Jaguar Libraries

This section describes the various libraries that are included with the Jaguar development kit.

Because Atari is constantly updating and improving the Jaguar libraries and sample code, it's possible that there may be differences between the documentation and the most current release of a library. Always check the library distribution archive for one or more text files with additional or replcement documentation.

The following libraries are included:

- Jaguar Startup Code
- 3D Graphics
- BPEG Image Compression & Decompression
- Cinepak Decompression & Playback (See separate **Cinepak For Jaguar** section)
- Networking (see **Jaguar Voice Modem** section)
- Music & Sound
- Jaguar Music Driver
- EEPROM Access Library
- NV-RAM Cartridge Access Library

See also the **Sample Programs** section.

Jaguar Startup Code

Starting up a Jaguar (initializing video, the object list, etc...) is the most important thing a program must do correctly. This startup code (STARTUP.S) performs all of the program initialization correctly and **must always** be used. ***Note that modifying, reordering, or omitting any part of this startup, except those portions explicitly marked as being changeable, will likely cause your software to fail our hardware testing procedures.***

How to Use It

Link STARTUP.S first to make it the first code to be executed. Do not perform any initialization of any kind prior to running this startup code. When this code finishes it will jump to the label ***_start*** to enter your code.

What Does It Do?

Our startup performs the following steps:

1. Sets GPU and DSP Endian registers correctly.
2. Disables video refresh.
3. Sets the 68k stack pointer to the end of DRAM.
4. Initializes video registers.
5. Creates an object list as follows:

   BRANCH Object (Branches to stop object if past display area)
   BRANCH Object (Branches to stop object if prior to display area)
   BITMAP Object (Jaguar License Acknowledgement - see below)
   STOP Object

6. Installs an interrupt handler, configures VI, enables 68k video interrupts, lowers 68k IPL to allow interrupts.
7. Uses GPU routine ***gSetOLP*** to stuff OLP with pointer to object list.
8. Turns on RGB video ($6C7 in VMODE).
9. Jumps to ***_start*** (your supplied code).

As soon as your code gains control you should perform whatever other initialization tasks your code may need to allow the graphic to be on screen for a reasonable amount of time.

When you need to transfer control to your object list (for your title screen or whatever else) you should poll the variable 'ticks' for a change. At this point (vertical blank) you should switch interrupt handlers (by placing a new value at LEVEL0 $100) and change the OLP. Remember, the OLP should only be changed by the GPU (you can use our DRAM routine if the GPU isn't already running).

The Licensing Graphic

The macro *license_logo* definition at the top of STARTUP.S should be changed as necessary to indicate either the "Licensed by" or "Licensed to" graphic respectively. The "Licensed to" graphic should only be used by our subcontractors doing a port of an existing game created by a company other than Atari. The "Licensed by" graphic should be used in all other cases.

Additional Note

This collection of files should always be used as the baseline startup reference. For example, at the time of this writing, many of our other sample programs have not yet been updated to reflect some of the new things this startup does more correctly. They will be updated soon. However, whenever an update needs to be made, this startup code will always be updated first.

3D Graphics

Please note that there is nothing preventing developers from using a different 3D modeling program to create their 3D objects. However, you will have to provide your own object conversion utilities and/or 3D transformation and rendering functions.

3DS2JAG Object/Texture Conversion Utility

The utility 3DS2JAG converts an object file created with AutoCAD 3-D Studio v2.0 or v3.0 into a format that can be used with the Jaguar 3D graphics routines. For detailed information on this utility, see the **Tools** chapter.

For a full description of the 3D-Studio object data format refer to the manual "3D Studio File ToolKit: reference, publication 100672-A, December 18, 1992". As newer versions of 3D Studio are created, 3DS2JAG will have to be modified to reflect any new commands. The structure of the .3DS binary data file can be found in Chapter 2, page 7, and the Data Structure Reference, page 35-47. The data in this file is grouped into chunks, defined by a Command, Size, and Data block. See Chapter 3, pages 49-79.

Construction Of A .JAG File

Once the .3DS model has been completely parsed and assembled, the .JAG model created by the conversion utility must be assembled and output. The following is a sample of output from 3DS2JAG for a cube created in 3D Studio:

```
;*==================================================================*
;*
;*     File:          cube.JAG
;*
;*     Created From:       cube.3ds
;*
;*
;*==================================================================*

      .data
      .phrase

SEGOFFSET   EQU   $4


.include    "blit.inc"
;*==================================================================*

      .globl      data

.phrase
data:
        dc.w  8                     ;* number of Vertices
        dc.w  12                    ;* number of Faces
        dc.l  .vertlist             ;* pointer to vertices
        dc.l  .texlist              ;* pointer to texture maps
        dc.l  .tboxlist             ;* pointer to texture boxes

        ;*=============================================================*
        ;*    FACE DATA - negative values signify reversing the segment vertext pair
        ;*=============================================================*

.facelist:
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   0: Segments in Face
        dc.w  $008f                 ;* color GREEN MATTE (GOURAUD)
        dc.w  4 * 8
        dc.w  6 * 8
        dc.w  7 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   1: Segments in Face
        dc.w  $008f                 ;* color GREEN MATTE (GOURAUD)
        dc.w  4 * 8
        dc.w  5 * 8
        dc.w  6 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   2: Segments in Face
        dc.w  $00f9                 ;* color ORANGE MATTE (GOURAUD)
        dc.w  0 * 8
        dc.w  5 * 8
        dc.w  4 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   3: Segments in Face
        dc.w  $00f9                 ;* color ORANGE MATTE (GOURAUD)
        dc.w  0 * 8
        dc.w  1 * 8
        dc.w  5 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   4: Segments in Face
        dc.w  $0089                 ;* color GRAY MATTE (GOURAUD)
        dc.w  1 * 8
        dc.w  6 * 8
        dc.w  5 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   5: Segments in Face
        dc.w  $0089                 ;* color GRAY MATTE (GOURAUD)
        dc.w  1 * 8
        dc.w  2 * 8
        dc.w  6 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   6: Segments in Face
        dc.w  $00f1                 ;* color RED MATTE (GOURAUD)
        dc.w  3 * 8
        dc.w  4 * 8
        dc.w  7 * 8
        dc.l  $FFFF0000             ;* Gouraud shaded. No texture.
        dc.w  3                     ;* Face   7: Segments in Face
        dc.w  $00f1                 ;* color RED MATTE (GOURAUD)
        dc.w    3 * 8
        dc.w    0 * 8
        dc.w    4 * 8
        dc.l    $FFFF0000           ;* Gouraud shaded. No texture.
        dc.w    3                   ;* Face   8: Segments in Face
        dc.w    $00ff               ;* color YELLOW MATTE (GOURAUD)
        dc.w    2 * 8
        dc.w    7 * 8
        dc.w    6 * 8
        dc.l    $FFFF0000           ;* Gouraud shaded. No texture.
        dc.w    3                   ;* Face   9: Segments in Face
        dc.w    $00ff               ;* color YELLOW MATTE (GOURAUD)
        dc.w    2 * 8
        dc.w    3 * 8
        dc.w    7 * 8
        dc.l    $FFFF0000           ;* Gouraud shaded. No texture.
        dc.w    3                   ;* Face  10: Segments in Face
        dc.w    $0001               ;* color BLUE MATTE (GOURAUD)
        dc.w    0 * 8
        dc.w    2 * 8
        dc.w    1 * 8
        dc.l    $FFFF0000           ;* Gouraud shaded. No texture.
        dc.w    3                   ;* Face  11: Segments in Face
        dc.w    $0001               ;* color BLUE MATTE (GOURAUD)
        dc.w    0 * 8
        dc.w    3 * 8
        dc.w    2 * 8

        ;*============================================================*
        ;*    VERTEX DATA
        ;*============================================================*

.vertlist:
                ;* vertex:  0
        dc.l    $FFCF0031           ;* X |Y  (16.0,16.0)     (-49,49)
        dc.l    $FFCFDB0D           ;* Z |Nx (16.0,0.16)     (-49)
        dc.l    $24F3DB0D           ;* Ny|Nz (0.16,0.16)

                ;* vertex:  1
        dc.l    $00310031           ;* X |Y  (16.0,16.0)     (49,49)
        dc.l    $FFCF24F3           ;* Z |Nx (16.0,0.16)     (-49)
        dc.l    $24F3DB0D           ;* Ny|Nz (0.16,0.16)

                ;* vertex:  2
        dc.l    $0031FFCE           ;* X |Y  (16.0,16.0)     (49,-50)
        dc.l    $FFCF24F3           ;* Z |Nx (16.0,0.16)     (-49)
        dc.l    $DB0DDB0D           ;* Ny|Nz (0.16,0.16)

                ;* vertex:  3
        dc.l    $FFCFFFCE           ;* X |Y  (16.0,16.0)     (-49,-50)
        dc.l    $FFCFDB0D           ;* Z |Nx (16.0,0.16)     (-49)
        dc.l    $DB0DDB0D           ;* Ny|Nz (0.16,0.16)

                ;* vertex:  4
        dc.l    $FFCF0031           ;* X |Y  (16.0,16.0)     (-49,49)
        dc.l    $0032DB0D           ;* Z |Nx (16.0,0.16)     (50)
```

```
     dc.l   $24F324F3           ;* Ny|Nz (0.16,0.16)

            ;* vertex:  5
     dc.l   $00310031           ;* X |Y  (16.0,16.0)    (49,49)
     dc.l   $003224F3           ;* Z |Nx (16.0,0.16)    (50)
     dc.l   $24F324F3           ;* Ny|Nz (0.16,0.16)

            ;* vertex:  6
     dc.l   $0031FFCE           ;* X |Y  (16.0,16.0)    (49,-50)
     dc.l   $003224F3           ;* Z |Nx (16.0,0.16)    (50)
     dc.l   $DB0D24F3           ;* Ny|Nz (0.16,0.16)

            ;* vertex:  7
     dc.l   $FFCFFFCE           ;* X |Y  (16.0,16.0)    (-49,-50)
     dc.l   $0032DB0D           ;* Z |Nx (16.0,0.16)    (50)
     dc.l   $DB0D24F3           ;* Ny|Nz (0.16,0.16)

     ;* Model Size = ( 232 = 0xe8 ) bytes

.texlist:

.tboxlist:
```

See the sources for the 3D Demo program for further detail.

Transformation & Display Routines

At this time, the only documentation for the 3D transformation & display routines is contained within the comments of the actual source code itself. Please examine the 3D demo program source code for more information.

3D Demo Program

The 3D demo program demonstrates the use of the 3D object transformation & rendering routines. It shows a detailed, texture-mapped spaceship and lets you move it around using the joypad. See the more detailed description in the **Sample Programs** section.

Jaguar BPEG Image Compression & Decompression

BPEG is a version of JPEG[1] for the Jaguar. The BPEG utility and library are provided to allow you to compress bitmapped RGB graphics to a small fraction of their original size, so that they use minimal space in your Jaguar programs.

JPEG is a "lossy" compression scheme, meaning that the after being compressed and then decompressed, the picture will not be exactly identical to the original. You can fine tune the compression quality as needed to strike the most acceptible balance between image quality and compression ratio.

Note: BPEG is primarily designed for RGB-mode graphics, and the compression utility takes RGB-mode graphics files as input. However, the BPEG decompression library is capable of converting the images to CRY-mode on the fly when they are decompressed (at the cost of longer decompression times).

Note: The BPEG package replaces the JAGPEG package previously included with the Jaguar Developer's kit. The BPEG utility is easier to use, and the decompression library is faster and includes complete source code so that you can make any modifications required by your specific application.

Using the Compression Utility

The first thing you have to do is have a compressed image. Atari provides a tool in the Jaguar developer's kit that allows you to compress Targa-format[2] picture files into BPEG format. See the **Tools** chapter for information about this utility.

Let's Compress Some Images

Using the compression tools is quite simple. Included in the BPEG package is a sample program that displays two compressed pictures on the Jaguar screen. Normally, compressing the images is taken care of automatically by the MAKEFILE used by the sample program, but let's do it manually so that you are familiar with the process.

1) Move to the \JAGUAR\BPEG directory. The sample pictures FISH.TGA and PATRICK.TGA provided are located in this directory.

[1] JPEG stands for Joint Photographic Experts Group. A JPEG picture is one that has been compressed using the JPEG lossy file compression scheme.

[2] Targa is a popular image file format for 16-bit and 24-bit RGB true color graphics. If your graphics programs do not support the Targa file format, then you should investigate one of the various file format conversion utilities. HiJack Pro for Windows is available at computer stores everywhere, and the shareware program Paint Shop Pro (for MS-Windows) is available online.

2) Type in the command:

```
cbpeg -quality 25 fish.tga fish.bpg
```

We are compressing the file FISH.TGA to get the file FISH.BPG, using a quality setting of 25. The compression process will normally take just a few seconds, but of course this will vary depending on the size of the image, the quality percentage selected, and the speed of your computer.

3) Now you should have a file named FISH.BPG which is 9112 bytes, that's less than 5% the size of the original FISH.TGA file!

4) Now type in the command:

```
cbpeg -quality 75 patrick.tga patrick.bpg
```

Now we are compressing the file PATRICK.TGA to PATRICK.BPG using a quality setting of 75. This should result in a file that is 6864 bytes long (less than 4% of the original file size). Note that this picture compressed to a smaller size than FISH.TGA even though we are using a higher quality setting.

Later we will examine the sample program that displays these pictures on the Jaguar.

BPEG Decompression Routines

The BPEG.S file contains the source for the BPEG decompression routines. This file contains several flags which customize the operation of BPEG. While these flags are meant to be used at assembly time, you may wish to modify the code so that they may be set at runtime. The source is provided so that this sort of program-specific modification can be made.

The flags CRY15, CRY16, RGB15, RGB16, RGB32 defined at the top of BPEG.S control the output mode of the decompressor. One, and only one, of these flags must be set to TRUE (non-zero) and the others set to FALSE (zero).

The BPEG functions are accessed via two 68000-based routines which call the GPU-based decompression code with the proper parameters. The decoding steps are:

1) Call ***BPEGInit*** (no input or output parameters).

2) Call ***BPEGDecode***

Input:
A0.l is the BPEG stream pointer
A1.l is the output buffer address
D0.l is the output buffer line width (in bytes)

Output:

D0 = 0 (no problem)/ 1 (bad format)

3) Test *BPEGStatus* (long). Possible values are:

-1 (decoding)
0 (finished)
2 (decoding aborted, Huffman error)

If you want to decode another image, just go to step 2.

What Exactly These Functions Do?

BPEGInit copies the GPU code in the GPU RAM, without using the blitter. You can change this if the blitter is not used at this moment.

BPEGDecode sets some variables in the GPU, and run it. The GPU uses (corrupts) ALL REGISTERS FROM BOTH BANKS, and almost all GPU memory (the exact amount of memory used depends on the chosen output mode).

If you require that some GPU registers be left alone (like for interrupt processing), then you will have to edit the BPEG.S source file so that it leaves a few registers free. However, recognize that this will result in slower decode times.

Note: If you're decoding an image in CRY15/CRY16 modes, you must have the 32Kb RGB->CRY conversion table, and declare the GLOBAL symbol *CRYTable*, at the start of the table. This table is included in the file RGB2CRY.S.

Tip: Don't forget that cartridge access is slower than RAM access. It's a good idea to copy some of the BPEG tables into RAM before running the decoder, for ultimate speed.

TESTBPEG Sample Program

TESTBPEG is a sample program that demonstrates how to take the files created with the BPEG tool and use them. This sample program is similar to many of the other sample programs for the most part, except that it sets up the video a bit differently with a 16-bit RGB mode instead of 16-bit CRY, and a creates a 16-bit RGB bitmap object instead of an 8-bit palette-based object. This is, of course, to accomodate the JPEG pictures which the program displays.

Do not use this sample program as a demonstration of anything other than how to use the BPEG library.

The interesting parts of this are in the TEST.S file, which sets up and calls the BPEG routines to decompress the pictures. It switches back and forth between two different pictures which were compressed with different quality settings. One of the pictures is 75% quality, the other is set to only 25% but still manages to look reasonably decent.

Excerpt from TEST.S

Below are some annotated excerpts from the TEST.S file of the TESTBPEG sample program.

First we must declare the external references to the pictures and decompression code that will be added in at link time.

```
        .extern BPEGInit        ; Copy over GPU code into GPU RAM
        .extern BPEGDecode      ; Execute decode routines
        .extern BPEGStatus      ; semaphore for "finished decoding" status

        .extern fish_jpg        ; picture #1
        .extern pat_jpg         ; picture #2
```

Here's the code to actually call the BPEG routine to decompress and display one image, wait for it to finish decoding, and then go onto the next image. Note that this simple example does not check for errors returned by the *BPEGDecode* function.

```
        bsr     BPEGInit                ; copy over GPU code

.show_fish:
        lea     fish_jpg,a0             ; Address of compressed picture data
        lea     bitmap_addr,a1          ; Get destination address
        move.l  #((WIDTH*DEPTH)/8),d0   ; Width of destination bitmap, in bytes
        bsr     BPEGDecode              ; Decode image

.wait_fish:
        tst.l   BPEGStatus              ; Wait for decompression to finish
        bmi.s   .wait_fish              ; before continuing...

        lea     pat_jpg,a0              ; Address of compressed picture data
        lea     bitmap_addr,a1          ; Get destination address
        move.l  #((WIDTH*DEPTH)/8),d0   ; Width of destination bitmap, in bytes
        bsr     BPEGDecode              ; Decode image

.wait_patrick:
        tst.l   BPEGStatus              ; Wait for decompression to finish
        bmi.s   .wait_patrick           ; before continuing...

        bra     .show_fish              ; Loop forever through both pictures
```

Note that the pictures are switched back and forth as quickly as the decompression code can spit them out. Also take a look at the MAKEFILE, which shows how you can specify a command input file for the ALN linker to get around the 128-byte MSDOS commandline length limitation. The "-c testbpeg.lnk" option specifies that the linker should read input from the file TESTJPG.LNK, which in turn contains additional commands for the linker.

From the MAKEFILE for TESTBPEG:

```
testjpg.abs: $(OBJ) dehuff.dat
     aln $(ALNFLAGS) $(OBJ) -c testjpg.lnk
```

The contents of the TESTJPG.LNK file shows how the .JAG picture files are included in the program, as well as the DEJAG routine's .BIN file and .DAT files.

Contents of the TESTJPG.LNK file:

```
-i fish.bpg fish_jpg
-i patrick.bpg pat_jpg
```

The "-i" option tells ALN to include the file specified by the next parameter, and to create a label at that address as specified by the next parameter after that. Therefore, the first line of this file tells ALN to include the file FISH.BPG (the BPEG-compressed version of FISH.TGA) and to create a label "*fish_jpg*" at the address where the data from this file ends up in the resulting file. Then our test program refers to " *fish_jpg* " when it decompresses the picture (as shown in the sample code above).

Cinepak Video Decompession & Playback

The Cinepak Video Decompression & Playback libraries, related sample programs, and utilities are discussed in a separate chapter. Please see the chapter **Cinepak For Jaguar** for more information.

Networking

There are two basic types of networking that can be used with the Atari Jaguar. The first type is a local area network (LAN) with multiple Jaguar consoles in the same room or building connected via the asynchronous serial port. This is similar to a computer LAN setup. The second type of network is two Jaguar consoles connected to each other over the telephone lines via the Jaguar modem.

At this time, the specifications for LAN-style networking is still in development within Atari. The specification for **The Jaguar Voice Modem** is given in its own section..

Music & Sound

The Jaguar Synth

Sound in Jaguar is produced by the requires a synthesizer program running in the Digital Signal Processor (DSP) in Jerry. This document describes the lowest level interface to one such program, FULSYN, aka “the Jaguar Synth”.

The Jaguar Synth is voice table driven. The main loop checks a voice table to see which voices are turned on, and then it calls the appropriate module for each active voice. There are twelve synthesis modules:

- 6 Sampler modules.
- 1 Wave Table module.
- 3 FM Modules.
- 2 Envelope-based Waveform modules

All of the modules can be placed at a stereo pan location.

Sampler Modules

The Sampler modules allow either 8-bit or 16-bit signed sample data, as well as a special compressed format where 16-bit data has been compressed 2:1[3]. This compression is slightly lossy. All Samplers use data that is not in Jerry's internal RAM. All samplers also support pitch shifting. The Samplers have the ability to loop within the sample so that long sustains may be achieved without using too much memory. The parameters for the Sampler modules are:

- Pitch
- Pointer to sample data
- Size of loop
- Envelope Information (optional)
- Loop flag/Volume
- End of loop
- Pan value

FM Modules

The FM modules are simple to understand but produce a wide variety of sounds. In simple terms, an FM synthesizer takes a 128 sample waveform where each sample consists of a 16 bit signed integer sign extended to a 32 bit long. The synth then modulates the frequency according to another waveform (built like the first). The simple FM parameters are:

- Pitch
- Pointer to Sample Waveform
- Frequency of modulation
- Pan Value
- Volume
- Pointer to Modulating Waveform
- Depth of modulation

[3] This compression is done by the SNDCOMP utility.

The complex FM module adds:

- Pointer to Modulator of Modulation
- Depth of modulation of frequency
- Depth of modulation of depth
- Frequency of modulation of frequency
- Frequency of modulation of depth

All envelope handling is done outside of the DSP by adjusting the volume of each voice.

Wavetable Module

The wavetable synth uses a conceptually complex synthesis technique that offers a very wide degree of flexibility of sound with a modest computational overhead. The wavetable synth plays a set of instructions. An instruction defines a waveform, a time, a volume change, a fade time and a next instruction. The waveforms consist of 128 samples. Each sample is a 16 bit signed integer sign extended to a long. The waveforms are 512 bytes long and must start on a 512 byte boundary. The instructions may loop to form a sustain. Much of the flexibility of the wavetable synth is derived from the fact that as the synth switches from one instruction to the next, the output waveform is the linear interpolation between the waveforms in the two instructions.

The parameters for the wave table synth are:

- Pitch
- Volume
- Pointer to Release Instruction
- Pan Value
- Release Flag
- Pointer to First Instruction
- Sample Length (2^N size)

The Instructions contain:

- Pointer to Sample
- Number of Ticks to Fade to Next Sample
- Pointer to Next Instruction
- Number of Ticks to Play the Sample
- Amplitude Fade

The wavetable amplitude fade control acts like a built-in envelope.

The Waveform module allows any 128 sample waveform (as defined for the wavetable synth) to be played to the DACs at any musical pitch. The volume of this is then modulated by what may be thought of as a very slow sample as an envelope. This envelope has the ability to loop so that long sustains may be achieved without using too much memory. The parameters for the waveform module are:

- Pointer to Waveform
- Pitch
- Volume
- End of loop
- Pan value
- Pointer to Envelope
- Loop flag
- Envelope rate
- Size of loop

Wavetable With Envelope Module

A second version of the waveform module exists. It uses a slope-destination, time envelope. The amplitude information is about the current point and the time is the amount of time it takes to get from the previous point's amplitude to this point's amplitude. The sustain point for this envelope is the second to the last point. The parameters for this version of the waveform module are:

- Pointer to Waveform
- Pitch
- Volume
- Pan value
- Pointer to Envelope
- Loop flag - loops at the sustain point
- Release slope

There are also two versions of the sampler module which use this slope-destination envelope. One is a 16 bit sampler and the other one is a compressed 16 bit sampler.

The last FM module, called the FM/Env synth, combines the Simple FM wave generation with the Waveform synth envelope generation.

Using The Synth

To use the synth follow these steps:

1) Load the synth code into the DSP.
2) Initialize some locations in DSP RAM.
3) Initialize the DAC and start the DSP.
4) Set up a "Voice Table".
5) Start the voice.
6) Turn off voices as required
7) Repeat from (4).

Voice Tables are stored in DSP RAM.

The DSP code, and all its internal variables, are in the bottom of DSP RAM. This allows TABLESTART (the start of the Voice Tables) to be quite low in DSP RAM (TABLESTART is a define, use it as the position may change). The size of the table at TABLESTART is not defined in the synth itself, it is determined by the programmer at run time (see table below). The remainder of DSP RAM should be used to store the following, (a) Custom samples for both wavetable and FM synthesis, (b) Voice Tables, these must be contiguous with TABLESTART, (c) Wave Table instructions and (d) Waveform envelopes. Other uses for DSP RAM may arise as new synthesis modules are written. Each Voice Table starts with a long (32 bit) value that indicates if the voice is active or not. The legal values are:

Value	Voice Type	Value	Voice Type
0	End of active voices	24	Waveform/Envelope
-4	Skip this voice	28	8-bit Sampler
4	FM/Env	32	Compressed Data Sampler
8	Simple FM	36	Waveform/Slope-Destination Envelope

Value	Voice Type
12	16-bit Sampler
16	Complex FM
20	2N wavetable
40	16-bit Sample/Slope Destination Envelope
44	Compressed 16-bit Sample/Slope Destination Envelope
48	Sound Effects Sampler Module (uses 16-bit compressed samples)

The values in the rest of the Voice table are given in the following pages. In the tables that follow, the symbol ✱ means this value may be changed while the note is active. Values not specified do not need to be set. The end of the Table list is indicated by a 0 where the next table would start.

When doing polyphonic synthesis (more than one note at a time), the volume of each voice **must** be reduced to avoid overflow. For example a single loud voice would have a volume of about $6000. Adding 3 of these would overflow 16 bits. To avoid this you must scale down the volume of each voice such that the total fits into 16 bits. In the preceding example a reduction of about 3 would work.

The values to use for pitch are given in the accompanying spreadsheet. Find the note that you want the value for. The values for the FM synths and the wavetable synth are in the column marked (64K) for the other modules the value to use is in the column (256).

The synth has a certain amount of time available to synthesize each sample, during that time it can do only so much. The total time available is 168 time units (these are not clock ticks). The following is a list of the approximate number of time units used by each synth module:

Simple FM	~15 time units
Complex FM	~24 time units
FM/Env	~23 time units
Samplers	~19 time units
Wave Table	~18 time units
Waveform synth	~19 time units
Waveform with slope-destination envelope	~17 time units
Sampler with slope-destination envelope	~23 time units
Skip a voice	~3 time units

These numbers may change as the synth modules are modified and optimized. The timings above assume that all table and sample data are in internal DSP memory (except for sample used by the Sampler module). The numbers given for the Sampler modules assume that the main bus is not busy doing other things. The total number of time units used can be computed from these numbers and kept below 167. The number available can be read from a location in DSP RAM called TIMELEFT.

Note: The 168 time units will reduce if oversampling is added to the synth.

The above timings assume that the synth is running at the default rate of ~20kHz. This can be changed by modifying the value stored in SCLK. If this is done then all of the pitch information will need to be modified.

Module Definitions

Simple FM	
Offset (longs)	Description
0	Voice type (8)
1	Pointer to Carrier Wave. Must be on a long (32 bit) boundary (should be DSP memory for speed). ✠
2	Pointer to Modulating Wave. Must be on a long (32 bit) boundary (should be DSP memory for speed). ✠
3	Reset to zero.
4	Pitch. Given as the size of a step in samples as a 15.16 number. ✠
5	Reset to zero.
6	Volume of this voice, 15 bits. ✠
7	Reset to zero.
8	Frequency of Modulation. Given as the size of a step in samples as a 15.16 number. ✠
9	Depth of modulation. This is a 7.8 number. ✠
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

Complex FM	
Offset (longs)	Description
0	Voice type (16)
1	Pointer to Carrier Wave. Must be on a long (32 bit) boundary in internal DSP memory. ✠
2	Pointer to Modulating Wave. Must be on a long (32 bit) boundary in internal DSP memory. ✠
3	Reset to zero.
4	Pitch. Given as the size of a step in samples as a 15.16 number. ✠
5	Reset to zero.
6	Volume of this voice, 15 bits. ✠
7	Reset to zero.
8	Frequency of Modulation. Given as the size of a step in samples as a 15.16 number. ✠
9	Depth of modulation. This is a 7.8 number. ✠
10-12	Reset to zero.
13	Frequency of modulation of frequency. Given as the size of a step in samples as a 15.16 number. ✠
14	Depth of modulation of frequency. This is a 7.8 number. ✠
15	Reset to zero.
16	Frequency of modulation of depth. Given as the size of a step in samples as a 15.16 number. ✠
17	Depth of modulation of depth. As a 7.8 number. ✠
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

Sampler

Offset (longs)	Description
0	Voice type (12 = 16 bit; 28 = 8 bit; 32 = compressed 16 bit)
1	High bit is the loop flag. The low 15 bits are the volume. ✻
2	Pointer to Sample. Must be on a word (sample size) boundary outside of internal DSP memory.
3	Pitch. Given as the size of a step in samples as a 23.8 number. ✻
4	End of loop in samples as a 23.8 number. For a non-looping sample this is the sample number at end of the sample. When the current pointer passes this point the Voice type is set to -4. For a looped sample this is end point of the loop. This is given in samples as an integer with no fractional part. ✻
5	Loop length in samples. This is a 23.8 number. ✻
6	Reset to zero.
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✻

Samples can be looped. (Note that this is a separate issue from looping in a music score.) Sample looping works like this. Assume a sample in memory. There are four points of interest.

- The start of the sample.
- The beginning of the loop.
- The end of the loop.
- The end of the sample.

To play a looped sample:

- Turn on the loop flag.
- Set the End Loop to the end of the loop. (In samples)
- Set the loop length (in samples) so that (Loop End - Loop length) = (beginning of the loop).

This will play the sample until it reaches the loop point, at which point it will loop backwards by *loop length* samples. Looping will occur continuously until you stop it. To stop looping, set the *End loop* value to the end of the sample (in samples) and clear the loop flag. At the end of a sample the voice type is set to -4 by the synth. This allows the voice to be skipped. The voice may be reused at this point.

2^N Wave Table

Offset (longs)	Description
0	Voice type (20)
1	Volume of this voice as a 15 bit number. This is the maximum volume reached by the voice. ✻
2	The high bit is the release flag. The remaining bits are the pitch as the size of a step in samples as a 23.8 number. ✻
3-4	Reset to zero.
5	Pointer to release instruction. May be anywhere in memory on a long (32 bit) boundary. For performance reasons it should be in DSP RAM.
6	Reset to zero.
7	Size of wavetable sample. This 23.8 number is 2^{n-1}.
8-9	Reset to Zero.
10	Pointer to the first instruction. May be anywhere in memory on a long (32 bit) boundary. For performance reasons it should be in DSP RAM. At the end of the release sequence this is set to -1.
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✻

After the release sequence completes, the pointer at offset 10 is set to -1 to indicate that the voice may be reused.

Waveform

Offset (longs)	Description
0	Voice type (24)
1	Pointer to Waveform. Must be on **512** byte boundary. For performance it should be in internal DSP memory ✻
2	Pointer to Simple Envelope (see separate definition). Must be on a long (32 bit) boundary. For performance it should be in internal DSP memory. ✻
3	Reset to zero.
4	Pitch given as the step size, in samples as a 15.16 number. ✻
5	Reset to zero.
6	Loop flag is the high bit, the rest is the overall volume as a 15 number. ✻
7	Envelope rate, given as the step size, in samples as a 15.16 number. ✻
8	End of loop in samples as a 15.16 number. For a non-looping sample this is the sample number at end of the sample. When the current pointer passes this point the Voice type is set to -4. For a looped sample this is end point of the loop. This is given in samples as an integer with no fractional part. ✻
9	Loop length in samples. This is a 15.16 number. ✻
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✻

Note: See the discussion on looping for the Sampler module.

FM Envelope

Offset (longs)	Description
0	Voice type (4)
1	Pointer to Carrier Wave. Must be on a long (32 bit) boundary (should be DSP memory for performance). ✠
2	Pointer to Modulating Wave. Must be on a long (32 bit) boundary (should be DSP memory for performance). ✠
3	Reset to zero.
4	Pitch. Given as the size of a step in samples as a 15.16 number. ✠
5	Reset to zero.
6	High bit is the loop flag. the low 15 bits are the volume. ✠
7	Reset to zero.
8	Frequency of Modulation. Given as the size of a step in samples as a 15.16 number. ✠
9	Depth of modulation. This is a 7.8 number. ✠
10	Pointer to Simple Envelope (see separate definition). Must be on a long (32 bit) boundary (should be DSP memory for best performance). ✠
11	Envelope rate given as the size of a step in samples as a 15.16 form. ✠
12	End of loop in samples as a 15.16 number.✠
13	Loop Length in samples as a 15.16 bit number.✠
14	Reset to zero.
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

Note: See the information on looping for the Sampler module.

Waveform with Slope-Destination Envelope

Offset (longs)	Description
0	Voice type (36)
1	Pointer to Waveform. Must be on **512** byte boundary. For performance it should be in internal DSP memory ✠
2	Release Slope. This is a 16.16 number.
3	Reset to zero.
4	Pitch given as the size of a step in samples as a 23.8 number.
5	Pointer to Slope-Destination envelope (see separate definition). Must be on a long (32-bit) boundary. For best performance in should be in internal DSP memory.
6	High bit is the loop flag. The low 15 bits are the volume. ✠
7	Current value. Reset to 1.
8	Current slope. Reset to 1
9	Current Destination. Reset to 1
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

Sampler With Envelope

Offset (longs)	Description
0	Voice type (40 = 16 bit, 44 = compressed 16 bit)
1	High bit is the loop flag. The low 15 bits are the volume. ✠
2	Pointer to sample. Must be on a long (32-bit) boundary outside of internal DSP memory.
3	Pitch given as the size of a step in samples as a 23.8 number.
4	End of loop in samples as a 23.8 number.
5	Loop length in samples. This is a 23.8 number.
6	Reset to zero.
8	End of Sample. This is a 23.8 number.
9	Pointer to Slope-Destination Envelope (see separate definition). Must be on a long (32-bit) boundary (should be DSP memory for best performance). ✠
10	Release Slope. This is a 16.16 number.
11	Current Value. Reset to 1.
12	Current Slope. Reset to 1.
13	Current Destination. Reset to 1.
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

Note: See the information on looping for the Sampler module.

Sound Effects Sampler

Offset (longs)	Description
0	Voice type (48 = compressed 16 bit)
1	High bit is the loop flag. The low 15 bits are the volume. ✠
2	Pointer to Sample. Must be on a word (sample size) boundary outside of internal DSP memory.
3	Pitch. Given as the size of a step in samples as a 23.8 number. only multiples of $1000 will sound exact, other pitches might add noise ✠
6	Reset to zero.
8	End of Sample. This is a 23.8 number.
19	Pan Value. 0 is full right, $3FFF is balanced, $7FFF is full left. ✠

This is a one-shot, non-looping, non-interpolated sampler module. The sample will only sound exact when played at its original pitch. The advantage of this module is that it is very fast, using only 12 to 13 time units. It is ideal for one-shot samples like sound effects or percussion instruments.

Other Structures used by the Jaguar Synth

Wave Table Instructions	
Offset (longs)	Description
0	Pointer to sample to be played. Must be on a 512 byte boundary. For performance should be it should be in internal DSP memory.
1	Unused.
2	Time. Length of time, in ticks to play this sample.
3	Fade value. This value sets the amplitude change per tick of fade. *A* becomes *A*n*, where *n* is a scaled 15 bit number. *n* = $4000 is no change, *n* = $2000 is divide volume by two, etc.
4	Fade length. The length of the fade given as N where the fade lasts $2^{(N-1)}$ ticks. $2 <= N <= 14$.
5	Pointer to next instruction. May be anywhere in memory on a long (32 bit) boundary. For performance reasons it should be in DSP RAM. This should be set to -1 to indicate the end of the voice.

Simple Envelope	
Offset (longs)	Description
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Slope-Destination Envelope	
Offset (longs)	Description
0	Must be set to 0x00010000
1	Must be set to 0x00000001
2	Slope value, in 15.15 format
3	Destination value, in 15.15 format
4	Slope value, in 15.15 format
5	Destination value, in 15.15 format
6	Slope value, in 15.15 format
7	Destination value, in 15.15 format
8	Must be set to 0x00000000
9	Must be set to 0x00020000

Jaguar Music Driver

The Jaguar Music driver is an extension to the sound system described in the section ***The Jaguar Synth***. It is assumed that the reader is familiar with that section. In either case, the code is the same, FULSYN. The only difference is that one of Jerry's timers is used to run a real time interpreter of preparsed MIDI data. This is then used to automatically turn the first *n* voices on and off. This requires the voicetable to be at least *n* entries in length. The number of voices used is set in the file PARSE.CNF. For simplicity, this document will assume that *n* = 8. The sample rate of the underlying synth is assumed to be the default ~20kHz. If this is changed then a new copy of NOTES.CNF must be generated.

The system is used as follows:

1) A MIDI file is created in file 0 format with no more than 8 note polyphony. This file is converted to a simplified format by the program PARSE, just type '`parse filename.mid`' on the commandline[4]. It creates a MADMAC assembly source code file containing data statements representing the MIDI score information. The default output filename is TEST.OUT.

   When PARSE runs, it also produces a description of the file to standard output (this can optionally be disabled). This should usually be redirected to a file. If one exists in the current directory, PARSE also reads a file named PARSE.CNF. This file is used to create patch maps. The default mapping is for all channels to map to the patch at their channel number (see the provided PARSE.CNF file for the format).

   Looping in the MIDI file is supported using the following controller events: Controller 12 marks loop targets, the value on controller 12 is the target number; Controller 13 selects a loop target and should be followed immediately by a Controller 14 event that gives the number of times to loop. A negative loop count causes it to loop forever. A comment is inserted into the output file that can be made into a label so that loop counts can be reset to loop more than 127 times. For more information see the format of the music events at the end of this document.

2) A set of patches and envelopes are created using the format described in **The Jaguar Synth** for voicetable entries, with a few differences.

   a) In all of the FM modulation frequency controls, the rate may be made proportional to the pitch of the note or left absolute. This is controlled by the high order bit of the frequency. The relative frequency is a 23:8 integer:fraction number. For example the value $80000100 results in the modulation frequency being the same as the pitch.

   b) A new parameter, the envelope/sample end point, is specified in the patch at the following locations:

[4] You can also manipulate your program's MAKEFILE so that the MIDI file is essentially the 'source' file and whenever it is updated, the PARSE and MADMAC programs will be called automatically by the MAKE utility. See the MAKEFILE for the sample program provided with the Jaguar Synth & Music Driver.

Module	Offset
Samplers	8
Waveform	10
FM/Env	15

c) For all samplers, the pitch may be adjusted by a factor placed in the pitch parameter of the patch. The value $1000 means no change, $800 drops the pitch by a factor of 2 (one octave) and a value of $2000 raises the pitch by a factor of 2.

d) For all patches, the volume may be adjusted by a factor placed in the volume parameter of the patch. The value $100 means no change, $80 drops the volume by a factor of 2, and a value of $200 raises the volume by a factor of 2.

3) The files are built into a program (see below)

4) The program is run and out comes the music.

The program PARSE converts the MIDI file into MADMAC assembler source code using **dc.l** directives. It is assembled and converted to a .SCR file[5]. At this time PARSE and the interpreter understand the MIDI functions for note on/off, MIDI volume, pitch bend, pan, tempo change, and looping. The system assumes envelopes are also provided using **dc.l** directives. These are assembled and loaded into the DSP at runtime

The Jaguar sound system may be thought of as having two separate components; a synthesizer and a music interpreter. These two sections are quite independent, although the second requires the first to actually generate sound.

To use the system, follow these steps. For clarity follow along in the sample code (DRIVER.S), Load the DSP code into DSP RAM, set up a voice table, turn on the I^{2}S port, start the DSP and turn off mute. The system is now ready for use as a synth. This functionality is primarily intended for interactive sounds.

Starting the Music Interpreter

To turn on the music interpreter set SCORE_ADD to the location of the tokenized music (this must be a long aligned address), set TIMER_ADD to 0, start the timer and out comes music. The remaining code shows how to add in custom effects. To play music and sound effects simultaneously make sure that you restrict sound effects to the voice table entries that the music interpreter does not use.

During each sample period the synth goes thru the voice tables (starting at TABLESTART) and checks the first longword of each one to find out which synth module to use next.

5 This is actually controlled by your MAKEFILE. You can use the standard .O extension normally used by object modules, or you can use a different extension to identify that this object module contains music score data. In the latter case, the .SCR filename extension (for Musical Score) is recommended.

The Music driver interprets a structure in memory to manipulate entries in the voice table. This structure is created by the program PARSE. A list is kept by the parser of all voices that are in use and a warning is given if the desired polyphony fails to accommodate the needs of the MIDI file being parsed. The voice assigned to a note on event is determined by taking the last used voice, adding one until an available voice is found. At any given time the voice table can be quite complex. A representative voice table follows (showing only the voice type in detail):

```
-4     x x x    ..x
 8     x x x    ..x
12     x x x    ..x
28     x x x    ..x
-4     x x x    ..x
-4     x x x    ..x
-4     x x x    ..x
16     x x x    ..x
-4     x x x    ..x
24     x x x    ..x
 0
```

This type of table would be expected while playing an eight voice music file with two channels reserved for sound effects.

More details may be found in the example files.

Stopping the Music Interpreter

To stop your music before the end of the score is reached, you do the following steps:

1) Ramp down the volume to fade out the music and/or sound effects. This step is optional, but it will probably sound better this way than if you just cut off the music abruptly.

2) Set the SCORE_ADD pointer to point at the end of your music score. This should contain a long word value of $7FFFFFFF.

3) Step 2 will cause the music driver to stop feeding the synthesizer's voice tables with new information, but it won't stop the synthesizer from processing the information already there. To do this, we must set the voice type value to -4 for each voice you want to turn off. (That's the first long word of each voice structure.) This tells the synth to do nothing for those voices.

You may want your sound effects to continue even if your music stops. If you are playing music only with the first five or six voices, and are using the last two or three voices for sound effects, then in step 1 you would change the volume parameters in the individual voice tables that are being used for music, and leave the volume of the sound effects voices alone (and don't turn off those voices in step 3). If you want to change the volume of everything, including sound effects, then you can either change all of the individual voices or you can change the UEBERVOLUME variable, which will affect all voices. The MIDIVOLUME variable will only affect new notes generated by the music driver; changing it will not change the volume of a note that has started but not yet finished.

When you want to restart your music, you would simply reset the voice types, volume, and SCORE_ADD variable to the appropriate values.

The Music Event Format

Each event consists of two long words. The first long is the time (in milliseconds) from the start of the song the the event is scheduled for (this limits the length of any individual tune, without loops, to about 6 weeks). The next long is the actual event encoded as follows.

Coded events look like this:

```
EEEV|VVxx|xxxx|xxxx|xxxx|xxxx|xxxx|xxxx
EEE = Event type
```

```
    1xx NOTE ON
1xxV|VVPP|PPPF|FFFF|FFFF|FFFF|FAAA|AAAA
```

```
V|VV = Voice number
PP|PPP = Patch number
F|FFFF|FFFF|FFFF|F = Frequency
AAA|AAAA = Amplitude
```

```
    000 NOTE OFF
000V|VVxx|xxxx|xxxx|xxxx|xxxx|xxxx|xxxx
```

```
V|VV = Voice number
```

```
    011 JUMP WITH COUNT
011D|DDDD|DDDD|DDDD|DDDD|DDDD|CCCC|CCCC
```

```
CCCC|CCCC is number of loops played
D|DDDD|DDDD|DDDD|DDDD|DDDD is the number of phrases to jump
```

```
    010 CONTROLLER CHANGE
010V|VVPP|PPPF|CCCC|CNNN|NNNN|NNNN|NNNN
```

```
V|VV = Voice Number
PP|PP = Patch Number
F = Flag to change the base pitch
CCCC|C = Controller Code
NNN|NNNN|NNNN|NNNN = Value
```

```
Controllers are:

        7 = Volume
        9 = Pitch Bend
        10 = Stereo Pan
```

Parse - MIDI File Parser

The MIDI parser is a command line program which translates a MIDI file into commands recognized by the Jaguar syntheziser. The output of the parser is a MADMAC assembler source file (ASCII) containing the sound data for the synthesizer in assembly language format. This file has to be assembled and linked in with your program, playing the music. The PARSE utility is documented in the **Tools** chapter of the documentation.

MERGE Utility

The MERGE utility is designed to take multiple music data files created with PARSE and merge them together into a single file that will contain everything interleaved together appropriately. The MERGE utility is documented in the **Tools** chapter of the documentation.

XNOTES Utility

The XNOTES utility is designed to automatically create a NOTES.CNF file with the correct note values for a given sampling rate. The NOTES.CNF file is used by the PARSE utility to control the frequency value that is used for each musical note. If you change the sample rate used by the Jaguar Synth, you should run XNOTES to create a new NOTES.CNF file, then run PARSE again on your MIDI files. If you skip these steps, the pitch of the notes will be incorrect. The use of the XNOTES utility is documented in the **Tools** chapter.

SNDCOMP Utility

The SNDCOMP utility is designed to take a 16-bit digitized sound file and compress it to 50% of its original size. The compression it does is a "lossy" compression, but the quality is quite good. The compressed sound files it creates are then used with the Jaguar Synthesizer. The SNDCOMP utility is documented in the **Tools** chapter of the documentation.

Jaguar Sound Tool User Guide

The Jaguar sound tool was written to provide a "user friendly" interface to the Jaguar synthesizer module. The sound tool provides a way of editing up to 8 voices by using one of the seven synthesizer modules. Each voice can be turned on individually or, together with other voices. Voices can be saved to or loaded from the host machine allowing you to save work in progress. Additionally, you may save your work in ASCII form, ready to be linked into your source code. For the rest of this section, it will be assumed that you have read ***The Jaguar Synth*** section.

General User Interface Rules

In general, each of the synth modules share the same user interface. Whenever possible, you'll find that the joypad keys display the same functionality throughout the different synth editors. You can move from object to object within an editor by holding down the **Fire B** button and then pressing up, down left, or right depending on the placement of the object that you would like to go to. An object is defined as a single slider, a group of buttons, or any other item that allows you to edit the voice that you're working on.

As you move to each object, you'll see it being selected by an green box drawn around it. The two main object types are numerical sliders and buttons. To change the value of a numerical slider, use the joypad up and down keys to add to or subtract from the total. Using the left and right buttons, you can move the slider cursor left or right. This will allow you to increment or decrement your slider value by a larger or smaller amount. Notice that the value will only increment or decrement by 1 each time you press the **up** or **down** key. To scroll through these numbers more quickly, hold down the **option** key while pressing **up** or **down**. Alternatively, you may type in the direct value and the number will appear at the cursor location. Button groups are much simpler to use. Simply select the joypad key which represents the button which you wish to select.

The following is a brief discussion of each of the synth editors along with a description of the main menu screen.

Main Menu

Each of the 8 synth voices can be edited through this main menu screen. As discussed earlier, use the **Fire B** key along with joypad up and down to scroll through each voice. When a voice is chosen, hit the **up** and **down** buttons to select a synth editor then hit **2** to edit the voice. Turn the voice on or off by hitting the **1** key. Hitting the **Fire A** key will turn on all of your enabled voices at once. Note that at startup, each of the voices except for the first one is disabled. Once you have edited a voice, you can return to the main menu by either using the main menu button or, by hitting the **pause** key.

The final row of buttons allows you to load or save out your current work. To save your work, move down until you've selected the last row of buttons. Hit the **2** key and the SNDTOOL program will cause a break command in the debugger on your host computer. You will be prompted by an alert box with instructions on saving your file. In the same manner, an ASCII file can be saved out by hitting the **3** key. Note that this is a 100% ASCII file which can be read into any text editor. Each of the voices is separated by a different label, voice1:, voice2:, etc. You will also find envelopes, user defined waveforms, and wavetable instructions saved out as well. All addresses within the voice table will be represented by a label. This label will either correspond to one of the labels embedded in the file, or, as in the case of sample addresses, simply be referenced as an external lable at the top of the file.

Use the ***Load Waves*** button to load in user defined waveforms. You can load in up to 5 different user defined waveforms. They are stored at the addresses UWAVE1, UWAVE2, ... UWAVE5. To read in a waveform for the first user defined wave, use the command:

```
    read filename .UWAVE1
```

The ***Cwave*** button performs harmonic synthesis using a table of 32 partials with user specified amplitude relationships. Briefly, any sound can be broken down into a series of sine waves called partials or harmonics. The Cwave utility allows the specification of the relative amplitudes of thirty-two harmonics, which are mathematically combined into a resultant waveform.

After pressing the **5** number key the harmonics can be entered by typing:

```
    sl .awave
```

At this point the first harmonic can be entered by typing a hexadecimal value and pressing [Return]. This automatically displays the field for the second harmonic. Pressing [Return] again brings up the field for the third harmonic, etc. After entering the last harmonic and pressing [Return] a dot ('.') has to be entered followed by a [Return] . The debugger then returns to its command line. To continue, type:

```
    g .continue
```

The Cwave utility stores the waveform it creates in user wave 1. After a wave has been created, it may be saved using the ***Waveform Load/Save*** button.

Simple FM Editor

Use the numerical sliders to change frequency and depth of modulation. Use the text sliders to select your waveforms and pitch. Select these values by using the up and down joypad keys until the selected pitch or waveform appears in the slider. Use the Frequency mode button to select the way the frequency value is calculated. When in "Fixed" mode, the frequency value in the voice table will be whatever is shown in the slider. When in "ratio" mode, the frequency value will be whatever is in the slider multiplied by whatever pitch value you have. Note that the frequency multiplier will be in the 15.16 format so for instance, 1.32768 in the slider will represent a multiplier value of 1.5. Exit the synth by using the **Main Menu** button or by hitting the **pause** key in any object. Play the sample by pressing the **Fire A** button. Press it again to turn the voice off.

Complex FM Editor

Identical to Simple FM except for extra sliders to provide an extra indirection of modulation. The synth documentation will provide the needed details.

16 Bit Sample Editor
16 Bit Compressed Sample Editor

From a user interface standpoint these two editors are virtually identical. There is currently a default 16 bit sample built into the sound editor. To load additional samples, select the **Load Sample** button from the first group of buttons.

The sound tool will currently handle Audio IFF files and AVR files as well as raw sample files. Since there is no header information stored with a raw sample file, you must set the variable ***.samplesize*** to let the sound tool know how big the newly loaded sample is. You can accomplish this by typing in the following:

```
sl  .samplesize  (type in new number of samples here)
```

You can now type in "g .continue" to return to the program. Currently the maximum sample file size that the sound tool will accept is 200000 bytes. (NOTE: The tool currently does not extract pitch information from AIFF files.)

Use the numerical sliders to set loop length, loop end and pitch values. You can play the sample by pressing the Fire A button at any time. If the **Loop On** button has been selected, the sample will play continuously, looping through the parameters which you have set up. Once the **Fire A** button has been released, the synth will play the rest of the sample.

Waveform Editor

Use the numerical sliders to set rate, loop end, and loop length. Use the **up** and **down** buttons to cycle through the given pitches and waveforms. You can edit the envelope by first making it the current object. Use the joypad **up** and **down** buttons to increase or decrease values at the current point. Move to the next point in the envelope by holding down the **Fire C** button and using the joypad **left** and **right** buttons. Insert points by pressing the **1** number key on the keypad. In the same way, delete points by the **4** key. Pressing the **0** number key will restore the envelope to a standard default. You may choose any one of five envelopes (through the envelope slider) to sample or edit. Each time you scroll through an envelope you will be able to see it change visually on the screen. The voice can be played by using the **Fire A** button. As with the sample editor, the sound will loop until the **Fire A** button is released.

A new envelope can be saved or loaded by selecting the load/save menu button. Load or save functions will affect the current envelope. (The one displayed in the slider) After breaking, you will be promted to input the correct commands to load an envelope. At this point you can also save out the current envelope to be used at another time.

FM Envelope

This synth editor combines the features of the waveform and simple FM synths. See **The Jaguar Synth** section for details.

2^N WaveTable Editor

The 2^N Wavetable editor will allow you to edit a set of wavetable instructions. Use the sustain/release buttons to select which list of instructions you want to edit. The large object in the center of the screen will hold your list of instructions. Notice that the current instruction in this list will be highlighted in green. Use the **up** and **down** joypad keys to scroll the list. This current instruction will also be represented by the sliders at the bottom of the screen. You can use these sliders to create a new wavetable instruction. Use the panel of buttons on the right side of the screen to insert the new instruction (represented by the slider values) into the actual wavetable instruction list. You can also change the existing instruction or remove an instruction using this bank of buttons. The last instruction in your sustain list will automatically loop to the first instruction. If you would rather loop to another instruction, place the index of the instruction that you want to loop to into the **Loop To** slider. Notice that the **Fade Length** slider shows positive values. The tool will negate the value before passing it on to the synth.

16 bit Sampler/Envelope
16 bit Compressed Sampler/Envelope

This synth editor combines the features of the waveform and 16 bit sampler synth. Note that the envelope is of a different kind in this module. The new envelope for this module is a basic slope-destination, time envelope.

The Amplitude information is about the current point and the Time is the amount of time it takes to get from the previous point's amplitude to this point's amplitude.

You can add points by pressing the **1** number key while inside the envelope window and delete points by pressing the **4** key. To move from point to point hold down the **Fire C** button and use the joypad. The point can be edited vertically as well as horizontally.

The two parameters that are available to the user are:

- Amplitude (0 - 32767)
- Time (0 - 2,000,000,000 ms)

The information (Amplitude and Time) about each point are updated as the points are moved. See **The Jaguar Synth** for details.

Procedure Summary

The basic tasks for processing MIDI files consist of:

- converting (or parsing) your MIDI file into a form that the Jaguar can use
- creating synthesizer and sample patches
- incorporating patch information into files used by the Jaguar synthesizer

Figure 1 illustrates these tasks. The following is a summary of the steps required to complete these tasks. Each of these steps is described in detail in later sections of this document.

1. Install the Jaguar Music System tools.

    a. Install the tools and sample code from the distribution archives
    b. Create a new directory for your music project.
    c. Copy the Jaguar sound files to the new directory.

2. Create your sound patches.

    a. Design and save your synthesized and sample patches.
    b. Save ASCII versions of your patches.
    c. Convert your samples to raw format, compress them, and write down sample information.

3. Prepare your MIDI file.

    a. Clean up your MIDI sequences.
    b. Write down information about your MIDI sequences.
    c. Save your MIDI file in sections as separate type 0 MIDI files.

4. Copy your MIDI Type 0 files, patch ASCII files, and samples.

5. Extract patch data, envelope, waveform and wavetable data to separate ASCII files.

    a. Extract patch data to separate ASCII files.
    b. Replace the label names in your patch data.
    c. Adjust other patch values in your patch data.
    d. Extract envelope data to separate ASCII files.
    e. Extract user waveform data to separate ASCII files.
    f. Extract wavetable data to separate ASCII files.

6. Modify the file `synth.s`.

    a. Set the number of patches.
    b. Include patch data files.
    c. Write down patch numbers.
    d. Add sample labels and include sample files.

   e. Initialize the voice table to the correct number of voices.
   f. Add waveform labels and include user waveform files.
   g. Add envelope labels and include envelope files.
   h. Add wavetable labels and include wavetable files.

7. Add MIDI information to `parse.cnf`.

8. Run the `parse` program to parse your MIDI files.

9. After testing your music one section at a time, run the `merge` tool to combine your sections.

10. For each MIDI file, change the `MIDIFILE` entry in the `makefile`.

11. Run the `make` tool.

12. Load and run `test.cof`.

13. Refine your MIDI files, patches, and voice settings.

14. Adjust volume and tempo in `synth.cnf` if necessary.

15. Repeat steps 5 through 14 until your music plays correctly.

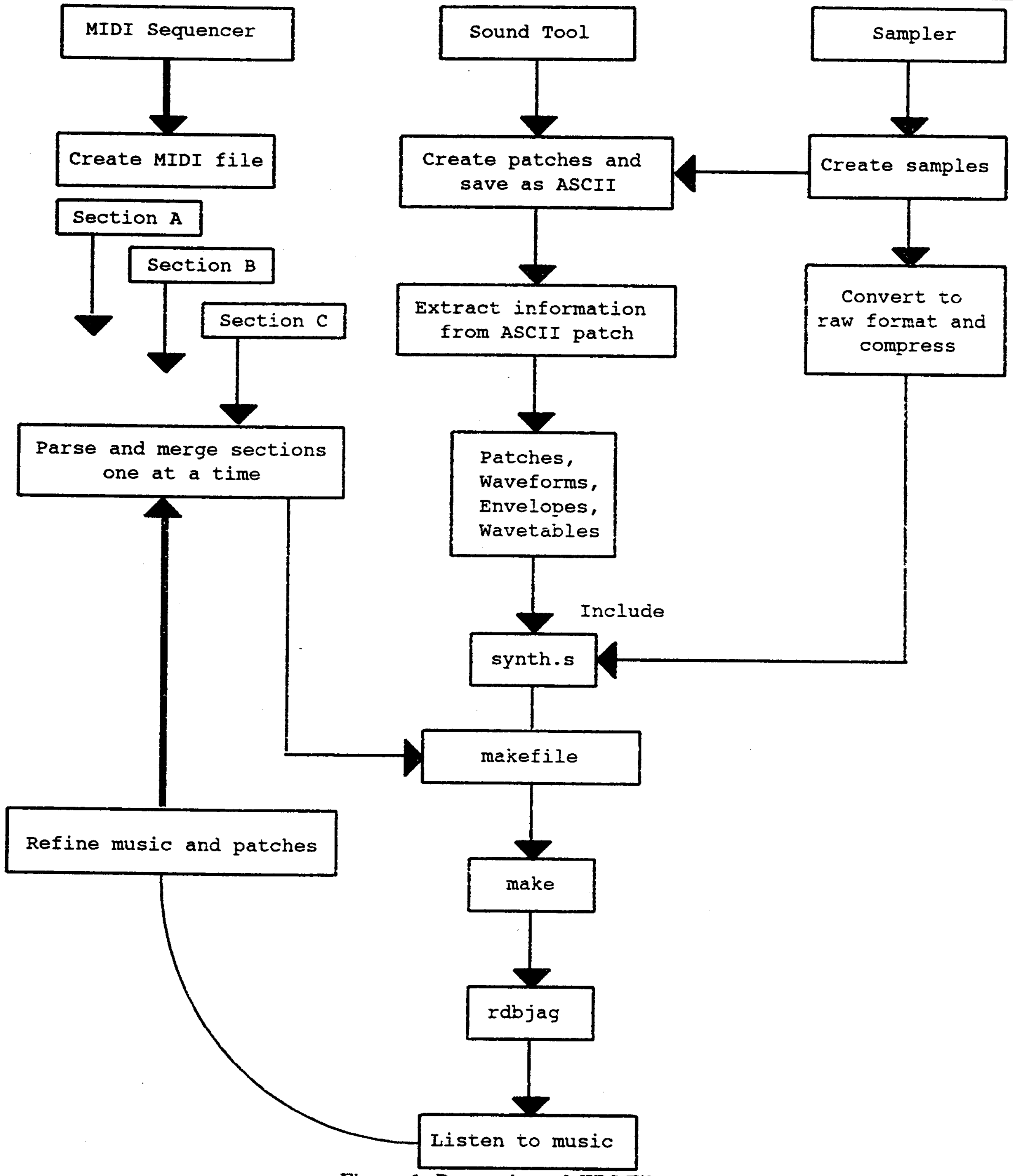
MIDI Sequencer
Sound Tool
Sampler
Create MIDI file
Create patches and save as ASCII
Create samples
Section A
Section B
Section C
Extract information from ASCII patch
Convert to raw format and compress
Parse and merge sections one at a time
Patches, Waveforms, Envelopes, Wavetables
Include
synth.s
makefile
Refine music and patches
make
rdbjag
Listen to music

Figure 1. Processing a MIDI File

Step-by-Step Procedure

This section presents the steps for processing a MIDI file in detail.

Step 1. Install the Jaguar Music System Tools

a. Install the tools and sample code from the distribution archives.

The Jaguar Music System tools and sample files are installed automatically when you install the disks that come with a Jaguar Development System. If you have received updated archives containing the tools (or downloaded them from an online service), then you should extract the archives into a temporary directory. The directory structure used in the archives is:

JAGUAR\BIN - Various tools such as the MIDI parser, sound sample file format conversion utilites, etc.

JAGUAR\MUSIC\FULSYN - The Jaguar Synthesizer, source code and linkable object code.

JAGUAR\MUSIC\SNDTOOL - The Jaguar Synthesizer Sound Tool - Used for creating patches for the Jaguar Synth.

JAGUAR\MUSIC\SNDTOOL.MID - The MIDI version of the Sound Tool.

JAGUAR\MUSIC\SOUNDS - A variety of ready-made sound patches for use with the Jaguar Synth and the Sound Tool.

JAGUAR\MUSIC\MUSICDRV - The sample program for the Jaguar Synth. This is the sample program described in this document.

JAGUAR\MUSIC\SYNDEMO - This is an alternate sample program for the Jaguar Synth. This one includes a more complex MIDI score that uses multiple instruments and looping. Also, this one uses multiple FM patches and no samples.

To extract the various archives using this directory structure, use the following command:

```
pkunzip -d music.zip
```

Where "music.zip" is the name of the archive you are extacting at the moment. The PKUNZIP tool is supplied on your original Jaguar Developer System disks.

If you are installing an update, please always extract the archives to a temporary directory first, so you can backup the existing files before copying over the new ones.

b. Create a new directory for your music project.

Make a new directory on your hard disk. You will use this directory to hold your MIDI file, synthesizer patches, samples, and several Jaguar files and programs .

The Jaguar Music System Tools distribution includes two sample projects. One plays a simple scale of notes using the Jaguar Synth's Sample module. This project is contained in the directory JAGUAR\MUSIC\MUSICDRV. The second sample plays a more complex song with multiple voices, and uses FM patches instead of samples. This project is found in the JAGUAR\MUSIC\SYNDEMO directory.

c. Copy the Jaguar sound files to the new directory.

This document uses the MUSICDRV project as its example. You will need the following files to perform the procedure described in this document. During this procedure, you will need to modify some of these files. Be sure to save the original copies of these files so you can use them for other projects.

You *will* need to change the following files using a text editor.

- `makefile`

  This file is used by the MAKE tool to compile various files into an executable program file.

- `parse.cnf`

  This file contains MIDI channel, MIDI note range, voice number, and transposition data for the MIDI parsing process. It is used by the PARSE utility.

- `synth.s`

  This file is used to assemble patch data, samples, envelopes, user waveforms, and wavetables that must reside in the Jaguar's memory.

- `synth.cnf`

  This file contains settings for global and MIDI volume of the synthesizer and the system clock used to adjust music tempo. This file also allows the Jaguar Synth to be reconfigured for the optimum performance and memory usage requirements for individual projects (this requires that the Jaguar Synth source code be reassembled -- see below).

You *will not* need to change the following files:

- `driver.s`

  This file contains initialization information for the Jaguar synthesizer.

- `fulsyn.inc`

  This file contains parameter settings and instructions for the Jaguar synthesizer. (This file is located in the JAGUAR\MUSIC\FULSYN directory.)

- `fS02_50.das`

This file is the Jaguar DSP source code for the Jaguar synthesizer. You should not have to change it, but you may recompile it to add or delete different synthesizer modules according to the needs of individual projects (controlled by the SYNTH.CNF file). (This file is located in the JAGUAR\MUSIC\FULSYN directory, but depending on the version, the filename may change.)

- `fS02_50.oj`

  This file is the linkable object module for the Jaguar synthesizer (This file is located in the JAGUAR\MUSIC\FULSYN directory. Depending on the version, the filename may change.)

Step 2. Create your sound patches.

a. Design and save your synthesized and sample patches.

Create the sound patches to be played by your MIDI file. You may want to perform this step before you compose your music, or perhaps at the same time. This way, you will have a better idea of what sounds the Jaguar is capable of producing.

You can use the Sound Tool to create synthesized patches or use sampling software to create 16-bit samples.

If you use samples, we suggest you use a sampling rate of approximately 20 KHz to match the default playback frequency of the Jaguar. You must use mono samples. If you have stereo samples, you can use the MONO utility to convert them to mono.

We suggest you use the Sound Tool to set parameters of your samples, including pitch, loop parameters, and envelopes. For more on voicing samples on the Jaguar, see the *More on Voicing Samples* section. Load the Sound Tool into the Jaguar using `rdbjag` by typing the following:

```
rdbjag
load sndtool.db
g
```

For more information about creating sound patches, see the *Jaguar Sound Tool Users Guide* and the *Jaguar Synth* document.

The Sound Tool creates two kinds of patch files. One is an ASCII file designed to be assembled as Madmac source code as part of your project. The other is a binary file used to load and save patches that are being edited. Although it creates both types of files, the Sound Tool only knows how to load the binary files. Therefore, after creating a patch, we suggest you always save it in a non-ASCII file so you can reload it into the Sound Tool at a later time and make changes as needed. When saving these files, we suggest you save the files with an extension of `.ptc` in a directory called `sounds`.

Important: Synthesizer patches use a lot less memory than samples. And, samples use outside resources that are shared by graphics, causing slower game play and possible sample distortion. Because of these problems, you should avoid using samples as much as possible and instead use synthesized sounds for your music. This is particularly important for games in which the available space for music is very limited. If you must use samples, restrict them to important sounds that you cannot synthesize.

b. Save ASCII versions of your patches.

For each patch you create, use the Sound Tool to save it as an ASCII file. If you created any patch data information for samples, you should save this patch data as ASCII as well.

To save a patch in ASCII format, go to the main page of the Sound Tool and select the `Save Patch` command. We suggest you name these files with an extension of `.asc`, and place these files in a directory called `ascii.`

c. Convert your samples to raw format, compress them, and write down sample information.

The Jaguar DSP plays raw samples only. Raw samples contain the sample sound information, but do not contain other information such as looping data. If you created your sample in another format, such as the Audio Interchange File (AIF) format, you need to convert your samples to raw format for them to play correctly on the Jaguar. To do this, use the `stripaif` tool on your samples, and create other sample parameters (looping and pitch) in the patch data using the Sound Tool.

Next, compress your samples using the `sndcmp` tool. This tool compresses samples from 16 bit to 8 bit. Also, write down the file name and file sizes of each sample. You may need the file size information when adding patch data to `synth.s`.

Step 3. Prepare your MIDI File.

a. Clean up your MIDI sequences.

After composing your music, you may want to clean up or modify your MIDI sequences before processing them for the Jaguar. Use your sequencing software to inspect each of your MIDI tracks. When examining your tracks, look for the following and make changes as needed:

1. Verify that the number of voices being played by all of your tracks at one time (the polyphony) does not exceed the polyphony you are allowed for your game music.

   The Jaguar's polyphony is determined by the amount of time the synthesizer has to create each sound. The amount of time the Jaguar takes to create a sound depends on which synth module is for the sound. The total time available for the Jaguar to create sounds is 168 time units. Therefore, when determining the polyphony for your music, you must add the time values for each module you use to make sure the total time is at or below 167. Also keep in mind that some of the Jaguar synth's time available may be used to synthesize sound effects instead of music. For more information about calculating polyphony, see the *Jaguar Synth* document.

2. Check the quantization of your tracks to be sure that the timing of your notes (when notes start and end) is what you want. You may choose to leave your music as you recorded it to give it a more natural feel. Or, you may need to quantize some or all of your notes to correct for timing problems.

3. Check that the note durations are what you want them to be. For example, if a note is used to trigger a sample that does not use an envelope, you may want to shorten the note duration to prevent undesired looping. You can also adjust the loop parameters of a sample and apply an envelope to it using the Sound Tool.

   Be aware that any notes that trigger patches with long decays may affect your polyphony calculations since decay of the patch sound may overlap new notes being triggered. Too avoid this problem, be sure that your patch envelopes decay before the next note is triggered for that patch. For example, suppose there are two sequential half notes, with the first note ending before the second is triggered. Also suppose that the tempo of your music causes each note to last for one second. If the patch you use for these notes has an envelope that decays in one second or less, there is no problem. However, it the envelope decays in longer than a second, another voice will be needed to play the second note. If you are at the limit of your polyphony, the second note may not play at all.

4. Verify that the note on velocities are set to the desired level. For example, you may want to make the attack of a track consistent. On the other hand, you may want to leave them exactly as you performed them.

5. Adjust the volume the instruments used for each track (MIDI controller 7) as needed. You will likely be using different sounds on the Jaguar than the ones you used to compose your music. Because of this, it is hard to predict the what the relative volumes will be for your Jaguar sounds. For example, you might set the volume of your kick drum to be just right when you play it back on your sequencer. But, when you play it on the Jaguar, the kick may not be loud enough. Because it is hard to know ahead of time what the relative volumes will be for your patches, you may want to set some or all of your instruments volumes to a constant level (such as MIDI value 100). You can then mix the volumes on the Jaguar as needed from within the patch data file (synth.s) until they sound right.

6. If you want to have your MIDI file loop in the game, you need to set loop points in your MIDI file. For more information about how to set MIDI file loop points, see the *Looping MIDI Files* section of this document.

b. Write down information about your MIDI sequences.

Write down your MIDI file information for later use.

1. Write down the MIDI channel numbers for each track in your MIDI sequences. You will need these numbers when you parse your MIDI file in step 11.

2. Write down the MIDI note ranges (as MIDI note numbers) for each track. This information is required if you intend to play different sounds on the same MIDI channel. For example, if you recorded a track using a split keyboard, or drum machine, you need to write down which notes are for which sounds. You will use this information when you parse your MIDI file.

c. Save your MIDI file in sections as type 0 MIDI files.

The Jaguar music driver software plays type 0 MIDI files. This is a standard MIDI file format that merges multiple-channel tracks into single tracks. Type 0 MIDI files still retain the MIDI channel information of your tracks.

Therefore, to play your MIDI music, you must first convert it to one or more type 0 MIDI files. To test your music on the Jaguar, we suggest you save individual tracks (or groups of musically related tracks) as separate type 0 MIDI files. This way, you can test and refine separate parts of your music, making it easier to identify and fix problems you may find.

After testing and refining your tracks, you can use the `merge` tool to merge these files into one file for use on the Jaguar.

When saving your MIDI sequences, we suggest you name them with an extension of `.mid`.

Step 4. Copy your MIDI Type 0 files, patch ASCII files, and samples.

If they are not already there, copy your MIDI type 0 files, each of the ASCII patch files you created, and your samples, to your music project directory.

Step 5. Extract patch data, envelope, waveform, and wavetable data to separate ASCII files

a. Extract patch data to separate ASCII files.

Edit each ASCII patch file you created and locate the patch data. This data is a column of `.dc.l` values used by the Jaguar synthesizer and music driver. The patch data is located after the label `_sounddata:`

Each ASCII patch file contains data for all pieces needed for your synthesis module. All envelopes, user waves etc. associated with your sound will be save in one file.

Once you have located the patch data, copy it from your ASCII patch file to a separate file.
We suggest you name these files with an extension of `.dat`, and place them in a directory called `patches`.

b. Replace the label names in your patch data

Replace the temporary label names (`_env0`, `_env1`, and so on) in your patch data to match the label names you will put in `synth.s`. For synthesized patches, you may need to replace envelope, user waveform, and wavetable labels within your patch data. For sample patches, you will need to replace sample and envelope labels.

We suggest you prefix label names for envelopes with `e_`, user waveforms with `w_`, wave tables with `t_`, and samples with `s_`. For consistency across platforms, we also recommend you use labels of eight or fewer characters.

c. Adjust other patch values in your patch data.

There are other voice parameters you may want to modify in the voice data of your patches. These parameters include the volume and pan value, among others. The location of the volume parameter varies with the type of patch you are editing.

The pan parameter is always the four rightmost digits in the last parameter in a patch. You can adjust the pan value between `00000000` (pan full right) and `00007FFF` (pan full left). Setting this parameter to `00003FFF` centers the balance.

Refer to the *Jaguar Synth* document for descriptions of these and other parameters for the type of patch you are adjusting.

d. Extract envelope data to separate ASCII files.

Edit each ASCII patch file you created that uses envelopes (such as FM envelope and sample patches). Within each file, locate the envelope data that your patch actually uses. Envelope data is located in the file after the patch and user waveform data.

Each ASCII patch file contains data for the envelope used in your sound (`_env0` - `_env7`).

Once you have located the envelope data for your patch, create a separate file and copy the data into the file. Do this for each patch that uses an envelope.

We suggest you name each file as *`patch`*`.env`, where *`patch`* is an abbreviation of the patch name associated with the envelope. Write down the file names for future reference. You will need to include these file names in `synth.s`.

When saving an envelope data file, we suggest you place it in one of two directories, `env` or `slopeenv`. Place envelopes you extracted from sample envelope patches in `slopeenv` directory. Place all other envelopes in the `env` directory.

e. Extract user waveform data to separate ASCII files.

Edit each ASCII patch file you created that uses a user waveform. Within each file, locate the user waveform data that your patch actually uses. User waveform data is located after the envelope data in the file.

Once you have located the wavetable data for your patch, create a separate file and copy the data into the file. Do this for each patch that uses a user waveform.

We suggest you name each file as `patch.wav`, where `patch` is an abbreviation of the patch name associated with the user waveform. Place these files in a directory called `waveform`. Write down the file names for future reference.

f. Extract wavetable data to separate ASCII files.

Edit each ASCII patch file you created that uses a wavetable. Within each file, locate the wavetable data that your patch actually uses. Wavetable data is located after the patch data in the file.

Once you have located the user waveform data for your patch, create a separate file and copy the data into the file. Do this for each patch that uses a wavetable.

We suggest you name each file as `patch.tbl`, where `patch` is an abbreviation of the patch name associated with the user waveform. Place these files in a directory called `wavetabl`. Write down the file name for future reference.

Step 6. Modify the file `synth.s`.

a. Set the number of patches.

Set the `dc.w` value under `patches::` to be the number of patches you are using. For example:

```
...
patches::
  dc.w    7                 ; NUMBER OF PATCHES
```

b. Include patch data files.

Once you have created separate ASCII patch files include the file names in `synth.s`. The location for including these patch files is labeled in `synth.s` as `patches::`

It is important to realize the *order* in which you put your patches in synth.s defines the patch number used by the Jaguar. For example, the first patch in synth.s will be patch 0.

```
...
; Patch 0
  .include 'patches\\strlow.ptc'      ; strlow patch  ( \\ is needed because \ is a
                                      ; special character ). uses 's_strlow' sample
                                      ; and 'e_strlow' envelope
...
```

For a complete example of this file, see the *Example Files* section.

c. Write down patch numbers.

Write down the numbers for the patches you add. You will need to know these numbers when you modify `parse.cnf` to map your MIDI channel numbers to the actual patches you use.

d. Add sample labels and include sample files.

Add labels for your samples and include your sample files. The labels you choose must match those you specified in your ASCII sample patch files. For example:

```
...
s_strlow:
     .incbin     "samples\\synstrgs.cmp" ; sample used in patch 0
...
```

e. Initialize the voice table to the correct number of voices.

Add a zero to the voice table field that is the last voice to be used. For example, the following table places a zero at voice 7, indicating eight voice polyphony:

```
...
  .ORG   tablestart

TABSSTART::                          ; DO NOT EDIT THIS LABEL
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 0
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 1
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 2
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 3
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 4
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 5
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 6
  dc.l    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 7-LAST
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 8
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 9
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 10
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 11
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 12
  dc.l   -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 13
  dc.l   0
...
```

f. Add waveform labels and include user waveform files.

Add labels for your user waveform and include your waveform files. The labels you choose must match those you specified in your ASCII patch files that use the waveform.

g. Add envelope labels and include envelope files.

Add labels for your envelopes and include your envelope files. The labels you choose must match those you specified in your ASCII patch files that use the envelopes.

h. Add wavetable labels and include wavetable files.

Add labels for your wavetable and include your wavetable files. The labels you choose must match those you specified in your ASCII patch files that use the wavetable.

Step 7. Add MIDI information to `parse.cnf`.

Edit the file `parse.cnf` to set the polyphony of your music, map your MIDI channels to the voice numbers you set in `synth.s`, define the note ranges for your voices, and transpose your tracks if necessary. The format for entering this information is:

`n = note_polyphony`
`MIDI_channel - 1: note_range patch_number transpose_value`

`MIDI_channel` `-` 1 sets the MIDI channel number. You must subtract one from it since the Jaguar voice numbers are zero-based.

note_range sets the range of notes played by a particular sound. This allows you to achieve the same effect as a split keyboard or a drum machine in which one MIDI channel is used but different sounds are triggered depending on the notes played. For example, for MIDI channel 1, MIDI note 36 may trigger a kick drum sound, while MIDI note 38 will trigger a snare.

patch_number is the number of the patch to use based on the sounds you defined in `synth.s`.

transpose_value is the amount in which to transpose the defined note range The transposition is in one note increments and can be either positive or negative A value of 12 will transpose up an octave, a value of -12 will transpose down an octave, and a value of 0 will leave the notes untransposed For example:

```
...
n = 8           ; 8 note polyphony
0: 36-36 0 0    ; kick
0: 42-42 1 0    ; clsdhat
```

```
0: 46-46 2 0     ; openhat
...
```

For a complete example of this file, see the *Example Files* section.

Step 8. Run the `parse` program to parse your MIDI files.

Normally you would edit the makefile file for your project to include the names of your MIDI files so that the PARSE tool is called automatically when required. See the makefile for the sample programs for examples of this. However, you can also run the PARSE utility directly from the commandline if necessary. Type the following command to parse your MIDI files:

```
parse -q yourMIDIfile
```

The `-q` is an optional flag to suppress the output of the parse command. If you want to examine the parsing process as it occurs, do not use this flag. The parse output will be displayed to the screen. You can also redirect this output to a file so you can inspect it later. The parsing information may be useful for finding a problem if your MIDI file does not play correctly.

A common error you may see is that note on or note off has failed. This occurs when the polyphony of your MIDI file exceeds the polyphony you defined in parse.cnf. If this happens, increase the polyphony value (if possible) or reduce the polyphony in your MIDI file.

See also the PARSE utility release notes (in the JAGUAR\DOCS directory).

Step 9. After testing your music one section at a time, run the `merge` tool to combine your sections.

Merge your separate MIDI sections into one file. Use the merge tool to do this as follows:

```
merge merged_file input_file1.out input_file2.out ...
```

where *merged_file* is the resulting merged MIDI file, and *input_files* are the parsed output files of your individual sections generated by the `parse` program.

Normally, you would edit your project's makefile so that the MERGE tool would be called by the MAKE utility when appropriate.

Step 10. For each MIDI file, change the `MIDIFILE` entry in the `makefile`.

Edit the makefile and change the file name of the MIDI file you are processing. For example:

```
...
MIDIFILE = cscale
...
```

For a complete example of this file, see the *Example Files* section.

Note: Do not change anything else in the makefile unless you are familiar with how it works. Changing other text , spaces, or tabs in this file may cause it to not work correctly.

Step 11. Run the `make` tool.

Run the `make` program as follows to create the file `test.cof`. This file is the executable version of your music for the Jaguar. Type:

```
make
```

Step 12. Load and run `test.cof`.

Run the debugger `rdbjag` and load the file `test.cof`. This command will play your music on the Jaguar as it will sound in the actual game. Type the following commands:

```
rdbjag
aread test.cof
g
```

Step 13. Refine your MIDI files, patches, and voice settings.

Repeat the steps above as needed to refine your MIDI files, patches, and voice settings. It is often necessary to adjust the volume of your instruments and mix between them using the pan parameters. You may also need to adjust the pitch and loop parameters for your samples.

Step 14. Adjust volume and tempo in `synth.cnf` if necessary.

If necessary, adjust the global or MIDI volume settings in `synth.cnf`. Also, adjust the tempo. If your music plays too slowly adjust the SCLKVALUE parameter down. If it plays too quickly, adjust the parameter up. For example:

```
...
  GLOBALVOLUME   equ   $7fff
  MIDIVOLUME     equ   $7fff
...
  SCLKVALUE      equ   19
```

Step 15. Repeat steps 5 through 14 until your music plays correctly.

Rerun `parse`, `merge`, and `make` to generate a new `test.cof` file. Then, run `rdbjag`, load `test.cof`, and type '`g`' to play your music. Repeat this process until your music plays correctly.

More About Voicing Samples

We suggest you minimize your use of samples in your music because they use a lot of memory. However, if you use samples, you can either use the Sound Tool to create sample patch data for you, or copy the patch data of any sample that already exists in `synth.s` and modify it as needed. In general, we suggest you use the Sound Tool to set sample parameters, particularly if you need to adjust loop parameters, such as beginning, ending, and length of the loop, or if you want to apply a volume envelope to your sample.

If you have *not* used the Sound Tool to create the voice data for your samples, and instead have copied data for an existing sample, you must change the following `.dc.l` parameters of the sample voice:

- voice type

  The first parameter in the voice data of a sample. The voice type must be `$0000002C` for 16-bit compressed samples.

- volume

  The second parameter in the voice data of a sample. The volume can be any hexadecimal number that occupies the four rightmost digits. The maximum volume is `00007FFF`.

- sample label

  The third parameter in the voice data of a sample. The sample label is a label you define to identify the sample in the `makefile`. This parameter is also known as the start of the sample.

- sample pitch

  The fourth parameter in the voice data of a sample. The sample pitch is typically `$00001000`, which indicates no change from the original sample pitch. A value of `$00002000` doubles the pitch (raises it an octave) and a value of `$00000800` halves the pitch (lowers it an octave).

- end of loop point

  The fifth parameter in the voice data of a sample. The end of loop point for the sample. The value for this parameter is:

  ```
  ((file_size/2) <<8) - 1
  ```
  where the `file_size` is the size of the sample you noted in step 9.

- loop length

  The sixth parameter in the voice data of a sample. The loop length for the sample. The value for this parameter is also:

  ```
  ((file_size/2) <<8) - 1
  ```

- end of sample

  The ninth parameter in the voice data of a sample. The end of sample point for the sample. The value for this parameter is also:

  ```
  ((file_size/2) <<8) - 1
  ```

- sample envelope label

  The tenth parameter in the voice data of a sample. The label of the sample envelope as defined in `tables.das`:

Looping MIDI Files

During game play, you may want one or more of your MIDI files to repeat until the player completes a task of moves to another level. To do so, you need to add loop parameters to your MIDI file before processing it. The following procedure describes how to add this information.

1. Identify the point in your MIDI file where you want to start looping. This is called the loop target. At that point in your MIDI file, insert a MIDI controller 12 event with a value of the target number (for example, a 0 for the first target, a 1 for a second target (if any).

2. Locate the position in your MIDI file where you want to stop looping. At this point in the file, insert a MIDI controller 13 with a value of the loop target you defined in step 1.

3. Insert a MIDI controller 14 event with a value of the number of times to loop (up to 127 times). If you set the value to a negative number, the MIDI file will loop forever. Insert controller 14 right after the controller 13 event.

4. You can loop for longer than the value you assigned for controller 14 by setting the loop count value in synth.s. For example, setting this value to 128 will cause the MIDI file to loop infinitely.

Example Files

The following code listings are examples of the four files (`makefile`, `parse.cnf`, `synth.cnf`, and `synth.s`) you need to modify when preparing music for the Jaguar.

makefile

```
#==========================================
# Makefile      MUSIC DRIVER
#==========================================

SYNTHPATH = /jaguar/music/fulsyn

#==========================================
# Use 'erase' and 'rename' on MS-DOS
# Use 'rm' and 'mv' on Atari w/ csh
#==========================================
ERASE  = erase
RENAME = rename

#==========================================
# MIDI FILE WITHOUT EXTENTION (!!)
#==========================================

MIDIFILE = cscale

#==========================================
# MIDI Parser flags
#==========================================

PARSERFLAGS = -q

#==========================================
# Assembler & Linker flags
#==========================================

MACFLAGS = -fb -i$(SYNTHPATH);$(MACPATH)
ALNFLAGS = -g -e -l -a 802000 x 4000

#==========================================
# Default Rules
#==========================================

.SUFFIXES:      .scr .mid

.mid.scr:
  parse $(PARSERFLAGS) -o $*.out $*.mid
  mac $(MACFLAGS) -o$*.scr $*.out
  $(ERASE) $*.out

#==========================================
.SUFFIXES:      .out .mid
```

```
.mid.out:
  parse $(PARSERFLAGS) -o $*.out $*.mid

#==========================================
.SUFFIXES:      .scr .out

.out.scr:
  mac $(MACFLAGS) -o$*.scr $*.out

#==========================================
.SUFFIXES:      .o .s

.s.o:
  mac $(MACFLAGS) $*

#==========================================
.SUFFIXES:      .oj .das

.das.oj:
  mac $(MACFLAGS) -o$*.oj $*.das

#==========================================

FULSYN    = $(SYNTHPATH)/fs02_50.oj
OBJS      = driver.o synth.o $(MIDIFILE).scr
SCORE     = $(MIDIFILE).scr
EXEC      = test.cof

#==========================================
# EXECUTABLES
#==========================================

$(EXEC): $(OBJS) $(FULSYN)
  aln $(ALNFLAGS) -o $(EXEC) $(OBJS) $(FULSYN)

#==========================================
# Dependencies
#==========================================

driver.o:       driver.s synth.cnf $(SYNTHPATH)/fulsyn.inc

synth.o:        synth.s synth.cnf $(SYNTHPATH)/fulsyn.inc

$(MIDIFILE).scr: $(MIDIFILE).mid

$(FULSYN):            $(SYNTHPATH)/fs02_50.das synth.cnf $(SYNTHPATH)/fulsyn.inc
  mac $(MACFLAGS) -o$*.oj $*.das

#==========================================
#         EOF
#==========================================
```

parse.cnf

```
* File:         parse.cnf
* Description:  MIDI information file for the parse utility.
* Project:
* Composer:
* Date:
*
* Format: Change the data in this file according to the
*         following format.
*
* n = max_note_polyphony (default is 8 note polyhony)
* midi_channel - 1: lowest_note - highest_note patch_number transpose_value
*

n = 8                 ; 8 note polyphony
0: 36-36 0 0     ; kick
0: 42-42 1 0     ; clsdhat
0: 38-38 3 0     ; snare
3: 43-55 6 0     ; bass
```

synth.cnf

```
;------------------------------------------------------------------------
; This is a simple sample program to play a tune on the synth code.
;
; MODULE: SYNTH CONFIGURATION FILE
; DESCR: THIS FILE CONTAINS THE FULSYN CONIFGURATION
; (WHICH MODULES TO INCLUDE), GLOBAL VOLUME, SCLK, etc.
;
; COPYRIGHT 1992,1993,1994 Atari U.S. Corporation
; UNAUTHORIZED REPRODUCTION, ADAPTATION, DISTRIBUTION,
; PERFORMANCE OR DISPLAY OF THIS COMPUTER PROGRAM OR
; THE ASSOCIATED AUDIOVISUAL WORK IS STRICTLY PROHIBITED.
; ALL RIGHTS RESERVED.
;------------------------------------------------------------------------
;
;________________________________________________________________________
; Configuration for Fulsyn.
; To save DSP memory, turn only those module on that are needed.
;________________________________________________________________________

  ON             equu   1
  OFF            equ    0

  FMSIMPLE_MOD   equ    ON
  FMCMPLX_MOD    equ    OFF
  FMENV_MOD      equ    ON
  WAVEFM_MOD     equ    ON
  WAVEFM2_MOD    equ    ON
  WAVETAB_MOD    equ    ON
  SMPL8_MOD      equ    OFF
  SMPL16_MOD     equ    OFF
  CSMPL16_MOD    equ    ON
  SMPLENV_MOD    equ    OFF
  CSMPLENV_MOD   equ    ON

;________________________________________________________________________
```

```
; The following is for the note on/off modules.
; This section does not need to be edited.
;____________________________________________________________________

  WAVEFM_NOTE    equ   WAVEFM_MOD + WAVEFM2_MOD
  FMCMPLX_NOTE   equ   FMCMPLX_MOD
  FM_NOTE        equ   FMSIMPLE_MOD + FMENV_MOD
  SMPL_NOTE      equ   SMPL8_MOD+SMPL16_MOD+CSMPL16_MOD+SMPLENV_MOD+CSMPLENV_MOD
  WAVETAB_NOTE   equ   WAVETAB_MOD

;____________________________________________________________________
; SET GLOBAL & MIDI VOLUME
;____________________________________________________________________
  GLOBALVOLUME   equ   $7fff
  MIDIVOLUME     equ   $7fff

;____________________________________________________________________
; SET SCLK
;____________________________________________________________________
  SCLKVALUE      equ   19

;____________________________________________________________________
;___________________________________EOF______________________________
```

synth.s

```
;--------------------------------------------------------------------
; This is a simple sample program to play a tune on the synth code.
;
; MODULE: SYNTH DATA FILE
; DESCR:  THIS FILE CONTAINS THE PATCHES, SAMPLES, ENVELOPES,
; USER WAVEFORMS AND AN INITIALIZED VOICE TABLE.
;
; COPYRIGHT 1992,1993,1994 Atari U.S. Corporation
;
; UNAUTHORIZED REPRODUCTION, ADAPTATION, DISTRIBUTION,
; PERFORMANCE OR DISPLAY OF THIS COMPUTER PROGRAM OR
; THE ASSOCIATED AUDIOVISUAL WORK IS STRICTLY PROHIBITED.
; ALL RIGHTS RESERVED.
;--------------------------------------------------------------------

;--------------------------------------------------------------------
; INCLUDE FILES
;--------------------------------------------------------------------
  .include      'jaguar.inc'
  .include      'fulsyn.inc'
  .include      'synth.cnf'
;--------------------------------------------------------------------
; DATA SECTION
;--------------------------------------------------------------------
  .data
  .even
;********************************************************************
;**                     EDIT AFTER THIS POINT                      **
;********************************************************************
```

```
;------------------------------------------------------------------
; PATCHES
;------------------------------------------------------------------
patches::
  dc.w    1             ; NUMBER OF PATCHES

; Patch 0
  .include 'patches\\strlow.ptc'    ; strlow patch
                                    ; uses 's_strlow' sample
                                    ; and 'e_strlow' envelope

;------------------------------------------------------------------
; SAMPLES
;------------------------------------------------------------------

strlow_s:
  .incbin "samples\\synstrgs.cmp" ; sample used in patch 0

;------------------------------------------------------------------
;                          +++ START OF DSP SECTION +++
;------------------------------------------------------------------
  .DSP

TABS_COPY::
  dc.l    TABSSTART             ; DO NOT EDIT THIS LABEL
  dc.l    TABSEND - TABSSTART         ; DO NOT EDIT THIS LABEL

;------------------------------------------------------------------
; INITALIZED VOICETABLE
; A zero in the first field tells FULSYN that this is the last voice
; to be used!
;------------------------------------------------------------------
  .ORG    tablestart

TABSSTART::                              ; DO NOT EDIT THIS LABEL
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 0
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 1
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 2
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 3
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 4
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 5
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 6
  dc.l     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 7-LAST
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 8
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 9
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 10
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 11
  dc.l    -4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  ; voice 12
  dc.l    0

;------------------------------------------------------------------
; USER WAVEFORMS
;------------------------------------------------------------------
```

```
;-----------------------------------------------------------------------
; ENVELOPES
;-----------------------------------------------------------------------

strlow_e::
  .include "slopeenv\\string5.env"  ; envelope used in patch 0

;***********************************************************************
;**                    EDIT UP TO THIS POINT                          **
;***********************************************************************

; have slop for sloppy loader
  .dc.l   0,0
TABSEND::                          ; DO NOT EDIT THIS LABEL
  .dc.l   0
  .end
;____________________________________EOF________________________________
```

EEPROM Access Library

The Jaguar provides several options for game developers to store non-volatile game information such as high scores, options, saved games, music/sound effect levels, etc… while the unit is powered down.

Standard Cartridge E^2PROM (128 bytes)

Standard Jaguar Cartridge PCB's are currently equipped with a 128 byte E^2PROM for non-volatile storage. Developer Alpine boards also contain a compatible part for use in game testing. These parts are rated for approximately 100,000 write cycles before failure though we have achieved a much higher number of successful writes in our testing.

In order to provide compatibilty with the parts we use in manufacturing, we supply tested code which must be used to access the E^2PROM. ***This code should not be modified in any manner unless prior approval is granted by Atari Corp.*** The JAGUAR\SOURCE\EEPROM directory contains EEPROM.S, which has six functions used for reading, writing, and performing checksums on this data. Use of these functions requires that a valid stack pointer has been set in A7. These functions are as follows:

eeReadWord

Input	d1.w = E^2PROM address to read from.
Register Usage	Preserves all other registers.
Returns	d0.w = Value read
Purpose	This function reads one 16-bit word (address #0–62) from the E^2PROM. This function pays no attention to the checksum and therefore has no way to be sure the data is valid. A call to ***eeValidateChecksum*** will ensure that successive calls to ***eeReadWord*** will return valid data.

eeWriteWord

Input	d1.w	E^2PROM address to write to.
	d0.w	Data to write.
Register Usage	Preserves all other registers.	
Returns	d0.w	0 → Successful. 1 → Write failed.
Purpose	This function attempts to write one 16-bit word (address #0–62) to the E^2PROM. This function does not update the checksum and will thus cause any subsequent calls to ***eeReadBank*** or ***eeValidateChecksum*** to fail. The function ***eeUpdateChecksum*** must be used after any series of ***eeWriteWord*** calls to make the checksum valid again.	

eeReadBank

Input	a0.l	Address of a buffer 63 16-bit words in length to receive data from the E^2PROM.
Register Usage	Preserves all other registers.	
Returns	d0.w	0 → Successful. 1 → Checksum invalid.

Purpose	This function reads 63 16-bit words from the E^2PROM into a supplied buffer and validates the data against the stored checksum to ensure the data read is good.

eeWriteBank

Input	a0.l Address of a buffer containing 63 16-bit words to write to the E^2PROM.
Register Usage	Preserves all other registers.
Returns	d0.w 0 → Successful. 1 → Write failed.
Purpose	This functions stores 63 16-bit words supplied to it in the E^2PROM, checksums the data, and stores the checksum at address #63. We recommend that this function only be used when a large amount of data needs to be stored since this counts as 64 writes against the 100,000 rated limit. If you only change a couple of words, use ***eeWriteWord***(s) followed by ***eeUpdateChecksum***.

eeUpdateChecksum

Input	None.
Register Usage	Preserves all other registers.
Returns	d0.w 0 → Successful. 1 → Checksum write failed.
Purpose	This functions checksums the first 63 16-bit words from the E^2PROM and stores the checksum at address #63.

eeValidateChecksum

Input	None.
Register Usage	Preserves all other registers.
Returns	d0.w 0 → Successful. 1 → Checksum invalid.
Purpose	This function checksums the first 63 16-bit words from the E^2PROM and compares the checksum to the value stored at address #63. This function does not change any stored data.

Extended Cartridge E^2PROM (16k)

We are currently in the design phase of a new cartridge PCB which will contain a 16k E^2PROM. Third-parties will be able to request this PCB to provide access to the greater amount of storage. Because this project is still under development, no further details are available yet. Atari will notify developers when this part becomes available.

CD-ROM NV-RAM Storage Cartridge

Because CD-ROM titles do not normally have access to non-volatile storage, Atari will be making available a Flash ROM cartridge as a consumer product that give end-users the option to save high scores and game information. The protocols for accessing this cartridge are given in the **NV-RAM Cartridge Access Library** section.

NV-RAM Cartridge Access Library

Because CD-ROM titles do not normally have access to non-volatile storage, Atari will make available a special NV-RAM cartridge as a consumer product. This will give end-users the option to save high scores, setup options, and saved game information for their CD-ROM games. This cartridge is accessed by your program through the NV-RAM cartridge library.

Overview

These calls are provided to allow developers writing CD-ROM based games to save game information into a special cartridge containing non-volatile Flash ROM memory in an efficient and easy to use manner. There will be 128K bytes available in NV memory in the first version of the hardware (later cartridges may include more or less memory, so developers should use the **Inquire** function to determine the actual space available). This memory will be used and allocated in a file system-like manner, so that multiple games may use the same non-volatile memory cartridge without conflict, and so that different cartridge sizes may easily be supported. The ***NVM_Bios*** calls are thus much like the GEMDOS or MS-DOS file system calls.

The length of each block of memory is some multiple of 512 bytes. Memory blocks must be given a size when they are created, and cannot exceed that size later. The total number of memory blocks depends on the size of the cartridge being used, but as long as you use the ***NVM_Bios*** calls you will be able to deal with whatever is available.

A memory block is uniquely identified by two strings: the application which created it, and a block-specific name (its "filename"). The application name is available so that users may quickly identify which applications are associated with which blocks of memory. Application names may be up to 15 characters in length, and file names may be up to 9 characters in length. Both application and file names must use only characters chosen from the following 40 character set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 : ' . space

Accessing NV Memory

There are eleven calls provided to access NV memory. When the calls are available, a magic cookie with the value '_NVM' (0x5F4E564D) will exist at address $2400, and a dispatcher will exist at $2404. To invoke a function, do a 68000 JSR to location $2404 with the opcode and parameters described on the following pages.

All of the functions return a 32 bit value in d0, although in many cases only the lower 16 bits will be of interest. If bit 31 of d0 is set (i.e. if d0.l is negative) then an error has occured. The following error codes are defined:

Error Name	Code	Description
ENOINIT	-1	the Initialize function has not yet been called
ENOSPC	-2	there is not enough free space for the operation
EFILNF	-3	the file was not found

Error Name	Code	Description
EINVFN	-4	an attempt was made to use an invalid function
ERANGE	-5	an attempt was made to seek out of range
ENFILES	-6	no more file handles are available
EIHNDL	-7	invalid handle passed to function

The following functions are available:

Function	Opcode
Initialize	0
Create	1
Open	2
Close	3
Delete	4
Read	5
Write	6
Search First	7
Search Next	8
Seek	9
Inquire	10

Initialize — Opcode 0

C Prototype	int NVM_Bios(short opcode = 0, char *app_name, char *work_area)

68000 Assembly Example	`pea work_area`
	`pea app_name`
	`move.w #0,-(sp)`
	`jsr NVM_Bios`
	`adda.l #10,sp`

Returns	0
Purpose	Initialize must be called before any other ***NVM_Bios*** function. Its purpose is to initialize the ***NVM_Bios*** system, and also to identify the current application to the NVM Bios. The application name (a null terminated string satisfying the rules listed above) is passed as the parameter *app_name*. All subsequent ***Create*** and ***Open*** operations will use this application name for the memory blocks being created or opened. The second parameter (*work_area*) must point to a 16K, phrase aligned buffer which the NVM Bios may use as a scratch buffer. Applications need not preserve the contents of this memory between ***NVM_Bios*** calls (i.e. they can also use it for other purposes when not using the NVM Bios) but they must be aware that the buffer will be modified by all ***NVM_Bios*** calls. In other words, you can do what you want with the 16K between ***NVM_Bios*** calls, but every time you call ***NVM_Bios*** the 16K will be trashed. It is legal to call ***Initialize*** more than once; indeed, this is the only way for applications to open another application's memory blocks or for an application to change the location of the 16K ***NVM_Bios*** buffer. Please note that calling ***Initialize*** will invalidate all currently open handles (returned by ***Create*** or ***Open***). All other ***NVM_Bios*** functions will return **ENOINIT** if called before the first call to ***Initialize***.

Create
Opcode 1

C Prototype	int NVM_Bios(short opcode = 1, char *file_name, long file_size)

68000 Assembly Example	`move.l file_size`
	`pea file_name`
	`move.w #1,-(sp)`
	`jsr NVM_Bios`
	`adda.l #10,sp`

Returns	A non-negative handle on success ENOINIT if the Initialize function has not yet been called ENOSPC if there is insufficient room to allocate the file
Purpose	***Create*** should be used to allocate a specified number of bytes from backup memory. The parameter *file_name* should point to a name for the memory block. If the current application (specified by the ***Initialize*** call) already has a memory block with the same name, then that block will be deleted and a new one created (i.e. the new block will replace the existing one). The *file_size* parameter should contain the size in bytes required for the block. This size will be rounded up to the nearest multiple of 256 before being used for allocation. Note that multiple applications may have files with the same name, without affecting one another; ***Create*** will only delete an existing file if both the file name AND the application name match. The file handle returned by ***Create*** must be used in any ***Read***, ***Write***, or ***Seek*** calls referring to this file. WARNING: do not make this call if there is an existing file handle (returned by a previous ***Create*** or ***Open*** call) referring to a file with the same name as the new file being created. Use the ***Close*** call to close all such file handles before re-creating the file.

Open
Opcode 2

C Prototype	int NVM_Bios(short opcode = 2, char *file_name)

68000 Assembly Example	`pea file_name`
	`move.w #2,-(sp)`
	`jsr NVM_Bios`
	`adda.l #6,sp`

Returns	A non-negative handle on success EFILNF if the application has no file with the given name
Purpose	Instructs the Bios to attempt to access the blocks of memory owned by the current application (as set in Initialize) and whose file name is *file_name*. The *file_name* parameter must point to a null terminated file name string of an existing file. As with the ***Create*** call, ***Open*** will search only for files owned by the current application; it will not open a file owned by a different application, even if the file names are the same. The handle returned by ***Open*** must be used in any ***Read***, ***Write***, or ***Seek*** calls referring to this file.

Close Opcode 3

C Prototype	int NVM_Bios(short opcode = 3, short handle)
68000 Assembly Example	move.w #handle,-(sp) move.w #3,-(sp) jsr NVM_Bios adda.l #4,sp
Returns	0 on success EIHNDL if passed an invalid handle
Purpose	Used by an application to indicate that it is finished working with a file previously opened by Open or Create. After the call to Close, the handle passed to close becomes invalid, and no further Read or Write calls on that handle will succeed.

Delete Opcode 4

C Prototype	int NVM_Bios(short opcode = 4, char *app_name, char *file_name)
68000 Assembly Example	pea file_name pea app_name move.w #4,-(sp) jsr NVM_Bios adda.l #10,sp
Returns	0 on success EFILNF if no file matching the given application name and file name is found
Purpose	Deletes a file, freeing the memory associated with it. Any application may delete any other application's file, by passing in the application name and file name (as determined by ***Search First*** and ***Search Next***) in *app_name* and *file_name* respectively. Note that applications should never delete files belonging to other applications unless specifically requested to do so by the user . If an application needs more space than is available on the cartridge, then it should tell the user and offer him or her the choice of either aborting the current operation or of selecting one or more files to delete from the cartridge. WARNING: do not make this call if there is an existing file handle (returned by a previous ***Create*** or ***Open*** call) referring to the file being deleted. Use the ***Close*** call to close all such file handles before deleting the file.

Read Opcode 5

C Prototype	long NVM_Bios(short opcode = 5, short handle, char * bufptr, long count)
68000 Assembly Example	move.l count,-(sp) pea bufptr move.w handle,-(sp) move.w #5,-(sp) jsr NVM_Bios adda.l #12,sp
Returns	number of bytes read in d0, if successful EIHNDL if passed an invalid handle

Purpose	The ***Read*** call may be used to fill a buffer pointed to by *bufptr* with *count* number of bytes from the file specified by *handle* (returned from a previous ***Open*** or ***Close*** call). The read will begin at the current position in the file. This position is initialized to 0 by ***Open*** or ***Create***, is incremented by ***Read*** and ***Write*** (by the number of bytes read or written, respectively), and may be changed by ***Seek***. The game code must provide a buffer large enough to hold *count* number of bytes. If successful, the call will return the number of bytes read. At the end of the file (i.e. when the file's current position exceeds its size) 0 bytes will be returned by ***Read***.

Write — Opcode 6

C Prototype	long NVM_Bios(short opcode = 6, short handle, char *bufptr, long count)

68000 Assembly Example	`move.l count,-(sp)`
	`pea bufptr`
	`move.w handle,-(sp)`
	`move.w #6,-(sp)`
	`jsr NVM_Bios`
	`adda.l #12,sp`

Returns	number of bytes written in d0, if successful EIHNDL if passed an invalid handle
Purpose	The ***Write*** call may be used to write *count* number of bytes from the file specified by *handle* (returned from a previous ***Open*** or ***Close*** call). The write will begin at the current position in the file. This position is initialized to 0 by ***Open*** or ***Create***, is incremented by ***Read*** and ***Write*** (by the number of bytes read or written, respectively), and may be changed by ***Seek***. The number of bytes actually written to the file is returned. This may be less than *count* if, for example, an attempt is made to write more bytes to the file than the space allocated for it in ***Create***.

Search First — Opcode 7

C Prototype	int NVM_Bios(short opcode = 7, NV_FILEINFO *search_buf, long search_flag)

68000 Assembly Example	`move.l search_flag,-(sp)`
	`pea search_buf`
	`move.w #7,-(sp)`
	`jsr NVM_Bios`
	`adda.l #10,sp`

Returns	0 on success EFILNF if no files match the search

Purpose	The ***Search First*** call can be used in conjunction with the ***Search Next*** call to browse through the backup memory table of contents. This can be useful for displaying to the user all of the games whose information is backed up on a given cart. It can also be used by a game to obtain application and file names to be used in the ***Delete*** call to make room on a cartridge for its own information. The game player must be given final authority on this type of action.
	
	The *search_buf* parameter should point to a word-aligned 30 byte buffer used as a structure as shown below:
	
	`typedef struct`
	`{`
	`        long    size;`
	`        char    app_name[16];`
	`        char    file_name[10];`
	`} NV_FILEINFO`
	
	If the search is successful, the *size* field will be filled in with a long word giving the total size of the file. The *app_name* field will be filled with a null terminated character string giving the name of the application that created this file. The *file_name* field will be filled with a null terminated string consisting of the name the application gave to the file. These two strings constitute the *app_name* and *file_name* parameters for the ***Delete*** call.
	
	The *search_flag* parameter must be either 0 or 1. If it is zero, then the search will include all files on the cartridge, regardless of which application created them. If it is one, only files created by the current application (as specified by the last call to Initialize) will be included in the search. The value of *search_flag* will be used in subsequent ***Search Next*** calls as well.

Search Next Opcode 8

C Prototype	int NVM_Bios(short opcode = 8, NV_FILEINFO *search_buf)

68000 Assembly Example	`pea      search_buf`
	`move.w   #8,-(sp)`
	`jsr      NVM_Bios`
	`adda.l   #6,sp`

Returns	identical to Search First
Purpose	To be used in conjunction with ***Search First*** to provide the caller with table of contents information. This call can be made successive times until EFILNF is returned in d0. This will mean that no other entries exist in backup memory. See the entry for ***Search First*** for the definition of the NV_FILEINFO structure.

Seek Opcode 9

Prototype	long NVM_Bios(short opcode = 9, short handle, long offset, short flag)

68000 Assembly Example	`move.w flag,-(sp)`
	`move.l offset,-(sp)`
	`move.w handle,-(sp)`
	`move.w #9,-(sp)`
	`jsr NVM_Bios`
	`adda.l #10,sp`

Returns	the new file position, if successful EIHNDL if passed an invalid handle ERANGE if the offset would be past the end of file
Purpose	Resets the file position (used by ***Read*** and ***Write***) for the file whose file handle (as returned by ***Open*** or ***Create***) is handle to be at offset bytes from the beginning of the file (if flag is 0) or from the current position in the file (if flag is 1). Subsequent ***Read*** or ***Write*** calls will begin their operations at this point (and will update the file position as usual).

Inquire Opcode 10

Prototype	int NVM_Bios(short opcode = 10, long *totspc, long *freespc)

68000 Assembly Example	`pea freespc ; Ptr to 'freespc' variable somewhere in RAM`
	`pea totspc ; Ptr to 'totspc' variable somewhere in RAM`
	`move.w #10,-(sp)`
	`bsr NVM_Bios`
	`adda.l #10,sp`

Returns	0 on success
Purpose	Inquires about the amount of space available on the cartridge. The *totspc* parameter points to a long word which is filled in with the total amount of cartridge memory which may be used for applications (i.e. the size of the largest possible memory block, assuming it is the only memory block on the cartridge). The *freespc* parameter points to a long word which is filled in with the amount of cartridge memory currently free (i.e. the size of the largest memory block which could be created at the present time). (Note that the amount of free memory is not the only constraint on the ***Create*** call; even if there is sufficient space for a memory block, ***Create*** may return ENOSPC if there is no room left in the cartridge's table of contents.)

Using The NV-RAM Simulator

The NV-RAM Simulator allows you to use an Alpine board plugged into your Jaguar CD-ROM development station to simulate a NV-RAM cartridge during the development process. It provides the same functions for accessing NV memory as described in the previous section.

The NV-RAM Simulator is normally located in the JAGUAR\NVRAMSIM directory. To use it, load the debugger and then type:

```
load nvmsim.db
```

The NVRAM BIOS will be installed into your system and then control will return to the debugger. At this point you may load and execute your main program.

The Alpine board's memory from $900000 to $91FFFF will be used to hold the cartridge data. A sample disk image (full of files containing random data) is included with the simulator. The file is called DISKIMG.IMG. To load this file, type "`read diskimg.img 900000`" while in the debugger. The debugger script NVMTEST.DB is also included. It will load both the NV-RAM simulator and the sample cartridge files in one easy step.

Keep in mind that the Alpine board's memory switch must be set for "write enable" in order for the simulator to work. Also keep in mind that any program or debugger script that clears DRAM below $4000 will erase the simulator from memory.

Save Cartridge Manager

If you hold down the "Option" key (and keep it held down) before typing the "`load nvmsim.db`" or "`load nvmtest.db`" command in the debugger, you will be presented with the *Save Cartridge Manager* screen. This is a sample application which users will also be able to access in order to delete files. ***(Please note that the existence of the*** *Save Cartridge Manager* ***does not excuse individual applications from providing similar functionality themselves!!!).*** The Save Cartridge Manager uses the following keys:

up arrow/down arrow	Selects files
A,B,C	To delete a file
OPTION	To choose how to sort files
OPTION + 1	To save preferences in a file
OPTION + 7 + 9	To create a (dummy) file
OPTION + * + #	To erase all files
OPTION + * + 0 + #	To do a test of free memory
* + #	To exit the manager

Once the Save Cartridge Manager has run, the BIOS will be copied to RAM (at $2400). You can then reset the machine and load and run your own application. The BIOS will remain in RAM until the machine is powered off.