow it in the file will have lower track numbers which in turn will affect the track values reported in the application event queue.

The *copyright* element is optional. Any ASCII alphanumeric string is acceptable on this line and will be copied into a JCOP chunk in the JET content file. The string is zero-terminated and may have an additional zero pad byte to make the length divisible by two.

The *app_data* element is optional. If included, it should be path to a file that contains application specific data. The data in the file is copied into a JAPP chunk that can be retrieved through the JET_GetAppData() function. If the file size is odd, the data will be padded with a zero to make the length divisible by two.

[segmentx]

The segment sections describe the source and processing options for JET segments. The segments can be defined in any order, but there must be a segment0 and the sequence numbers must be contiguous regardless of order (i.e. segment2, segment3, segment1, segment0 is OK, but segment3, segment1, segment0 is not).

The supported entries are as follows:

```
    filename=<file-spec>
    start=<start-time>
    end=<end-time>
    length=<end_time>
    quantize=<ticks>
    clipx=<ID>,<track#>,<channel#>,<start-time>,<end-time>
    end_marker=<track#>,<channel#>,<marker_time>
```

The *filename* element is a required entry and sets the path to the source MIDI file for the segment.

The *start* element is optional and if not specified, the default is 0:0:0.

The *end* element is optional and if not specified, the default is the end of the source file, i.e. the time of the last end-of-track meta-event in the source file. This option may not be used if the *length* element is present.

The *length* element is optional, and if not specified, the length is determined by the time of the last end-of-track meta-event in the source file. This option may not be used if the *end* element is present.

The *quantize* element is optional and defaults to 0 if omitted. This value sets a window size in ticks for the breaks in a segment when notes are extracted from a larger file. See the section on Quantization for further detail on the operation of this parameter.

The *clipx* element specifies a triggerable clip within a segment. The *x* portion of is a serial number starting with 0, i.e. the first clip is specified as *clip0*, the second as *clip1*, etc. The ID identifies the clip to the JET engine (i.e. the clipID in the JET_TriggerClip() function). Multiple clips can share the same clipID provided that they do no overlap in the segment. The track number is the track on which the clip event markers are to be placed. The channel number is the MIDI channel of the event marker. The start- and end-times are the points in the segment where the event markers should be placed. Note that these times are relative to the start of the source file and not the segment itself.

The *end_marker* element is optional. If specified, a JET end-of-segment marker is placed at the specified time. The track number and channel number specify which track and channel to place the marker. JET will respond to a marker on any track or channel in the segment. The application may use the track and channel number to encode additional information that can be retrieved in the application event queue.

[libraries]

The [libraries] section describes the custom DLS sound sets used in the JET file. It is not necessary to have a DLS library and the entire libraries section can be omitted if no custom sounds are included. Library numbers must be sequential and start with 0, e.g. “lib0”, “lib1”, “lib2”, etc.

JET Programming

The JET library builds on functionality in the EAS library. It is assumed that the reader is familiar with EAS and has implemented basic EAS audio functionality in the application. Specifically, the application must first initialize EAS by calling EAS_Init() and must call EAS_Render() at appropriate times to render audio and stream it to the audio hardware. JET also requires the use of the dynamic memory model which uses malloc() and free() or functional equivalents.

Most JET function calls return an EAS_RESULT type which should be checked against the EAS_SUCCESS return code. Most failures are not fatal, i.e. they will not put the library in a state where it must be re-initialized. However, some failures such as memory allocation or file open/read errors will likely result in the specific open content failing to render.

JET Application Initialization

The JET library is initialized by the JET_Init() function. The application must first call EAS_Init() and then pass the EAS data handle returned by EAS_Init() to the JET_Init() function. Currently, only a single JET application can be active at a time.

The JET_Init function takes 3 arguments: The first is the EAS data handle. The second is a pointer to a configuration structure S_JET_CONFIG and the third is the size of the configuration structure. For most applications, it is sufficient to pass a NULL pointer and size 0 for the configuration data.

However, if desired, the configuration can be modified to allow the application to monitor MIDI events outside the normal range of controllers allocated for JET application events. In this case, a configuration structure should be allocated and the data fields initialized with the appropriate values with the low and high controller numbers to be monitored. The size field should be the sizeof() of the data structure. This is to allow for future enhancement of the configuration data while maintaining compatibility.

JET Application Termination

When the JET application terminates, it should call JET_Shutdown() to release the resources allocated by the JET engine. If the application has no other use for the EAS library, it should also call EAS_Shutdown().

JET Audio Processing

To start the JET engine, the content must first be opened with the JET_OpenFile() function. Just as with EAS_OpenFile(), the file locator is an opaque value that is passed to the EAS_HWOpenFile() function. It can either be a pointer to a filename, or a pointer to an in-memory object, depending on the user implementation of file I/O in the eas_host.c or eas_hostmm.c module. Only a single JET content file can be opened at a time.

Once the JET file is opened, the application can begin queuing up segments for playback by calling the JET_QueueSegment() function. Generally, it is advisable to keep a minimum of two segments queued at all times: the currently playing segment plus an additional segment that is ready to start playing when the current segment finishes. However, with proper programming, it is possible to queue up segments using a “just-in-time” technique. This technique typically involves careful placement of application controller events near the end of a segment so that the application is informed when a segment is about to end.

After the segment(s) are queued up, playback can begin. By default, the segments are initialized in a paused state. To start playback, call the JET_Play() function. Playback can be paused again by calling the JET_Pause() function. Once initiated, playback will continue as long as the application continues to queue up new segments before all the segments in the queue are exhausted.

The JET_Status() function can be used to monitor progress. It returns the number of segments queued, repeat count, current segment ID, and play status. By monitor the number of segments queued, the application can determine when it needs to queue another segment and when playback has completed.

When playback has completed and the application is finished with the contents of the currently open file, the application should call JET_CloseFile() to close the file and release any resources associated with the file.

JET_Init

```
EAS_PUBLIC EAS_RESULT JET_Init (EAS_DATA_HANDLE easHandle,
S_JET_CONFIG *pConfig, EAS_INT configSize)
```

Initializes JET library for use by application. Most application should simply pass a NULL for pConfig and 0 for configSize, which means that only controller events in the application range (80-83) will end up in the application event queue. If desired, the application can instantiate an S_JET_CONFIG data structure and set the controller range to a different range. In this case, the configSize parameter should be set to sizeof(S_JET_CONFIG).

JET_Shutdown

```
EAS_PUBLIC EAS_RESULT JET_Shutdown (EAS_DATA_HANDLE
easHandle)
```

Releases resources used by the JET library. The application should call this function when it is no longer using the JET library.

JET_ OpenFile

```
EAS_PUBLIC EAS_RESULT JET_OpenFile (EAS_DATA_HANDLE
easHandle, EAS_FILE_LOCATOR locator)
```

Opens a JET content file for playback. Content must be formatted for use by the JET library, which is typically accomplished with the jetfile.py script (see “Creating JET Content”). Only a single JET content file can be opened at a time. However, since JET can contain many MIDI files and DLS libraries, this limitation is normally not an issue.

JET_ CloseFile

```
EAS_PUBLIC EAS_RESULT JET_CloseFile (EAS_DATA_HANDLE
easHandle)
```

Closes a JET file and release the resources associated with it.

JET_ Status

```
EAS_PUBLIC EAS_RESULT JET_Status (EAS_DATA_HANDLE
easHandle, S_JET_STATUS *pStatus)
```

Returns the current JET status. The elements of the status data structure are as follows:

```
     typedef struct s_jet_status_tag
     {
          EAS_INT   currentUserID;
          EAS_INT   segmentRepeatCount;
          EAS_INT   numQueuedSegments;
          EAS_BOOL  paused;
     } S_JET_STATUS;
```

currentUserID: An 8-bit value assigned by the application.

segmentRepeatCount: Number of times left to repeat. Zero indicates no repeats, a negative number indicates an infinite number of repeats. Any positive value indicates that the segment will play n+1 times.

numQueuedSegments: Number of segments currently queued to play including the currently playing segment. A value of zero indicates that nothing is playing. Normally, the application will queue a new segment each time the value is 1 so that playback is uninterrupted.

JET_ QueueSegment

```
EAS_PUBLIC EAS_RESULT JET_QueueSegment (EAS_DATA_HANDLE
easHandle, EAS_INT segmentNum, EAS_INT libNum, EAS_INT
repeatCount, EAS_INT transpose, EAS_U32 muteFlags, EAS_U8
userID)
```

Queues up a JET MIDI segment for playback. The parameters are as follows:

segmentNum: Segment number as identified in the JET content configuration file.

libNum: The library number as specified in the JET content configuration file. Use -1 to select the standard General MIDI library.

repeatCount: The number of times this segment should repeat. Zero indicates no repeat, i.e. play only once. Any positive number indicates to play n+1 times. Set to -1 to repeat indefinitely.

transpose: The amount of pitch transposition. Set to 0 for normal playback. Range is -12 to +12.

muteFlags: Specific which MIDI tracks (not MIDI channels) should be muted during playback. These flags can be changed dynamically using the mute functions. Bit 0 = track 0, bit 1 = track 1, etc.

userID: 8-bit value specified by the application that uniquely identifies the segment. This value is returned in the JET_Status() function as well as by the application event when an event is detected in a segment. Normally, the application keeps an 8-bit value that is incremented each time a new segment is queued up. This can be used to look up any special characteristics of that track including trigger clips and mute flags.

JET_ Play

```
EAS_PUBLIC EAS_RESULT JET_Play (EAS_DATA_HANDLE easHandle)
```

Starts playback of the current segment. This function must be called once after the initial segments are queued up to start playback. It is also called after JET_Pause() to resume playback.

JET_ Pause

```
EAS_PUBLIC EAS_RESULT JET_Pause (EAS_DATA_HANDLE easHandle)
```

Pauses playback of the current segment. Call JET_Pause() to resume playback.

JET_ SetMuteFlags

```
EAS_PUBLIC EAS_RESULT JET_SetMuteFlags (EAS_DATA_HANDLE
easHandle, EAS_U32 muteFlags, EAS_BOOL sync)
```

Modifies the mute flags during playback. If the *sync* parameter is false, the mute flags are updated at the beginning of the next render. This means that any new notes or controller events will be processed during the next audio frame. If the *sync* parameter is true, the mute flags will be updated at the start of the next segment. If the segment is repeated, the flags will take effect the next time segment is repeated.

JET_ SetMuteFlag

```
EAS_PUBLIC EAS_RESULT JET_SetMuteFlag (EAS_DATA_HANDLE
easHandle, EAS_INT trackNum, EAS_BOOL muteFlag, EAS_BOOL
sync)
```

Modifies a mute flag for a single track during playback. If the *sync* parameter is false, the mute flag is updated at the beginning of the next render. This means that any new notes or controller events will be processed during the next audio frame. If the *sync* parameter is true, the mute flag will be updated at the start of the next segment. If the segment is repeated, the flag will take effect the next time segment is repeated.

JET_ TriggerClip

```
EAS_PUBLIC EAS_RESULT JET_TriggerClip (EAS_DATA_HANDLE
easHandle, EAS_INT clipID)
```

Automatically updates mute flags in sync with the JET Clip Marker (controller 103). The parameter *clipID* must be in the range of 0-63. After the call to JET_TriggerClip, when JET next encounters a controller event 103 with bits 0-5 of the value equal to *clipID* and bit 6 set to 1, it will automatically un-mute the track containing the controller event. When JET encounters the complementary controller event 103 with bits 0-5 of the value equal to *clipID* and bit 6 set to 0, it will mute the track again.

JET_ GetEvent

```
EAS_BOOL JET_GetEvent (EAS_DATA_HANDLE easHandle, EAS_U32
*pEventRaw, S_JET_EVENT *pEvent)
```

Attempts to read an event from the application event queue, return EAS_TRUE if an event is found and EAS_FALSE if not. If the application passes a valid pointer for *pEventRaw*, a 32-bit compressed event code is returned. If the application passes a valid pointer for *pEvent*, the event is parsed into the S_JET_EVENT fields. The application can pass NULL for either parameter and that variable will be ignored. Normally, the application will call JET_GetEvent() repeatedly to retrieve events until it returns EAS_FALSE.

JET_ ParseEvent

```
EAS_PUBLIC void JET_ParseEvent (EAS_U32 event, S_JET_EVENT
*pEvent)
```

Parses a 32-bit compressed event code into a data structure. The application passes the event code received from JET_GetEvent(). The parsed event data is returned in the memory pointed to by *pEvent*.

JET_GetAppData

```
EAS_RESULT JET_GetAppData (EAS_DATA_HANDLE easHandle,
EAS_I32 *pAppDataOffset, EAS_I32 *pAppDataSize)
```

Returns the offset and size of the JAPP chunk in the JET file. The application can use the file I/O functions in the eas_host module to retrieve application specific data from the file.