Developer Documentation

Version 4.0
April 24, 1990

Evan Brooks
Robert Currie
Digidesign Inc.

Confidential

Contents

Introduction

This manual describes Digidesign's family of digital audio hardware and development software. The emphasis is on technical details of the hardware as well as software issues related to third party development. Since the first version of this documentation appeared in 1988, the number of registered developers, both commercial and academic, along with the number of hardware products has grown faster then our ability to provide support and documentation. This manual represents a comprehensive description of our hardware products to date. Over the next year, the number of hardware products will grow even more. As new products are released, we will provide additional developer documentation in the form of additions to this manual as well as additional sample source code. As the first third party commercial products and research projects using our hardware are now appearing, we hope that through our support, developers will continue to utilize our hardware in creative and new ways.

If you have been programming our hardware and/or Macintoshs for a while, you will have noticed that things seem to be getting quite complicated (as if they weren't complicated enough already) and in many ways non-orthogonal. We plan to continue to develop high end hardware such as the Sound Accelerator, as well as low end consumer products such as the Audio Media card. This dichotomy usually translates into 'compatibility hell' for programmers. Given that Apple will be changing its hardware and operating system drastically in the near future, and that our hardware will continue to evolve, please bear with various additions and exceptions that enable everything to work together. Trust that as we come out with new hardware, and Apple changes the rules, we will try to make everything work together, but this will most likely be at the expense of pure elegance. We STRONGLY recommend that you adopt some sort of object oriented approach to our hardware (as well as Apple's) so that the changes will be easier to weather.

- ✔ The core of Digidesign's hardware is the Sound Accelerator. The majority of this manual relates to programming the Sound Accelerator, but the ideas and techniques are almost identical when dealing with other hardware (such as the Audio Media card). As future version of the Sound Accelerator are released and documented, the general sections in this manual on the Sound Accelerator will still be applicable and relevant, while specific details will be addressed in updates similar to the hardware descriptions at the end of the manual. Unless specifically stated, Sound Accelerator refers to any of our cards, and Mac II means any Mac II class machine (II,IIx,IIcx,IIci,IIfx and SE30 cards).

Organization Of This Manual

The information in this manual is organized loosely into three sections. The first covers overall system information. The second covers related software such as the driver, snth, and example source code. The third covers specific hardware such as the Sound Accelerator, Audio Media, and DAT I/O and is organized in the order that the actual products were introduced. As future hardware products are introduced, additions to the third section will be provided.

Required Background Materials

This documentation assumes that you are already familiar with writing a Macintosh application. The Macintosh programming environment is extremely complicated, yet exceedingly rich. Although you may be able to start programming our hardware concurrently with learning how to program the Mac, we suggest you become familiar with programming the Mac first.

The 'bible' for programming the Mac is Inside Macintosh Volumes 1-5. Certain sections are essential for even the most trivial Macintosh programs, while other sections are mandatory when programming our hardware. Particularly important sections that relate to our hardware are:

- The 'snth' resource documentation assumes that you are familiar the Sound Manager as described in Inside Macintosh Volume 5.
- Operations on sound files, as well as their various formats assume that you are familiar with the HFS file system as described in Inside Macintosh Volume 4 as well as the Resource Manager chapter in Inside Macintosh Volume 1.
- The driver documentation assumes that you are familiar with Device Manager chapters of Inside Macintosh, Volumes 1-5.
- In order to make use of the sample programs you are expected to have a basic syntactic level of familiarity with the Motorola 56000 assembler, as well as with the 56000 instruction set. The example application code is written primarily in MPW Pascal, 68000 Assembler, and 56000 Assembler.
- A basic knowledge of MPW and the MPW Shell editor is also assumed, although it is not required if you are not using MPW to develop your software. Our development environment at Digidesign is MPW and MacApp. The sample source code is primarily in MPW Pascal and Assembler, with the more extensive examples in Objective Pascal and MacApp. Many of our developers use Think C. The major disadvantage to using Think C is the need to use the standalone versions of the 56000 Assembler, Linker and Parse56k, where as in MPW these are all shell tools allowing a completely automated build. If you plan to do extensive 56000 code development, we suggest you consider MPW due to its ability to use shell tools as well as the ability to compile multiple languages. If you plan to only use a few 56000 algorithms (for example, only playing and recording sound), then either environment is suitable and the choice will be your personal preference.

Getting Started

For those applications that only need high-quality playback of short digital audio data (ie a few seconds long), the simplest and quickest way to play sound is the Sound Manager 'snth' resource. See the Sound Manager 'snth' Resource section for more information.

For applications that need to directly access the hardware (such as playing long sounds from a hard disk, or digitally processing sound or data), you will need to come up a bit of a learning curve. The following is a suggested path:

- Read the article "Digidesign's Sound Accelerator: Lessons Lived and Learned". It will give you a general idea about the core of our hardware (The Sound Accelerator) as well as an idea of how you go about programming it.
- Look over and compile the source code to SASample as a simple example of playing mono 16 bit sound.
- Look over the rest of this manual, in particular the section titled Sound Accelerator Rev. A.
- Look over the source code to DSPWorkshop, which contains more advanced uses of the hardware as well as how to access various peripheral hardware. Specifically, the object TSACard in the files USACard.p and USACard.a if the definitive source for driver structures and card I/O examples.

- If you plan to program the 56000 extensively, read chapters 1 through 7 Motorola's DSP56000/DSP56000 Digital Signal Processor User's Manual (The 'Red Book').

Support Policies

Through its developer support program, Digidesign will supply its third-party developers with the information necessary to begin developing both hardware and software to be used in conjunction with our hardware and software.

Digidesign **WILL** provide technical support on subjects dealing with specific features of our hardware, such as the Sound Accelerator's expansion port, or how to set up the 56000's registers to access certain features of the card, such as stereo playback.

Digidesign **WILL NOT** be able to answer questions regarding Macintosh programming, 56000 programming, digital signal processing algorithms, or hardware design. For information on these subjects, refer to the following sources:

Macintosh Programming:

- Inside Macintosh, Volumes 1-5
  Apple Computer, Inc.

- Designing Cards and Drivers for the Macintosh II and Macintosh SE.
  Apple Computer, Inc.

56000 Programming:

- 56000 Programming and Hardware Design:
  Audio Cassette Learning Course
  Motorola Technical Operations, Phoenix
  (602) 244-7579
  Cost: $125.00

- Motorola Technical Training Seminars
  (800) 521-6274
  Cost: $795.00 for a single student

Digital Signal Processing:

- Digital Signal Processing
  A.V. Oppenheim and R. W. Schafer
  Prentice-Hall, Englewood Cliffs, NJ 1975,1988

- Theory and Application of Digital Signal Processing
  L. R. Rabiner and B. Gold
  Prentice-Hall, Englewood Cliffs, NJ 1975

- Appendix C in the 56000 Users Manual contains an excellent list of books ranging from general DSP to graphics and speech processing. In addition it details how to access Motorola's DSP bulletin board, Dr. Bub, which is a source of 56000 DSP code, and best of all it's free.

Writing Sound Accelerator Software

Applications which require customized use of the 56000 require the Motorola 56000 development software (assembler and linker). Typically an application stores already-compiled 56000 object code within the application itself (embedded in the 68000 assembly code) or in the Resource File of the application. When the application wants to use a card, it makes a driver call to allocate a card. If a card is available, the driver will return the base address of the card as well as other information about the card. It then makes another control call to the driver, and passes a pointer to a block of 56000 object code somewhere in the Mac's memory, and asks the driver to load it into the card. After the card is allocated and loaded with code, the application communicates directly with the card as a memory mapped peripheral. Typically this interaction is through the 56000's host port. When the application is finished with the card, it calls the driver to free the card so that other applications in the system can use the card.

- ✔ The differences between the Mac II and Mac SE versions of the Sound Accelerator are only in their register addressing. If you are planning on writing code that will run on both versions of the Sound Accelerator, we highly recommend that you have both a Mac II and Mac SE available, each with their respective Sound Accelerators, even if the actual code development is only done on the Mac II. Their are no software or register differences between the Mac II cards and the SE30 card.

Standard Development Tools

All the software tools run under the MPW environment as shell tools or as standalone applications. The Motorola CLAS-B software package includes the following tools:

asm56000 - 56000 macro assembler (MPW shell tool and standalone application)
lnk56000 - 56000 relocatable linker (MPW shell tool and standalone application)
sim56000 - 56000 simulator (standalone application)

Also available from Motorola is a C Compiler for the 56000, but this is not required for development. The CLAS-B package are available from Digidesign, separate from the standard Developer Documentation.

Included in the Developer Documentation is Parse56K, a tool that can read the 56000 linker output files, and generate a 68000-compatible assembler format source file, so that you may embed 56000 code within your Macintosh application. Parse56K can also output the 56000 code as a resource of any type and ID should you wish to keep your 56000 code data in resources. Parse56k is written in C and the source is included. Those using Think C may wish to customize its I/O interface. Also included are two example Macintosh applications with 56000 code as well helpful suggestions as to how to make the most efficient use of the Sound Accelerator.

Third Party Development Tools

If you are planning on writing custom DSP algorithms, Zola Technologies has developed an excellent environment called DSP Designer that is designed specifically for the development of digital signal processing algorithms and software. DSP Designer is a set of tools and scripts which expand the basic MPW environment, providing capabilities for design, analysis, and floating-point simulation of DSP systems, with particular emphasis on digital filters. Additional tools create assembly code to implement digital filters in Motorola 56000 assembly language that can be down loaded to a Sound Accelerator for processing. Their address is:

- Zola Technologies, Inc.
  6195 Heards Creek Dr., N.W.
  Suite 201
  Atlanta, GA 30328
  (404) 843-3913

Application Construction

The 56000 source code is typically written in the MPW shell editor. It is then assembled and linked using the Motorola software. Then, Parse56K is run on linker output creating a resource which is stored in the resource fork of your application. When your application wants to use the card, a simple GetResource() call produces the required code which the driver can then load. All the example code stores 56000 code in this manner.

If you need to embed 56000 code directly in your application itself (useful when writing INIT's, CDEV's, or DA's), Parse56k can produce Motorola 68000 assembler output consisting of DC directives, containing 56000 opcodes in Hex. The exact packing format of the 24 bit opcodes depends on whether or not you choose the -p (packing) option in Parse56K. See section on Parse56K for more details. You can then "INCLUDE" this output file within an assembler file in your Mac application. In this way, you can have 56000 object code stored within your application. An example script that creates a 68000 compatible output file is as follows:

```
asm56000 -l -B  YourProgram.asm
lnk56000 -B -MYourProgram.map YourProgram
Parse56K YourProgram.lod > YourProgram.a
```

The first line invokes the 56000 assembler on your 56000 code source file, called YourProgram.asm. The assembler output file is YourProgram.lnk. Then, the linker is invoked on the assembler output, producing the link output file YourProgram.lod. In practice, most 56000 code is written within a single file, and the linker is only run to catch misspellings and typos as it tries to resolve references. Then, Parse56K reads the linker output and generates a Motorola 68000 assembler file called YourProgram.a. The .map file is useful to find the memory addresses of various symbols when using the Motorola simulator.

The Device Driver

On a Mac II, each Sound Accelerator contains a piece of system software, called a Device Driver, on its configuration ROM. When the Macintosh boots up, this Device Driver is read into system memory. On a Mac SE, the Device Driver is loaded into the system by an INIT resource in the System Folder. Most of the communications that an application has with the Sound Accelerator, other than real-time data transfer, are done by talking to the card's driver.

The main purpose of the driver is to free the application from having to know which slot the card is in, how many cards are installed, and fighting with other applications over use of the card itself. We highly recommend and urge that you use the driver in the suggested fashion. The consequences of not doing so are that your code may no longer work in the future, on different Macintosh models, or different Sound Accelerator models. It may not work if the Sound Accelerator has been removed, or if it has been put into a different slot. And it will have a difficult time working and coexisting with other applications that are also trying to use the card, such as the operating system itself (for example, SysBeeps).

The driver is responsible for maintaining a list of all available Sound Accelerator cards installed in the system. This includes multiple cards in a Mac II, as well as multiple cards in a Mac SE expansion chassis (available from several third-party developers). When an application needs to use a card, it asks the driver to allocate a card from those available and free. The calling application can specify minimum memory sizes for the card it needs, as well as an application signature and algorithm ID number, so that the driver can select a card that has the requested DSP code already loaded into it.

If such a card is available, the driver returns a pointer to a block of information describing the card. If no card is available, the driver returns an error. Since the driver is solely responsible for allocating and distributing the cards to the running applications, many co-existing applications can use a card, as long as two do not need it at exactly the same time. To make the most efficient use of a card, your application should allocate a card only when it needs it, and free the card (with another control call to the driver) when it no longer needs it.

The driver control and status calls are summarized below, along with their passed parameters. Included in the Developer Documentation is some example code showing how to access and talk to the driver. Note that in a given Macintosh there is only one Sound Accelerator driver no matter how many Sound Accelerators there are in the Mac. The one driver takes care of all Sound Accelerators in the system. The driver itself is opened at boot time, and closed upon shut-down so you never need to call CloseDriver(). You only need to call OpenDriver() to get the driver's refNum when your application starts up. That call to OpenDriver just returns the driver's refNum since the driver is already open.

✔ This document describes the device driver present on the Rev. 4.2 Configuration ROM on the Mac II Sound Accelerator cards, and the Rev. 2.0 Sound Accelerator INIT for the Mac SE.

ctlKillIO

```
csCode = 1
```

This is called by a KillIO driver call. It calls ctlReset for each Sound Accelerator in the system.

ctlLoadDSPCode

```
csCode = 4
 -> csCardRefNum      EQU   csParam            ;refNum of card (word)
 -> csCodePtr         EQU   csCardRefNum+2     ;pointer to code to load
 -> csCodeSize        EQU   csCodePtr+4        ;number of words of 56k
                                               ;code to load
 -> csCodeUserSig     EQU   csCodeSize+4       ;User Signature of owner
 -> csCodeAlgorithm   EQU   csCodeUserSig+4    ;Algorithm ID of this code
 -> csForceLoad       EQU   csCodeAlgorithm+4  ;0 = don't load code if
                                               ;already loaded
                                               ;1 = load code regardless
                                               ;(byte)
 -> csPackingType     EQU   csForceLoad+1      ;0 = unpacked 32 bits
                                               ;1 = packed 24 bits
                                               ;(byte)
```

Load the 56000 object code pointed to by csCodePtr into the Sound Accelerator given by refNum, after first performing a board reset. csCodeSize indicates the number of 56000 words of program code to load (24 bits per 56000 word), and csPackingType indicates how this code is packed in the Mac's memory. 0 means that it is packed into consecutive 32-bit longwords, whose most significant word is ignored, and the lower 24 bits contain the 56000 program word. 1 means that the data is packed as consecutive 24 bit words, with no padding in-between words. sCodeUserSig is the 4-byte OSType signature of the application that is loading this 56000 code, and csCodeAlgorithm is a unique longword identifying this 56000 code as a unique algorithm within this application. A User Signature of ' ' (four spaces) combined with an Algorithm ID of 0 are considered illegal and should not be used. If the board referred to by refNum already has the given User Signature and already has the given Algorithm loaded into it, nothing happens and the call returns immediately.

ctlAllocCard

```
csCode = 5
 -> csAllocParamPtr  EQU   csParam            ;pointer to param block below

Allocate Parameter Block:
<-> csAllocFlags     EQU   0                  ;requested/returned flags (word)
 -> csUserSig        EQU   csAllocFlags+2     ;requested/returned user signature
 -> csAlgorithm      EQU   csUserSig+4        ;requested/returned user algorithm
<-> csXMemWords      EQU   csAlgorithm+4      ;requested/returned X memory size
<-> csYMemWords      EQU   csXMemWords+4      ;requested/returned Y memory size
<-> csPMemWords      EQU   csYMemWords+4      ;requested/returned P memory size
<-  csBaseAddr       EQU   csPMemWords+4      ;returned base address of 56000
<-  csInfoRecPtr     EQU   csBaseAddr+4       ;returned ptr to this card's
                                               ;info record
<-> csACardRefNum    EQU   csInfoRecPtr+4     ;requested/returned card
                                               ;refNum (word)

csAllocFlags bit fields
```



```
GetAnyAlgorithm     EQU   0             ;1 = ignore card's
                                        ;signature/algorithm
GetAnyMemSize       EQU   1             ;1 = get card regardless
                                        ;of mem sizes
GetAnyRefNum        EQU   2             ;0 = get card regardless
                                        ;of refNum (Rev 4+)
HasRequestedAlg     EQU   15            ;1 = card has requested
                                        ;sig/alg in it
```

This control call is used to allocate a Sound Accelerator card for use by an application. It has only one parameter, which is really a pointer to a large structure shown above. csAllocFlags is a word of flags which are both read and set by this control call. The msbit (bit 15) is set by the driver if the card it has allocated already has the User Signature and User Algorithm requested by the other parameters in this control call. The application can test for this, sot hat it may avoid calling the driver again to load that code into the board.

Bit 0 is set by the application if it wishes the driver to ignore any User Signature and Algorithm that might happen to be in a card, and choose a card regardless of its current contents.

Bit 1 is set by the application if it wishes the driver to ignore the amount of memory that a given card has, and choose the first available card regardless of how much memory it has on it. By not setting this bit, an application can force the driver to allocate only a card which has the necessary memory on it to perform a particular algorithm.

Starting with the Rev. 4 ROM, Bit 2 is set by the application if it wishes to allocate a specific card, whose refNum is passed in csACardRefNum. csUserSig and csAlgorithm are set by the Application to tell the driver that it would like a card with those two parameters already in it, if possible. Note that the user signature and algorithm ID for the card are not actually set until ctlLoadDSPCode is called!

csXMemWords, csYMemWords and csPMemWords are set by the application to tell the driver that the card it needs must absolutely have at least that amount of each type of memory on board.

csBaseAddr is set by the driver upon return to indicate the base address of the DSP chip on the allocated board. This address can be used to talk directly to the board for real-time transactions. This is set to NIL if the driver cannot allocate a card a requested.

csInfoRecPtr is a pointer to a structure containing data about the card which is internally used by the driver, and should not be changed by the application. Some of the information in this structure may be useful to more advanced applications. See the file USACard.p in DSPWorkshop for deatails on the structure of this record, as well as how to use the information in it. csACardRefNum is the refNum of the card allocated by the driver. This refNum is used in all future driver control and status calls to refer to the particular card that the driver just allocated. Starting with Rev. 4 ROM, if you set bit 2 of the flags word (GetAnyRefNum), you can pass a card refnum to the driver in csACardRefNum . The driver will attempt to allocate the card with that refNum, if it is not already allocated. This way, an application can ask for a specific card in a machine which has more than one in it. If the card asked for is not available, or if the refNum is invalid, the driver returns an error.

On NuBus machines, the card refNum is the same as its slot number. On the Mac SE, the card refNum is a positive integer, starting with 1, and additional cards in the system are given refNums of 2, 3, 4, etc.

ctlFreeCard


```
csCode = 6
 -> csCardRefNum      EQU   csParam           ;refNum of card (word)
 -> csCardStatus      EQU   csCardRefNum+2     ;Free status for InUse
                                               ;parameter (byte)
                                               ;<0 = free, but will be needed soon
                                               ;0 = free
```

The currently allocated Sound Accelerator board referred to by refNum is to be freed and de-allocated by the driver. The board's current User Signature and Algorithm will remain installed in it, and the board will not be reset.

The csCardStatus parameter lets the application specify several different types or categories of de-allocation. If csCardStatus is set to 0 by the application, the board is marked as "free", meaning that the driver can re-allocate it at any time, to any application wanting it. If csCardStatus is less than 0, then the board is marked as "free, but needed soon". This tells the driver that the next time the driver is called to allocate a board, it should attempt to allocate another board before allocating this one. Essentially, the driver will allocate a board marked "free" before it will allocate one marked "free, but needed soon", unless the requested User Signature and Algorithm match identically.

ctlAllocateAllCards

```
csCode = 7
```

Used during development only. All known Sound Accelerator cards in the system are marked as "In Use", and none will be able to be allocated until ctlFreeAllCards is called. This is useful for determining an application's response to a situation where there are no free cards available to it.

ctlFreeAllCards

```
csCode = 8
```

Used during development only. All known Sound Accelerator cards in the system are marked as "Free", regardless of whether or not they are currently in use by an application. This is useful for freeing cards that may have become allocated by an application which crashed, and left the card in an allocated state.

ctlInstallSEInterruptHandler

```
csCode = 9
 -> csCardRefNum      EQU   csParam           ;refNum of card (word)
 -> csIntHandler      EQU   csCardRefNum+2     ;pointer to interrupt
                                               ;routine (pointer)
 -> csIntA1Value      EQU   csIntHandler+4     ;value to load A1 with (longword)
```

This routine will install an interrupt handler for a specific Sound Accelerator card in a Macintosh SE only. Do not use this for Macintosh II code - you should use slot interrupts instead. csIntHandler is a ProcPtr to your interrupt handler code. The value you pass in csIntA1Value will be loaded into A1 before your interrupt routine is called. The Sound Accelerator interrupt handler will save and restore A0-A3 and D0-D3 - your interrupt handler must preserve the rest. Your interrupt handler will be called at interrupt level 1, and must return via an RTS with the interrupt level still at 1.

If you are writing code for both Mac II and Mac SE Sound Accelerators, you can use the same interrupt handler code (except for board register addressing differences), passing a pointer to it to SIntInstall on a Mac II (to use Slot Interrupts), and passing it to this driver control call on a Mac SE (to use level-1 interrupts). See the example code for more details.

ctlRemoveSEInterruptHandler

```
csCode = 10
 -> csCardRefNum      EQU   csParam           ;refNum of card (word)
```

This routine will remove the current interrupt handler for the given Sound Accelerator card in a Macintosh SE only. Do not use this for Macintosh II code - you should use slot interrupts instead.

ctlGetUser

```
csCode = 11
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the 4-byte OSType User Signature installed in the board referred to by refNum.

ctlGetAlgorithm

```
csCode = 12
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the User Algorithm ID installed in the board referred to by refNum.

ctlGetXMemWords

```
csCode = 13
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the total number of X Data memory words in the board referred to by refNum.

ctlGetYMemWords

```
csCode = 14
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the total number of Y Data memory words in the board referred to by refNum.

ctlGetPMemWords

```
csCode = 15
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the total number of Program memory words in the board referred to by refNum.

ctlGetClockRate

```
csCode = 16
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
```

Returns the 56000 crystal oscillator frequency in Hz for the board referred to by refNum.

✔ Note: Early versions of the driver (ROM Rev. 3 and lower) incorrectly returned the value of the frequency * 100! This has been fixed in the Rev. 4 ROMs and Rev. 2 SE INIT. See the file USACard for code that determines the version and corrects the value.

ctlGetCardCPUType

```
csCode = 17
 -> csCardRefNum      EQU   csParam           ; refNum of card (word)
 -> csResultPtr       EQU   csCardRefNum +2   ; ptr to longword return value
                                              ; msword = card type
                                              ; 0 = Sound Accelerator SE Rev. A
                                              ; 1 = Sound Accelerator Mac II Rev.A
                                              ; 2 = Sound Accelerator Mac II Rev.B
                                              ; 3 = Sound Accelerator SE/30 Rev.A
                                              ; 4 = Audiomedia Rev.A
                                              ; lsword = DSP CPU type
```

```
                                        ; 0 = 56000 Rev. A silicon
                                        ; 1 = 56000 Rev. B silicon
                                        ; 2 = 56000 Rev. C silicon
```

Returns a longword describing the type of Sound Accelerator board and type of DSP chip on that board. The ms word is the card type, and the lsword is the CPU type. refNum identifies the board in question. This call should be used to determine whether a Mac II or Mac SE Sound Accelerator is installed, as well as the version of 56000 silicon on that board. Different versions of silicon may have slightly different capabilities, and software developers that want to take advantage of any new features should use this call to determine the exact version of 56000 silicon they are using.

ctlResetAllCards

```
csCode = 18
```

Used during development only. All known Sound Accelerator cards in the system are reset, regardless of whether or not they are currently in use by an application. This is useful for resetting cards that may have become allocated by an application which crashed, and left the card in an allocated and possibly hung state.

ctlReset

```
csCode = 19
 -> csCardRefNum      EQU   csParam            ; refNum of card (word)
```

Perform a board reset on the board identified by refNum. The call returns after the board has been completely reset, and can be safely accessed. The UserSignature of the board is set to ' ' (four spaces) and the User Algorithm ID is set to 0 to indicate that no code has been loaded into this board.

stNumFreeCards

```
csCode = 2
 <- csResult          EQU   csParam          ; longword return result
```

Returns the total number of un-allocated Sound Accelerator cards installed in the system.

stNumUsedCards

```
csCode = 3
 <- csResult          EQU   csParam          ; longword return result
```

Returns the total number of allocated Sound Accelerator cards installed in the system.

stNumCards

```
csCode = 4
 <- csResult          EQU   csParam          ; longword return result
```

Returns the total number of Sound Accelerator cards installed in the system.

Sound Manager 'snth' Resource

If your application is currently using the standard Apple snth ID 5 (the sampled sound snth), then after installation of the Sound Accelerator snth, your application will automatically play sound through the Sound Accelerator instead. This requires almost no work, but you will only get 8-bit monophonic sound.

If you want 16 bit sound, or stereo sound, you will need to slightly modify your code that calls the Sound Manager. You will want to package your sounds as Chunky or Interleaved buffers (see below) to take advantage of the fact that the Sound Accelerator snth can play 16-bit stereo sounds. Of course, if your sounds currently are only 8-bit monophonic, you will need to re-record them in 16-bit mono or stereo.

On a Macintosh SE or Plus, System 6.0 or higher is required to be able to use the Sound Manager. It does not exist in earlier systems on these machines.

A 'snth' resource allows the Macintosh Sound Manager (not the Sound Driver!) to play sounds through a Sound Accelerator. In fact, when the Sound Accelerator 'snth' resource is installed in the system or an application, any calls that previously went to the Apple Sampled Sound Snth (ID 5) are automatically rerouted to play through the card, with no rewriting or recompiling of any software.

The immediate advantages of this approach are obvious: you plug the card in, run an install program to install the snth resource into your system and/or applications, and all sampled sounds that previously came out the Mac speaker will now come out of the card.

Of course, this can only work with sound produced by calls to the Sound Manager, not the Sound Driver, and only to those calls that are passed to snth ID 5 (the Sampled Sound snth). The resulting sounds come out in 8-bit, monophonic 44.1 KHz playback through the card. The bandwidth and pitch-shifting that the card provides are much superior to what you can get from the Macintosh speaker and audio output, so that even your existing 8-bit files will sound brighter and clearer.

To take full advantage of the Sound Accelerator, however, the snth resource will accept sampled sound data (using SoundCmd and BufferCmd commands) in two other formats.

One format, called the "Chunky" format, is designed to allow up to 16-bit stereo sound data, while still remaining compatible with the original Apple 8-bit monophonic sound hardware. The data format of this type of sound is not very convenient for editing or displaying, but you are guaranteed of forward and backward compatibility. A Macintosh with an available Sound Accelerator will play such a sound as a 16-bit stereo soundfile, while a Mac without a card will play the exact same file as an 8-bit mono (left channel of the stereo pair) soundfile through the Macintosh speaker.

The other format is much easier to edit and display, being stored in an interleaved format, but it is not backward compatible with the Apple sound hardware. It is forward compatible with future Digidesign sound hardware and software, though.

The choice between the current 8-bit mono format, and the two new 16-bit stereo formats is up to the developer. The Sound Accelerator snth resource can deal with all three. If an application needs to produce a sound through the Sound Accelerator, but there are no available cards at that time, or there are no cards installed in the Mac, the snth will re-route the request for sound to the original Apple snth resource, and the sound will then play out of the Mac speaker. Of course, if the

sound is stored in the Interleaved format, it cannot be passed on, since the original Apple snth resource cannot read this type of sound.

The Sound Accelerator comes with Sound Installer, a snth Installer/DeInstaller program. Normally, the snth needs to be installed only in the System file. However, for applications that have their own versions of Apple's ID 5 snth resource (such as HyperCard, Sound Designer and TurboSynth), the installer will install the new snth resource directly into those applications. The new snth will work with or without a Sound Accelerator card installed in the Mac, so de-installation is rarely necessary.

The Sound Accelerator snth will accept and respond to the following standard commands:

InitCmd, AvailCmd, NoteCmd, FreqCmd, SoundCmd, BufferCmd, ContBufferCmd, RequestNextCmd, QuietCmd, FreeCmd, howOftenCmd, gainCmd, releaseCmd, flushCmd.

In addition, it also responds to the following commands which support the alternate Chunky and Interleaved sound data formats:

ChunkyBufferCmd, ChunkyContBufferCmd, InterleavedBufferCmd, InterleavedContBufferCmd, ChunkySoundCmd, InterleavedSoundCmd.

The new commands are described and documented below. They are just variants on the standard BufferCmd and SoundCmd. The ContBufferCmd, and its counterparts for the Chunky and Interleaved commands, are identical to the regular BufferCmd and its counterparts, with the exception that the pitch-shifting algorithm on the DSP will reset its resampling phase at the start of each BufferCmd, but will not reset it for a ContBufferCmd.

The reason for this is to allow the smooth, continuous playback of a sound divided up into a series of consecutive buffers. To play such a sound, you would send a BufferCmd for the first packet of sound data, and then a ContBufferCmd for each packet of data thereafter.

snth Initialization

When your application calls SndNewChannel to allocate a new sound channel and snth resource, you pass an Init longword in the SndNewChannel call. This longword shows up in the param2 field of the initCmd which is passed to the snth after the snth has been loaded. Currently, only one bit in this init longword is defined. All other bits are ignored, but should be set to 0 for future compatibility.

```
ForceAppleSNTH = $80000000
```

When this bit is set in the init longword, the snth will act as if it is unable to allocate a Sound Accelerator card, and attempt to locate and load the original Apple snth ID 5 (or its variants) instead. The result is that any Normal or Chunky buffers passed to the snth will end up playing out of the Mac's speaker in 8-bit resolution.

This is useful for applications which want to allow the user to disable the use of an installed card. Perhaps the user does not have any headphones or amplifiers available at a particular instant, and needs to get audio output. In any case, it allows the application to override the snth's desire to use a Sound Accelerator card if there is one present.

The benefit of this technique is that your application only has to call SndNewChannel for snth ID 5, regardless of whether you want to use the Sound Accelerator or not. Only the Init longword in the SndNewChannel command needs to change.

Pascal 'snth' Data Structures

```
CONST
   ChunkyBufferCmd = -81;
   ChunkyContBufferCmd = -83
   InterleavedBufferCmd = -10001;
   InterleavedContBufferCmd = -10002;

   ChunkySoundCmd = -80;
   InterleavedSoundCmd = -10003;

TYPE
   NewSoundBuffer = RECORD
      SoundPtr: Ptr;          {Pointer to the sound data. NIL if it
                                 follows this record}
      Length: LongInt;        {the number of SAMPLES of sound. X2 if stereo.}
      SampleRate: LongInt;    {Unsigned Fixed type: 0 to 65535.9999 Hz}
      LoopStart: LongInt;     {same as for BufferCmd}
      LoopEnd: LongInt;       {same as for BufferCmd}
      Flags: SignedByte;      {Bit 7: 1 = 16 bit samples, 0 = 8-bit samples}
                              {Bit 6: 1 = Stereo sound,   0 = monophonic sound}
                              {Bits5,4: these form a 2-bit
                                 encoded field as follows:}
                              { 11: Reserved - do not use}
                              { 10: Interleaved Buffer follows}
                              { 01: Chunky Buffer follows}
                              { 00: Normal Buffer follows}
                              {Bit 3: Reserved.
                                 Set to 0 for future compatibility}
                              {Bit 2: Reserved.
                                 Set to 0 for future compatibility}
                              {Bit 1: Reserved.
                                 Set to 0 for future compatibility}
                              {Bit 0: Reserved.
                                 Set to 0 for future compatibility}
      BaseNote: SignedByte;   {MIDI note number to play this sound at}
   END;

   NewSoundBufferPtr = ^ NewSoundBuffer;
```

Format of a "Chunky" Sound File Buffer

Chunky sound buffers are stored as Offset Binary data, where $0000 is the lowest value and $FFFF is the highest value. This is the native data format for the DAC on the Macintosh. Chunky buffers are encoded in Offset Binary so that they are backwards-compatible with the original Apple snth ID 5. There are 4 different versions of a Chunky soundfile buffer, one for each possible combination of 8/16-bit and mono/stereo as follows:

8-bit mono:
Same as current sound resources. A single block of consecutive 8-bit offset binary samples.

8-bit stereo:
A block of all the Left Channel samples followed by a block of all the Right Channel samples.

16-bit mono:
A block of all the most significant bytes of each sample, followed by a block of the least significant bytes of each channel.

16-bit stereo:
4 consecutive blocks of data as follows:

The most significant bytes of the Left Channel
The least significant bytes of the Left Channel
The most significant bytes of the Right Channel
The least significant bytes of the Right Channel

Format of an "Interleaved" Sound File Buffer

Interleaved sound file buffers store their data in Two's Complement encoding, as opposed to Offset Binary for Chunky and Normal buffers. Because of the encoding and interleaving, an interleaved buffer is not backwards-compatible with the original Apple snth ID 5, and will not play through it.

An interleaved buffer is stored in the same manner as an AIFF sound file. That is, the samples are stored consecutively as sample frames, where a frame is the sample data for all sound channels at that instant. For example, a stereo sound file stored in this format would look like this:

Left Channel sample #1
Right Channel sample #1
Left Channel sample #2
Right Channel sample #2
Left Channel sample #3
Right Channel sample #3
etc.

If there are 8 bits per sample, then each line above would be one byte of data. If there are 16 bits per sample, then each line above would be 2 bytes of data, most significant byte first.
In a stereo soundfile, the Left Channel data always comes first, as shown above.

snth Command Summary

cmd = ChunkyBufferCmd
param1 = NIL
param2 = NewSoundBufferPtr
Sent by an application, ChunkyBufferCmd plays the buffer pointed to by param2 at the given sample rate and at the last set amplitude. The sample rate conversion phase values are reset to zero at the start of this buffer. The buffer format ("Chunky Format") is shown below. The channel pauses until the sound has played. If a Sound Accelerator is not available, this command is translated into a BufferCmd, and the 8 most significant bits of the left channel are played by the standard Mac snth resource on the Mac sound hardware. Otherwise, the Sound Accelerator can play 8 or 16 bit samples, in mono or in stereo.

✔ LoopStart, LoopEnd and BaseNote are ignored.

cmd = ChunkyContBufferCmd
param1 = NIL

param2 = NewSoundBufferPtr
Same as ChunkyBufferCmd, except that the sample rate conversion phase values are not affected at all.

cmd = InterleavedBufferCmd
param1 = NIL
param2 = NewSoundBufferPtr
Same as ChunkyBufferCmd, except that the data format of the sound samples themselves corresponds to the Interleaved Buffer format as described above. Also, if a Sound Accelerator is not available, this command is ignored and no sound is played, since the standard sampled sound snth resource cannot play buffers of this format.

cmd = InterleavedContBufferCmd
param1 = NIL
param2 = NewSoundBufferPtr
Same as InterleavedBufferCmd, except that the sample rate conversion phase values are not affected at all.

cmd = ChunkySoundCmd
param1 = NIL
param2 = NewSoundBufferPtr
Sent by an application, ChunkySoundCmd specifies the sound to be played by succeeding note and frequency commands. Param2 contains a pointer to the NewSoundBuffer format sound header to be played. The sample rate conversion phase values are reset to zero at the start of this buffer. The buffer format is the "Chunky Format". If a Sound Accelerator is not available, this command is translated into a SoundCmd, and the 8 most significant bits of the left channel are played by the standard Mac snth resource on the Mac sound hardware. Otherwise, the Sound Accelerator can play 8 or 16 bit samples, in mono or in stereo.

- ✔ LoopStart and LoopEnd are ignored.

cmd = InterleavedSoundCmd
param1 = NIL
param2 = NewSoundBufferPtr
Same as ChunkySoundCmd, except that the sound data is stored in the "Interleaved Format". Also, if a Sound Accelerator is not available, this command is ignored and no sound is played, since the standard sampled sound snth resource cannot play buffers of this format.

Sound Installer Application

In order for the above commands to work, the Sound Accelerator snth resource must be installed. If the sound to be played is done so by the system (SysBeep, etc.), then it is sufficient to install the snth resource only in the System file. However, if the sound is to be played by the Application (via calls to the Sound Manager), then the snth resource should be installed in the application itself as well, but only if the application has its own snth ID 5 resource to begin with.

Examples of such programs are Hypercard and most Digidesign software. If you need to only rely on the standard snth ID 5 resource that is in the Standard System file, then you do not need to install the Sound Accelerator snth into your application. Installation is accomplished by using the Sound Installer application that comes with the developer materials.

The Sound Accelerator snth works by renaming the existing snth ID 5 resource, and installing itself as snth ID 5. When your program makes a call to the Sound Manager, it should specify snth ID 5 as

the snth to play on your sound channel. The Sound Accelerator snth will attempt to allocate a Sound Accelerator card when it needs to play a sound. If it cannot get one for any reason, it then invokes the renamed original snth ID 5, and passes the translated command on to it. In this way, you will hear the sound in the best possible manner that the current state of the hardware allows. If one piece of sound hardware is busy, you will hear it on the next best thing.

Because both the snth and the Device Driver are working together to efficiently manage the existing sound hardware resources (the internal Apple sound hardware and any Sound Accelerator cards installed), it is up to the application to be careful in its use of the Sound Manager. The best practice is to allocate a Sound Channel from the Sound Manager only at the time you need to make a sound, and to dispose of the Sound Channel immediately after. In this way, the Sound Accelerator that played your sound can be released for use by other applications and the System.

If you absolutely must keep a Sound Channel open for the duration of your application, then you should use the ReleaseCmd and GainCmd to release and regain control of the sound hardware when you do not need it, as Hypercard does.

Debugging Applications

Debugging Sound Accelerator applications can be particularly difficult because much of the I/O is done during interrupts and there is no easy way to stop the 56k and see what it is doing. The following are a few suggested approaches to this problem.

Debug your 56k algorithm using the simulator first, or make sure your algorithm (particularly complicated digital signal processing algorithms) works in non-real time using a high level language - before you burn up huge amounts of time getting it to run in real time. There are basically two parts to Sound Accelerator algorithms: the I/O (transferring samples and parameters to and from the card), and the DSP algorithm itself. You will save a lot of time if you are confident that one of these two processes is correct.

The slot interrupt handlers in DSPWorkshop provide a method for the 56k to notify the Mac about its status and current position. The 56k always sends an opcode to generate a slot interrupt which the Mac can interpret as a request to transfer data or an error/debugging message. When you are developing an algorithm, it is useful to activate the reporting of the messages (remove the semi-colons in USampler.a) and use this method to single step the 56k.

If you get persistent SSI underrun exceptions (ie, not just when you turn the SSI on or off), then your SSI interrupt routine is probably too long. This symptom is a good way to find out just how much you can do in real time. If you are developing an algorithm that can be scaled (such as the number of taps in a filter algorithm), then you can find out just how much you can do by increasing the algorithm until you get persistent SSI exceptions. This is useful because the I/O to and from the Mac is often hard to calculate as discussed in the section Speed Considerations. If you use this, be sure that you scale the algorithm depending on the machine you are using. A Mac II may run slower then a IIcx and an SE may run faster or slower then either of these.

A common 56k code error is to execute a DO or REP loop with a count of zero. This is interpreted not as zero iterations, but as 65536 iterations (see page 1-123 of the gray book) which will be evident when your algorithm seems to run incredibly slow or just dies altogether. Also note that REP loops are not interruptable. If you need to run a long one instruction loop in the background while an interrupt can occur (ie an SSI Tx interrupt), use a DO loop so that the interrupt can occurr.

The host flags HF0-3 can be very useful when trying to coordinate the 56k and the Mac. Note that the host flags are in HCR and HSR. It is very easy but terribly wrong to try to set a flag in the HSR register, as it is a read-only register for the 68000. Make sure that if you are setting a flag, it can be set from the processor you are setting it from. See the gray book page 7-13 and 7-15 for more details.

Note that the host port is double buffered. A common error in transfering data between the Mac and the 56k occurs if you control the transfer using the host flags. The problem occurs when one end signals the other to stop sending/receiving by setting a flag before the current data is read in. This will result in a piece of data remaining in the host port causing the next routine that transfers data to read garbage. You can avoid this by either making sure both ends know how many pieces of data will be transferred, or by reading from the host port until the TXDE/TRDY (see page 7-19 in the gray book) flags are clear.

The developer source code disk includes a primitive 56k debugger which allows you to read the value of data locations in the Sound Accelerator's data memories from the MPW shell. This can be quite useful if you are tracking the value of a certain variable stored in the 56k's memory, or for doing post mortems on an algorithm (providing the 56k side of the debugger isn't dead also). If you are developing on an a card with DMA abilities (such as the Audiomedia), we suggest you store

variables in off chip RAM (ie DMA RAM) so that you can read it from Macsbug. When you have debugged your algorithm, you can move the variables into on chip RAM for faster access.

Speed Considerations

The 56000 on the Sound Accelerator is an extremely fast processor. Its instruction cycle time is 100 ns, and with the fast static RAM on board, it always runs at zero wait states. The 56000's host port and SSI port both operate simultaneously and independently of the main DSP, and thus do not slow it down.

Because the 56000 host port is only 8 bits wide, and its native word width is 24 bits, it is necessary to transfer a word of data to it with 3 byte-wide accesses. In addition, you must read a status bit in one of the host port registers to determine if it is okay to send or receive data to/from the card before you access the data register in the host port. All summed up, a 24-bit data transfer to/from the card really requires up to 4 byte-wide, sequential Nubus accesses.

This number can be whittled down somewhat by clever programming, however. The check of the status bit can be eliminated if your 56000 code is fast enough to read/write data at the same rate as the Mac's processor. Considering that the 56000 is many times faster than the 68020, this situation is not difficult to achieve.

Also, since most digital audio data that is transferred to or from the card is really 8 or 16-bit data, and not 24-bit data, you only need to transfer the 8 or 16 least significant bits of the 24-bit word. The 24-bit data register in the host port is divided up into three 8-bit registers: high, mid and low. When the low byte register is written or read, the 56000 considers the entire 24-bit register to have been accessed. Therefore, when reading or writing to the host port data register, the low 8-bit register should be the last one of the three that is read or written. So, for an 8-bit data transfer, you access only the low register. For a 16-bit data transfer, you access the mid register, and then the low register. For a 24-bit data transfer, you access the high register, then the mid register, and finally the low register.

Since we end up placing the data in the least significant bits of the host port's data register, it is necessary to shift them up into the most significant bits of the 24-bit word once they have been read into the 56000. This is usually accomplished by having the 56000 execute a MPY (multiply) instruction, which accomplishes the desired shift. See the example 56000 code for specific implementation details of this technique. The important point here is that we are trading one or two 1.25 µsec Nubus accesses for a single 100 ns 56000 instruction cycle. This results in a significant increase in throughput to and from the card.

On the Mac SE version of the Sound Accelerator, the card is attached directly to the 68000's address and data bus lines. Although the 68000 is slower than the 68020 in the Mac II, the direct connection to the 68000 makes up for the decrease in processor speed. The overall throughput of the two versions of the card are therefore about the same, about 1.25 µsec per card access.

A simple calculation shows that at a digital audio rate of 44.1 kHz, a single channel of 16-bit data has a sample period of 22.6 µsec. If we assume that a 16-bit data transfer to the card requires one Nubus read for a status check, and then 2 Nubus reads or writes to transfer the 2 bytes of data per 16-bit word, then we have an overhead of 3 * 1.25 = 3.75 µsec per sample. At this rate, we should be able to run 22.6 / 3.75 = 6.02 channels of 16-bit digital audio to a card simultaneously. This number goes up as sample rate or word length goes down, or if the status check can be eliminated by careful programming.

Note that the above discussion is referring only to the transfer rate of data in and out of the card. While this data transfer is going on, the 56000 on the card is capable of simultaneously processing the data that is going through it. At 44.1 kHz mono, the 56000 can execute 223 instructions per

sample! And this is all happening in real-time. The overall processing power of this system is several orders of magnitude greater than what can be done with the Macintosh alone.

If you are coding for the Audiomedia card using its DMA transfer abilities, the total bandwidth is considerably higher. This increase comes from a number of sources. First, you are able to transfer 24 bit words into DMA memory during only one NuBus transaction. Using the calculation above, 1 NuBus read per 1.5 16-bit samples (ie if you pack consecutive samples into 24 bit words) yields .83 µsec per sample. At this rate, we should be able to run 22.6 / .83 = 27.2 channels of 16-bit digital audio to a card simultaneously. How close you come to this "ideal" rate involves a number of factors. If the 56000 is continuously accessing its external memory (ie the DMA memory) the 56000 and the NuBuss will have to arbitrate for the memory which effectively increases their access times. Synchronizing NuBuss and 56000 accesses to the memory, and moving commonly accessed code and data on the 56000 into on chip RAM (which is not accessible to the NuBus) can help reduce arbitration. In addition, if you are doing disk I/O, a large part of the 68000's processing can be used up handling the disk transfers. We have found that using real world algorithms and applications, you can achieve 8-10 channels of audio if you are using RAM based storage on the Mac, and 4-6 channels of audio if you are reading from a fast hard disk. As before, if you use smaller then 16 bit word widths or lower then 44.1kHz sampling rates, these numbers will go up.

Miscellaneous Notes, Suggestions, and Warnings

The surgeon general has determined that 99.9% of 56k algorithm bugs are caused by the incorrect use or interpretation of fixed point arithmetic. Make sure you understand the consequences of page 4-15 in the gray book and how this affects moving data from location to location (ie, from A into R0, as opposed to A1 into R0). This is best learned by experimenting in the simulator. Also, make sure you understand how SASample shifts the sample data by fixed point multiplying in the routines FillBufLo and FillBufHi. This is a powerful technique that can reduce the time spent transfering data to and from the 56k as well as turning multiple shift operations in an algorithm into one single-cycle multiply.

MIDI Manager users: Beware of interrupt level conflicts between MIDI manager routines which operate at a HIGHER level than slot interrupts. This presents a problem for applications which need to play audio (a time-critical process) in response to MIDI data. The solution depends on your application, but generally you should keep the MIDI manager routines as short as possible, so that pending slot interrupts are not postponed too long.

If you are working with multiple cards or multiple applications using the Sound Accelerator, SATest can be useful running in MultiFinder to allocate a card and hold it. This is available using the menu option 'Allocate a card' which turns into 'Release card in slot xx' after being selected. It just grabs a card from the driver so that you can see if your program can gracefully deal with being denied a card. It also has an option to reset all the cards and clear out the driver (ie, de-allocate all cards). This is useful after you exit a program (using ES in Macsbug) that has crashed, while still holding onto a card. It is also useful when changing DSP code, so that the driver will load the new code and not just use the old code already loaded in the card. The object TSACard in USACard.asm clears the driver and resets all card automatically each time you run the application, if you compile with the debug option in MacApp.

To save space on the distribution disk, SATest only has a 'snd ' resource with id 10. To use the menu options to play other 'snd ' resources, you must install your own in the application with id numbers corresponding to the ones appearing in the File menu.

Note that after system reset the Sound Accelerator driver always allocates cards starting from the highest address and decreasing. Using this fact and the new driver control call which requests a particular card, you can be sure that if there are multiple cards in a system, the one you receive is the one with an Ad In connected.

Finally, please note that the DSP clock frequency of the 56000 is subject to change. Do NOT assume it to have a certain value. Use the ctlGetClockRate call of the device driver to find the DSP clock rate for a particular card. This is especially important if you are using the on-chip clock generator of the 56000, instead of the off-chip, on-card clock. The on-chip clock is divided down from the DSP's clock, so any change in the DSP clock will affect the on-chip clock generator. The frequency of the off-chip, on-card clock is guaranteed to always be 44.1 kHz (mono or stereo). As such, we recommend that you use the off-chip, on-card clock.

If you program the DSP to allow the use of its internal memories, you can access the 'overlapped' external memory by wrapping around at the high end of memory. For example, if you have 2048 words of external program memory, locations $0000 - $01FF will refer to the DSP's internal memory, locations $0200-$07FF will refer to external program memory, and locations $0800-$09FF will refer to the lowest $0200 locations in external program memory. You can treat program and data memories as a contiguous space, with the external memory immediately following the internal memory. For applications that require extremely large buffers, you can place these buffers

at $1000 to $2000 (ie 4k buffers) so that you satisfy the modulus register constraints on a 56000, and still use the low 256 words of internal memory for local fast processing. The buffers will in affect start at the base of external memory because of the address wraparound. Your memory map will then have a 'hole' in it from $0100 to $1000, but you will be able to utilize the entire 4k + 256 of memory completely.

- ✔ It is not a good idea to use this technique with program RAM, if you have 32K words of program RAM installed, as the overlap address range will run into the control latch on Rev. B cards!

Questions And Answers

Q: Does Digidesign plan to incorporate a software interface to Sound Designer II so that developers can develop signal processing 'XCMD's?
A: At this point we have no plans to add a software interface to Sound Designer. We researched the idea of an XCMD/XFCN interface but found that signal processing applications developers wanted to add were in general too extensive to live in a simple XCMD interface, and that most would resort to writing their own application in the end.

Q: What kind of filters do you use in Sound Designer II?
A: The parametric EQ and graphic EQ modules use variable IIR (infinite impulse response) filters. The sample rate conversion module and tapedeck module use FIR (finite impulse response) filters.

Q: How do you implement hard disk recording and playback?
A: It's really not so complicated. You basically set up a double buffer on the card (as in DSPWorkshop SamplerPlay.asm) and another larger double buffer (about 32k depending on disk access time) in the Mac's main memory. The main program keeps reading or writing the Mac's buffers to and from disk while the slot interrupt jumps in and reads/writes samples (depending on whether you are recording, playing, or doing both). The general form of the code is as follows:

1) Double buffers are used throughout. On the card their are two double buffers (one for each channel) that are approximately 1k samples in size. In the Mac's main memory there are also two double buffers that hold samples that are being read from or about to be written to the disk. These are about 32k depending on the access time of the hard disk in question.

2) Playback is achieved as follows: Fill the low buffer in the Mac's memory and start the card playing. The card's algorithm generates a slot interrupt and asks for a card buffer of samples from the Mac, which the Mac proceeds to send (ie 1k samples). This is the same as DSPWorkshop except that when the slot interrupt routine reaches the end of the Mac's RAM buffer, it wraps around to the start of the buffer (like the buffer in the card ie modulo addressing). Meanwhile, the program running below the slot interrupt level watches the RAM buffer pointer that the slot interrupt routine uses and waits until it crosses a buffer boundary. When this occurs, it initiates a disk read to fill the buffer half that the slot interrupt has just left. This continues until the end of the disk file is reached. If the slot interrupt pointer has not crossed a boundary, the main program can update screen graphics, or decided which file to play next.

3) Recording is similiar to playback, except that the card sends a buffer of samples to the Mac's memory during each slot interrupt and the main program writes out a buffer to disk after the slot interrupt pointer crosses over a buffer boundary.

4) All disk reads and writes can be done using standard PB calls (see Inside Mac Vol IV). This is primarily because FSWrite does a read verify therefore doubling the access time when recording.

Several developers have managed to write their own hard disk code given the above information and the DSPWorkshop source code. Typically the 56000 and slot interrupt code can be taken directly from DSPWorkshop (with additions if you want to support crossfades ie varying an amplitude scaler in time). The only thing you have to add is the code to manage and fill the disk buffers. Typically, the choice of which file to read and when is rather application specific hence developers have to code it themselves (ie our internal code would not necessarily apply).

Q: What's James Brown's address in jail?
A: See Apple's January 1990 Develop Journal.

MPW Shell Tool Parse56k

Syntax
Parse56k [-r rezFile rezType rezID] [-s] [-p] [file]

Description
Parse56k is an MPW shell tool (as well as standalone application) that reads the output of the Motorola 56000 linker and converts it into a number of useful forms for Sound Accelerator program development. The default option is DC.L directives for embedding 56000 code into Mac assembler code files. The source code to Parse56k is included on the developer documentation disk. It is essentially a simple text file parser. If you work under another development environment (ie Think C) you may want to create your own version.

- ✔ Because the Sound Accelerator driver only loads P memory, Parse56k IGNORES X AND Y DATA INITILIZATION. This is most often used in FFT algorithms to load a sine table into data memory. The correct way to load tables is by using a host interrupt as demonstrated in the file UFilter.asm (see DSPWorkshop code). It is useful to use the directives org X:.. and org Y:... to initialize tables before using the 56000 simulator in which case you just don't call the corresponding host commands when simulating. Because the simulator works using the linker's output file, it will get the X/Y data directives even though Parse56k ignores them..

- ✔ Parse56k requires that the linker file be in 'order'. This means that in your .asm files you must make sure that ORG P:xxxx statement are in order of INCREASING xxxx or address.

Input
Motorola 56000 linker output.

Output
Mac assembler directives or modification of an existing resource file/application.

Status
0 - Worked.
1 - Had problems usually with opening the resource file.

Options
-s – Just output the size in the number of 24 bit words that the code would occupy in the 56000's program memory space.

-p – Output the assembler directives in packed format (ie: DC.B followed by DC.W). This avoids wasting the high byte in the DC.L default output format.

-r rezFile rezType rezID – Modify or create a data resource in rezFile of type rezType and ID number rezID. This is useful in a make file for automatically inserting your code into a resource so that at run time the program can just get a handle to the code and pass this to the driver to load it.

Examples
Parse56k foo.lod
Outputs to standard output something like the following:

```
DC.L    $0C0040  ;PC=0
DC.L    $081234  ;PC=1
etc...
```

Parse56k -p foo.lod
Outputs to standard output something like the following:


```
DC.B      $0C       ;high byte PC=0
DC.W      $0040     ;low word PC=0
DC.B      $08       ;high byte PC=1
DC.W      $1234     ;low word PC=1
```

See Also

SASample and DSPWorkshop makefiles both show how to use Parse56k to automate your build process. The source code to Parse56k is included with the Developer Documentation.

SASample and DSPWorkshop

SASample and DSPWorkshop are sample programs provided in source form which are intended as examples of how to write programs that use the Sound Accelerator and Ad In. They are not meant to be the most optimal or the basis for a full blown Macintosh audio application.

SASample is written in MPW Pascal and Assembler. It is an extremely simple example with as little Macintosh interface included as possible. It load Sound Designer I format files into memory an plays them out of the card.

DSP Workshop is a more complicated program written in MacApp version 2.0 (Apple's object-oriented Macintosh shell) that graphically displays buffers, reads and writes Sound Designer II format files, and allows you to record, play, and low pass filter buffers of sound stored in RAM. In addition it contains a very simple spectrum analyzer that displays the magnitude of a 256 point FFT as computed by the card on incoming audio data. It uses an object called TSACard (which lives in the file USACard.p and .a) that hides many of the hardware details of interacting with the Sound Accelerator.

If you are not familiar with object oriented programming or MacApp, DSP Workshop is still a useful example. In particular, the file USACard.a includes a variety of routines which transfer data to and from the Sound Accelerator as well as test and set certain status bits. In addition, USACard.a includes examples of tranferring data using the Audiomedia's DMA I/O. The file UFilter.asm demonstrates how to use the card purely as a compute engine. USamplerRecord and USamplerPlay demonstrate how to record (using the Ad In) and play stereo or mono sound data. Note that both of these operate only at 44.1 kHz. If you want to play or record at other rates you will have to develop your own sample rate conversion routines.

If you recompile DSPWorkshop, it is interesting to note what proportion of time is spent designing the filter (which uses SANE) and what proportion of time is spent actually filtering the data. Almost all of the time is spent designing the filter. In fact, because you have about 220 instructions per output sample, the algorithm in UFilter.asm can run in real time depending on the number of taps you select. A good first exercise in writing Sound Accelerator code would be to merge USamplerPlay.asm and UFilter.asm so that the sound is filtered in real time as it is played. This involves adding a host command to USamplerPlay.asm to load the desired filter table as well as modifying the SSI interrupt routine to perform the same operations as the routine CrunchBuffer in UFilter.asm before playing a sample.

Sound Designer I File Format

Sound Designer I file format is the original Digidesign sound file format first released in 1985. It is widely used and supported as evidenced by the many CD ROM discs with sound effects stored in this format. It is primarily used to store mono 16 bit short duration (on the order of seconds) audio samples. We recommend that you support this format for short sounds, but that you use Sound Designer II format as a primary format due to its flexibility.

File Type: 'SFIL'

Resource Fork

The resource fork is not used.

Data Fork

The first 1336 bytes of the data fork contain the sound header (see the Pascal HeaderType record below) followed by the sample data itself. The most important fields are in bold. Default values are given in ( ) within the comment next to a field. Byte offsets are indicated to the left at various offsets. Fields with the comment DO NOT USE are for Sound Designer internal use and should not be changed except when initializing the file. The size of the data types are detailed in Inside Macintosh, Volume 1, page 86.

```
MarkerType = RECORD
   Free: Boolean;                {TRUE if this marker is free for use (TRUE/1)}
   Position: LongInt;            {byte position in file (0)}
   Name: STRING[32];             {name of the marker ("Untitled")}
END;

SideType = (leftSide,rightSide);

EditRecord = RECORD
   HiAddr: LongInt;              {DO NOT USE (0)}
   LoAddr: LongInt;              {DO NOT USE (0)}
   ExtendSide: SideType;         {DO NOT USE (0)}
END;

ScaleNames = (Time,SampleNumber,HexSampNum,Volts,Percent,dbm,User);

ModeType = (Select,Draw,ZoomSelect);

ZoomType = RECORD
   v: Integer;                   {vertical scale factor:
                                    positive = magnification
                                    negative = reduction (-256)}
   h: LongInt;                   {horizontal scale factor:
                                    positive = magnification
                                    negative = reduction (1)}
END;

ScaleType = RECORD
   VFactor: LongInt;             {scale factor for vertical axis tick units:
                                    positive = magnification
                                    negative = reduction (327)}
   VType: ScaleNames;            {type of vertical axis tick mark unit (4)}
   VString: STRING[32];          {vertial axis tick units string ("%Scale")}
   HFactor: LongInt;             {scale factor for horizontal axis tick units:
```

```
                                positive = magnification
                                negative = reduction (1)}
   HType: ScaleNames;           {type of horizontal axis tick mark unit (0)}
   HString: STRING[32];         {horizontal axis tick units string:
                                µsec,msec,sec,samples ("sec")}
END;

HeaderType = RECORD
   HeaderSize:  Integer;        {size in bytes of the file header (1336)}

   Version: Integer;            {DO NOT USE (32)}
   Preview: Boolean;            {DO NOT USE (0)}
   WPtr: WindowPtr;             {DO NOT USE (0)}
   WPeek: WindowPeek;           {DO NOT USE (0)}

   HInxPage,HInxLine: LongInt;  {DO NOT USE (0)}
   VInxPage,VInxLine: LongInt;  {DO NOT USE (0)}
   HCtlPage,HCtlLine: LongInt;  {DO NOT USE (0)}
   VCtlPage,VCtlLine: LongInt;  {DO NOT USE (0)}

   VOffset: LongInt;            {position of vertical center of window in
                                quantization units ie -32768 to 32767 (0)}
   HOffset: LongInt;            {position of left edge of sample window
                                in #SAMPLES (0)}
   VOffConst: Integer;          {DO NOT USE (0)}

   Zoom: ZoomType;              {see above}
   Scale: ScaleType;            {see above}

   VScrUpdate: Integer;         {DO NOT USE (0)}
   BufPtr: Ptr;                 {DO NOT USE (0)}
   BufBytes: Size;              {DO NOT USE (0)}
   BufOffset: LongInt;          {DO NOT USE (0)}
   WaveRgn: rgnHandle;          {DO NOT USE (0)}
   ClipArea: rgnHandle;         {DO NOT USE (0)}
   ScaleArea: rgnHandle;        {DO NOT USE (0)}
   CtlWidth: Rect;              {DO NOT USE (0)}
   VScroll: ControlHandle;      {DO NOT USE (0)}
   HScroll: ControlHandle;      {DO NOT USE (0)}

   FileSize:  Size;             {2 * number of samples in this file
                                ie number of bytes of sound data}

   BUName: STRING[64];          {name of edit backup file}
   FileName:  STRING[64];       {name of this file (Mac Filename)}

   BURefNum: Integer;           {DO NOT USE (0)}
   refNum: Integer;             {DO NOT USE (0)}
   vRefNum: Integer;            {DO NOT USE (0)}
   BufChanged: Boolean;         {DO NOT USE (0)}
   FileChanged: Boolean;        {DO NOT USE (0)}
   NoBackup: Boolean;           {DO NOT USE (0)}
   Mode: ModeType;              {DO NOT USE (0)}

   Edit: EditRecord;            {see above}

   CursorPos: LongInt;          {cursor position relative to window start (0)}
   CursorRgn: RgnHandle;        {cursor region in which it can be grasped (0)}

   MarkerData: ARRAY[0..9] OF MarkerType;    {see above}

   MarkerOffset: LongInt;       {offset to get relative time (0)}
```

```
    LoopStart: LongInt;          {starting byte # of loop (-1)}
    LoopEnd: LongInt;            {ending byte # of loop (-1)}
    ZeroLineOn: Boolean;         {(FALSE/0)}
    CursorOn: Boolean;           {(FALSE/0)}
    ScalesOn: Boolean;           {(FALSE/0)}

    Comment: Str255;             {file comment (" ")}

    SampRate:  LongInt;          {sample rate in hertz ie 44100}
    SampPeriod:  LongInt;        {sample period in microseconds}
    SampSize:  Integer;          {number of bits in a sample (16)}

    CodeType: STRING[32];        {type of sample data ("Linear")}

    UserStr1: Str255;            {for user comments or reserved for future}
    BufSize: Size;               {size of the RAM wave buffer in bytes}

    Loop2Start: LongInt;         {release loop start in bytes (-1)}
    Loop2End: LongInt;           {release loop end in bytes (-1)}
    Loop1Type: signedByte;       {type of loop: 1 = forward 2 = forward/backward}
    Loop2Type: signedByte;       {type of loop: 1 = forward 2 = forward/backward}
    User4: Integer;              {DO NOT USE (0)}
END;
```

Header Notes

Integers (2 bytes) and LongInts (4 bytes) are stored in Motorola 68000 format with the most significant bytes stored first, followed by the least significant bytes. For example, to store the hexadecimal value 0123 4567 as a LongInt, we would store in ASCENDING memory locations: 01, 23, 45, 67. Leading zeros must be considered as part of the number.

The total length of the header is 1336 bytes. It is the first thing in the file, so if a file is rewound to an offset of 0 from the beginning of the file, the file position marker will be pointing to the first byte in the header.

If a default value is given for a variable in the header, it MUST be set to that value when creating the header for the first time. All default values for numeric data in the header are given in DECIMAL. A header variable with the comment "DO NOT USE" should be set to its default value when creating a file, and under no circumstances used otherwise.

The notation for strings in the Macintosh is STR[NN] where NN is the length in bytes of the string. The actual string stored in memory or the header will have one extra byte preceding it. This byte contains the length of the string, NN. For example to store a STR[32] in the header, the first byte will be 32 (decimal), followed by 32 bytes of ASCII representing the string. A string constant is specified above by using quote marks ie (" "). DO NOT include the quote marks as part of the string itself. Note that the string values in the header give the MAXIMUM length of that string variable. If you wish to use a string which is shorter (and you usually will), then the first byte of that string should give the actual number of ASCII characters in the string you wish to use. For example, although FileName is specified as a STR[64], you may wish to use less characters for a filename. If you wish to use "New File" as the filename, you would store the byte 08 (ie the length not including the length byte itself) followed by the ASCII values for the characters in the string "New File". The byte 08 says that there are 8 bytes in the string itself.

End addresses, such as Loop End addresses are the address of the first byte AFTER the last byte IN THE LOOP. They DO NOT refer to the last byte of the loop. Start addresses, however, refer to the first byte of a selection.

There are currently no user bytes available in the header for independent use. This does not preclude the possibility that there will be some date, however at this time you must assume that no bytes in the header may be used for any purpose other than described above. All bytes which are NOT listed above are reserved for use exclusively by Digidesign Inc. until further notice.

Sound Designer II File Format

Sound Designer II files store all sound samples in the data fork and all sound parameters in the resource fork. This is extremely conveniant for sound data where the data fork may grow to a hundred megabytes or more. Regardless of the size of the data fork you can add, delete, and modify sound parameters at will without compacting the sound data or moving it around the disk (and extremely time consuming procedure if the file is 100 MB). In addition, you may add your own parameters to a file (as long as their resource types don't conflict with Sound Designer II's) while allowing the file to be read by both Sound Designer and your program. We recommend that developers standardize on the Sound Designer II file format as the primary format due to its customizability. For multi-track operations, we recommend that each track be recorded in a separate single mono Sound Designer II file using stereo Sound Designer II files as the mastering medium.

File Type: 'Sd2f'

Resource Fork

The resource fork contains many different kinds of resources which specify everything from the sample rate to loop points to graphic eq settings. Most of these are application-specific to Sound Designer II. There are three core parameters/rsrcs in an SDII file:

Type: 'STR ' ID: 1000 Name: 'sample-size'
Value: Integer numeric string specifying the number of bytes per sample (ie 2 for standard 16 bit samples).

Type: 'STR ' ID: 1001 Name: 'sample-rate'
Value: Floating point numeric string specifying the sample rate in hertz of the file (ie '44100.0000' for a standard 44.1kHz sample rate file).

Type: 'STR ' ID: 1002 Name: 'channels'
Value: Integer numeric string specifying the number of channels in the file (ie 2 for stereo).

The above resources are all that are specifically required in an SDII file. Given only these three parameters, Sound Designer II can read in and play the file. If your program modifies the sound data or above 'STR ' resources in any way, it is your responsibility to delete any other resources that may have become out of sync with the file. This can happen if you delete a large section of the file without updating the playlist pointers. Notice that the length of the file is not stored. This is because you can derive the length given the sample size, number of channels, and the length in bytes of the data fork (using file system calls) using the following formulas:

Length of file (in sample frames)
= Length of data fork / (number of channels * sample size)

Length of file (in seconds)
= (Length of data fork / (number of channels * sample size)) / sample rate

Other Parameter/Resource Types

The following resources store file information used by Sound Designer II for various advanced playback and processing. Unlike the above core parameters, these are stored as Pascal RECORD structures written out to resources. Therefore they are defined by the associated Pascal RECORD

definitions below. Default values are given in ( ) within the comment next to a field. Byte offsets are indicated to the left at various offsets. Fields with the comment DO NOT USE are for Sound Designer internal use and should not be changed except when initializing the file. The size of the data types are detailed in Inside Macintosh, Volume 1, page 86.

Type: 'sdDD' ID: 1000
Use: Stores general document information such as comments.
Structure: A DocumentDataRecord as defined below.

```
EditRecord = RECORD
   HiAddr: LongInt;              {DO NOT USE (0)}
   LoAddr: LongInt;              {DO NOT USE (0)}
   ExtendSide: SideType;         {DO NOT USE (0)}
END;

ZoomType = RECORD
   v: Integer;                   {vertical scale factor:
                                    positive = magnification
                                    negative = reduction (-256)}
   h: LongInt;                   {horizontal scale factor:
                                    positive = magnification
                                    negative = reduction (1)}
END;

ScaleType = RECORD
   VFactor: LongInt;             {scale factor for vertical axis tick units:
                                    positive = magnification
                                    negative = reduction (327)}
   VType: ScaleNames;            {type of vertical axis tick mark unit (4)}
   VString: STRING[32];          {vertial axis tick units string ("%Scale")}
   HFactor: LongInt;             {scale factor for horizontal axis tick units:
                                    positive = magnification
                                    negative = reduction (1)}
   HType: ScaleNames;            {type of horizontal axis tick mark unit (0)}
   HString: STRING[32];          {horizontal axis tick units string:
                                    µsec,msec,sec,samples ("sec")}
END;

DocumentDataRecord = RECORD
   Version: Integer;             {version/format of this resource(1)}
   BufOffset: LongInt;           {DO NOT USE (0)}
   MarkerOffset: LongInt;        {offset to get relative time of markers}
   Comment: Str255;              {file comment (" ")}
   HDPlayBufMultiple: Integer;   {HD play buf multiple for this file (8)}
   SMPTEStartTime: LongInt;      {SMPTE start time of this file (0)}
   FramesPerSec: Integer;        {SMPTE frame rate of this file (30)}
   FilmSize: Integer;            {16mm or 35mm film (35)}
   StartBarNum: LongInt;         {starting Bar Number (0)}
   StartBeatNum: LongInt;        {starting Beat Number (0)}

   StartFrame: LongInt;          {the sample frame this new entry starts at}
   Tempo: Fixed;                 {tempo in beats/sec}
   TimeSignature: Fixed;         {HiWord = numerator, LoWord = denominator}

   CursorPos: LongInt;           {cursor position relative to window start (0)}
   CursorOn: Boolean;            {(FALSE/0)}
   Zoom: ZoomType;               {see above}
   Scale: ScaleType;             {see above}
   Edit: EditRecord;             {see above}
   VOffset: LongInt;             {position of vertical center of window in
```

```
                                    quantization units ie -32768 to 32767 (0)}
   HOffset: LongInt;                {position of left edge of sample window
                                    in #SAMPLES (0)}
   CtlWidth: Rect;                  {DO NOT USE (0)}
   ZoomIndex: Point;                {h and v indices in zoom increments}
   SelectMode: Integer;             {DO NOT USE (0)}
END;
```

Type: 'sdML' ID: 1000
Use: Stores a list of text and numeric markers.
Structure:

```
Version: Integer;                   {version/format of this resource(1)}
MarkerOffset: LongInt;              {DO NOT USE (0)}
NumMarkers: Integer;                {number of MarkerRecords to follow}
```

Followed by 'NumMarkers' of the following records:

```
MarkerRecord = RECORD
   MarkerType: Integer;             {1 = numbered marker, 2 = text marker (1)}
   MarkerType: Integer;             {DUPLICATE of previous value
                                          - for historical reasons }
   Position: LongInt;               {sample frame in file}
   Text: LongInt;                   {DO NOT USE (0)}
   CursorID: Integer;               {24430 = numeric, 3012 = text (24430)}
   MarkerID: Integer;               {unique unsigned ID for each marker}
   TextLength: LongInt;             {length of marker text}

   >> The Text (ie TextLength bytes of ASCII characters) <<
END;
```

Type: 'sdLL' ID: 1000
Use: Stores a list of audio loops (commonly used with samplers).
Structure:

```
Version: Integer;                   {version/format of this resource(1)}
HScale:Integer;                     {DO NOT USE (0)}
VScale:Integer;                     {DO NOT USE (0)}
NumLoops: Integer;                  {number of LoopRecords to follow}
```

Followed by 'NumLoops' of the following records:

```
LoopRecord = RECORD
   LoopStart: LongInt;              {reference to start sample frame of this loop}
   LoopEnd: LongInt;                {reference to end sample frame of this loop}
   LoopIndex: Integer;              {identifies which loop this is (1..NumLoops)}
   LoopSense:Integer;               {117=foward loop, 118=backwards/forwards loop}
   Channel:Integer;                 {channel which loop is on (0..NumChannels-1)}
END;
```

Reserved Parameter/Resource Types

Digidesign reserves all resource types that begin with the letters 'sd' or 'dd' (capital and lower case) ie 'sdPL', 'SDxx', 'DDxx' or 'dDxx', for present and future use by Sound Designer II and other Digidesign programs.

Data Fork

Byte one of the data fork is the first byte of sound data. The sound data is organized as interleaved samples (if more then one channel) of either 8 or 16 bit samples depending on the value of the 'sample-size' STR resource (see below).

For example, a standard 16 bit stereo file would be organized as follows:

```
Left Channel sample #1
Right Channel sample #1
Left Channel sample #2
Right Channel sample #2
Left Channel sample #3
Right Channel sample #3
etc...
```

Where each sample is the MSB (most significant byte) followed by the LSB (least significant byte) or better known as little endian format.

A four channel file would be as follows:

```
Channel 1 sample #1
Channel 2 sample #1
Channel 3 sample #1
Channel 4 sample #1
Channel 1 sample #2
Channel 2 sample #2
Channel 3 sample #2
Channel 4 sample #2
etc...
```


AIFF File Format

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing for the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths. It also allows for the storage of various sound file parameters such as markers, instrument definitions, MIDI data, recording session information and comments. The CD-I (Compact Disc Interactive) industry has adopted it as a standard format for CD-I audio files. **IT IS PRIMARILY AN INTERCHANGE FORMAT**, although application designers may chose to use it as a primary file format. Sound Designer II can read, write, and edit AIFF format files, but it cannot record directly into them. This is because the actual sound data in an AIFF file can be located BEFORE the file parameters. A typical sound file can contain a sound data chunk of 10-100 MBytes followed by parameter chunks totaling at most 100kbytes. Recording into the file, and extending the sound data's length would require moving the sound data to the end of file, which given the Macintosh's file system, would be quite time consuming to say the least. We recommend that your application be able to read and write AIFF files, but not necessarily edit AIFF files.

We have included version 1.2 of the AIFF standard from Apple Computer, Inc. with the developer documentation for your convenience.

Macintosh Expansion Chassis

For developers wishing to use the Sound Accelerator with the Mac Plus, or use multiple cards with an SE or SE30, Second Wave provides a variety of external card cages. Second Wave also manufactures a NuBus expansion chassis, which contains slots 1-8 on the NuBus. Programs must access all cards in this chassis in 32-bit mode, since the Mac OS will not decode the chassis addresses in 24-bit mode. Version 4.2 of the Sound Accelerator configuration ROM as well as all Audio Media cards contain 32-bit clean drivers. Sound Designer II is NOT 32-bit clean at this point in time but will be in the near future. The example code is for the most part written to be 32-bit clean and should be used as a model of card access if you plan to address external expansion chassis, run under system 7.0, or A/UX.
For more information contact:

- Second Wave, Inc.
  9430 Research Blvd.
  Echelon II, Suite 260
  Austin, TX 78759
  (512) 343-9661

Control Signals and Magic Bits

The Sound Accelerator utilizes the 56000's I/O interface registers to control the DAC's, clock sources, and expansion port. If you don't plan to use any of these (ie you are using the card as a compute engine only) then see the 'magic bit' settings in the file UFilter.asm from DSPWorkshop. These basically turn the I/O and interrupts off so the 56k can just compute.

In general, it is best to just copy the appropriate lines from the sample programs. If you need to use non-standard settings, the following is provided and assumes that you have read chapter 11 in the 56000 users manual (the red book). A graphic picture of these bits can also be found in the appendices.

- ✔ The function's of some of the following bits have been replaced by the CTL_LATCH on Rev. B Sound Accelerators, the SE30, and the Audiomedia card. See the CTL_LATCH section for more details.

CRB - SSI Control Register B:

Bit 5: SCKD (Output):

1 Select on-chip baud rate generator to drive the SSI.

0 Select off-chip clock. This will be either the on-card crystal oscillator of the expansion port clock input depending on the PCD register.

Bit 1: OF1 (Output):

1 Stereo clock generation.

0 Mono clock generation.

- ✔ Sound Accelerator Rev. A and SE versions only. All later hardware uses the CTL_LATCH to select between mono and stereo.

Bit 0: OF0 (Output):

1 Select on-chip baud rate generator to drive the SSI.

0 Select off-chip clock. This will be either the on-card crystal oscillator of the expansion port clock input depending on the PCD register.

- ✔ **OF0 AND SCKD SHOULD ALWAYS BE SET TO THE SAME VALUE.**

PCD - Port C Control Register:

Bit 2: PC2 (Output):

1 Select off-card clock source. Off-card means that the clock comes from the expansion port on the end of the card. If you are using an off-card clock source, you must also have enabled the expansion port, by setting PC0 (see below).

0 Select on card clock source. This can be either the 56k's internal clock or the on card crystal oscillator depending on the setting of OF0/SCKD (see above).

✔ Sound Accelerator Rev. A and SE versions only. All later hardware uses the CTL_LATCH to select between mono and stereo.

Bit 1: PC1 (Input):

1 In stereo mode, indicates card expects the left sample to be output and that the input sample is from the right channel.

0 In stereo mode, indicates card expects the right sample to be output and that the input sample is from the left channel.

Bit 0: PC2 (Output):

1 Enable expansion port. In general it is a good idea to enable the expansion port so that external boxes (such as external DAC's or a digital I/O box) can get hold of the data you are generating.

0 Disable the expansion port. After board reset, the bit will be 0 (ie the expansion port will be disabled).

✔ Sound Accelerator Rev. A and SE versions only. All later hardware uses the CTL_LATCH to select between mono and stereo.

PCC and PCDDR:

The I/O pins in the PCD and CRB must be configured as general purpose I/O pins so the Sound Accelerator can use them for the above purposes. To use PC0, PC1 and PC2, their respective bits in the Port C Control Register (PCC) must be cleared to indicate that these pins are now used for general purpose I/O and not for the SCI port. In addition you must properly set up their respective bits in the Port C Data Direction Register (PCDDR). Generally, PCC and PCDDR will be set to the following values:

PCC: $01F8

PCDDR: $0005

See the 56000 register diagrams in the appendices and in the user manual for more details.

The CTL_LATCH

The CTL_LATCH is an I/O control register first introduced on the Sound Accelerator Rev. B NuBus card. It is used to control communication with external devices (such as the Ad In or DAT I/O) as well as replace some of the functionality of the SCI I/O pins as described in the last section. CTL_LATCH is set by writing the appropriate bit pattern to address $FFFF in P (program) memory. At boot time, the Sound Accelerator device driver sets the control latch to an initial value. Since this value is subject to change, you should not depend on it being in any particular state when your code is loaded. Make sure that you set it as one of the first things your program does, so that the card's hardware is set correctly to begin with.

- ✔ Currently, all cards EXCLUDING the Rev. A Sound Accelerator for the Mac II and the Rev. A Sound Accelerator for the SE have a CTL_LATCH.
- ✔ CTL_LATCH is a write-only register. If you change individual bits within the CTL_LATCH while your algorithm is running, you will want to store a copy of the settings before you write them so that next time you change a bit, you know the settings of all the other bits.

Note About 'Old' vs 'New' Mode Signals and Local Mode

You will notice that certain C-Bit settings for external devices such as the DAT I/O or the Ad In refer to old mode and new mode. Generally, old mode means that the external device generates bit clock and data while the Sound Accelerator generates the L/R or frame sync signal. This mode was designed for the original Rev. A Sound Accelerator which had a 15 pin serial connector, and therefore was not able to support multiple signals. In general, new mode means that the device generating the data also generates all clock signals (bit clock, frame sync, and L/R). In the case of external peripherals that can generate as well as receive data (such as the DAT I/O) there are two 'new' modes depending on which source generates the clock signals.

The term 'local mode' means that all settings for the external peripheral (ie left-right-stereo switch on the Ad In) are controlled manually on the peripheral itself. Non-local modes allow front panel settings to be controlled remotely from the Sound Accelerator using various C-Bit settings.

Standard Settings

To remain compatible with all versions of the Sound Accelerator and the Audiomedia card, there are 3 primary CTL_LATCH setting you should use:

Mono, On-Chip Clock: CTL_LATCH = $001327
The SSI generates its own clock using the 56000's internal baud rate generator. The DACs clock the same sample out of both channels. This mode should be used only when you need to control the speed of the SSI clock directly. Generally this is desired when you are interfacing the SSI to custom external hardware.

Stereo, On-Card Clock: CTL_LATCH = $001377
The SSI gets its clock from the Sound Accelerator's on-card (but off-chip) clock and the DACs send separate samples out of the left and right channels. This is the most common mode since it requires no external device to generate a clock, and can accommodate both mono and stereo. Mono playback is achieved by sending the same data out of the left and right channels. The files USamplerPlay.asm and SASample.asm both use this mode for playback.

Stereo, Off-Card Clock: CTL_LATCH = $0013F7

The SSI gets its clock from the expansion connector and the DACs send separate samples out of the left and right channels. Typically, the external clock will be supplied by the Ad In or DAT I/O. If you want to record mono, you should just record one channel. You will need this mode only if you require an off-card clock to drive the SSI or want to receive external data. The file USamplerRecord.asm uses this mode to record and monitor/play sound data.

CTL_LATCH Bit Descriptions

The details for each bit follow but **BEWARE**, all the magic bits (CRB, PCD, CTL_LATCH, etc.) are not mutually exclusive. Some of the bits must be coordinated with others (ie, selecting mono or stereo with CTL_LATCH and OF1). We haven't tested all possible combinations, so start out using the settings in the example programs and then modify things from there making sure that everything is coordinated.

Bits 23-13: Not Used
These bits are reserved for other present or future products.

Bits 12..8: External device control bits:
The control bits (or C-bits) are 4 general purpose output lines used by cards to control external devices such as the Ad In or the DAT I/O. Each external device uses the bits in a different way. For the specific meaning of the C-bits, see the C-bits section in the appropriate external device documentation. Bit 12 has the same use on all cards.

Bit 12: C-bit enable

1 Enable the C-bit output lines.

0 Tri-state the C-bit lines.

Bits 7-4: Clock signal control:

Bit 7	-	Bit 4		Description of state
0	0	0	0	audio output off, 56k internal on-chip clock selected
0	0	0	1	reserved
0	0	1	0	mono audio, 56k internal on-chip clock selected
0	0	1	1	stereo audio, 56k internal on-chip clock selected
0	1	0	0	audio output off, on-card clock selected (88.2 kHz)
0	1	0	1	reserved
0	1	1	0	mono audio, on-card clock selected (88.2 kHz mono)
0	1	1	1	stereo audio, on-card clock selected (44.1kHz stereo)
1	0	0	0	reserved
1	0	0	1	reserved
1	0	1	0	reserved
1	0	1	1	reserved
1	1	0	0	audio off, external (expansion port) clock selected
1	1	0	1	reserved
1	1	1	0	mono audio, external (expansion port) clock ÷ 2 selected
1	1	1	1	stereo audio, external (expansion port) clock selected

Bit 3: Left-Right Signal Source

1 56k reads the left-right signal from the expansion port.

0 56k reads the left-right signal (ie PC1) from the on card circuitry.

✔ **BIT 3 SHOULD ALWAYS BE SET TO THE OPPOSITE OF BIT 1.**

Bit 2: Bit Clock Signal Source

1 SSI derives its bit clock from an off-chip device (either the on card crystal or from the expansion port).

0 SSI derives its bit clock from the on-chip baud rate generator.

Bit 1: Frame Sync/Channel Source and Direction

1 Send frame sync and left-right signals out of the card.

0 Receive frame synch and left-right signals from the expansion connector.

This is useful when trying to synchronize multiple cards whereby one card becomes the master sending frame sync, bit clock, and channel signals to the other cards.

Bit 0: Bit clock/data output control

1 Send bit clock and data from the SSI out to the expansion connector.

0 Don't send bit clock and data from the SSI out to the expansion connector.

This is similar to PC0 in the PCD register.

Sound Accelerator Rev. B and Rev. A SE30 CTL_LATCH bit summary

Bit Number	Function
23	not used (0)
22	not used (0)
21	not used (0)
20	not used (0)
19	not used (0)
18	not used (0)
17	not used (0)
16	not used (0)
15	not used (0)
14	not used (0)
13	not used (0)
12	C-bit enable (1=enable,0=tri-state C-bit output lines)
11	C-bit Bit 3 (see C-bit definitions for a given external device)
10	C-bit Bit 2
09	C-bit Bit 1
08	C-bit Bit 0
07	PAL clock control Bit 3 (see above table)
06	PAL clock control Bit 2
05	PAL clock control Bit 1
04	PAL clock control Bit 0
03	L/R channel source (1=receive off card, 0=generate on card).
02	SSI bit clock source (1=off chip, 0=on chip)
01	Frame Sync direction (1=send frame sync & L/R signal, 0=receive)
00	SSI bit clock enable (1=send bit clock/data to expansion, 0=don't)

Sound Accelerator Rev. A

The Sound Accelerator (Mac II version) is a slave-only Nubus card. As a slave, it cannot initiate Nubus transactions. It can, however, interrupt the main processor on the motherboard via a slot interrupt when it needs attention. Communication with the board takes place via an 8-bit bidirectional host port on the DSP.

✔ The Sound Accelerator Rev. A was the first piece of hardware Digidesign produced. As such, the information in the section is assumed in the sections covering other hardware.

Memory Space

On a Mac II, the board's registers are memory-mapped into the normal Nubus slot space. In 24-bit addressing mode, the base address of the board is $00s00000, where s is the slot number ($9-$E). In 32-bit addressing mode, the base address of the board is $Fs000000, where s is the slot number ($9-$E). All board addresses mentioned will be written as an offset from the board's base address. The board supports byte-wide transactions on ByteLane 3 (byte-wide accesses on addresses whose 2 lsbits are both set, ie.: $00s00003, $00s00007, etc.).

Board Reset

The board will automatically reset on power-up. To manually reset the board under program control, write any byte value to the reset register. The approved method of resetting the board is by making a driver control call to the board's driver. The reason this is safer is that the board takes a small amount of time (typically from 4-20 µsec) to internally reset, and trying to access the board before that time has passed will result in errors.

DSP Registers

The only 56000 registers accessible from Nubus are the registers in its Host Port. These are 8-bit registers, sequentially numbered (in ByteLane 3 addressing) starting at an offset of $00080003. The registers and their offsets are listed below:

Register offsets for Mac II NuBus Boards:

Register Name	Mac II Address Offset
Reset (write-only)	$00000003
ICR (r/w)	$00080003
CVR (r/w)	$00080007
ISR (read-only)	$0008000B
IVR (r/w)	$0008000F
RXH (read-only)	$00080017
RXM (read-only)	$0008001B
RXL (read-only)	$0008001F
TXH (write-only)	$00080017
TXM (write-only)	$0008001B
TXL (write-only)	$0008001F

✔ A read or write of any host port register takes 2 Nubus cycles to complete, which is the minimum amount of time that a NuBus read or write cycle can take.

Board Interrupts

On a Mac II, the board can issue a slot interrupt to the main processor on the motherboard. The HREQ* line of the DSP's host port is attached to the NMRQ* line of the board's Nubus slot. The user can program the DSP's host port to activate the HREQ* pin (driving it low) whenever the the host port's data register is empty or full. Thus, the DSP program can request service via a slot interrupt just by writing to its host port. See chapter 10 in the 56000 User's Manual for more information on the host port and host port interrupts. A useful technique is to write an opcode to the host port in order to generate a slot interrupt which the Mac can interpret as a request to transfer data or an error/debugging message. See USampler.a and USampler.asm as an example.

Board Architecture

The 56000 on the board is connected to the Nubus via its host port. Its external address and data lines are connected to several banks of high-speed, no wait-state static RAM. One bank is for Program Memory and the other two banks are for Data Memory.

- ✔ If you change the memory configuration, you must also change the position of the jumpers in the on-board jumper blocks. Refer to the appendices for details.

The program memory comes with 2K X 24 using 300-mil 2K X 8 45 ns. static RAMs. It is expandable to 8K X 24 using 300-mil 8K X 8 static RAMs, and to 32K X 24 using 300-mil 32K X 8 static RAMs.

The data memory comes with 8K X 24 using 600-mil 8K X 8 55 ns. static RAMs. It is expandable to 16K X 24 using a second bank of 8K X 8 static 45 ns. static RAMs, and to 64K X 24 using two banks of 32K X 8 45 ns. static RAMs. The total amount of available data memory is equally divided between X and Y external memory. On Rev. A cards and Mac SE cards, the second bank of RAM fits into 600-mil sockets. On Rev. B cards, the second bank fits into 300-mil sockets.

The external program and data memory is mapped starting at location $0000 to the DSP. This means that the first 256 X and Y data memory locations and the first 512 program memory locations overlap with the internal DSP memory at those addresses.

The SCI of the DSP is not used, as several of its function pins are used for general purpose I/O. The SSI of the DSP is used for digital audio transmission and reception. The transmit lines go to the two DAC channels on the card, and both the transmit and receive lines go to the digital expansion port.

One clock drives both the transmit and receive lines, and its source can come from one of three places: the on-chip baud rate generator, the on-card crystal oscillator, or an off-card clock generator (via the expansion port).

Using the on-chip baud rate generator, you can generate a wide range of sample rates, in both mono and stereo. However, you cannot generate 44.1 kHz stereo with the on-chip baud rate generator. If you need stereo at that sample rate, you must use the on-card crystal oscillator, which will only generate 44.1 KHz mono and stereo.

Although it is possible to vary the actual sample rate of the card, the audio output circuitry is optimized for a sample rate of 44.1 kHz. This is because the output reconstruction filters are fixed, high-quality filters that cut everything above 22 kHz. If you use a sample rate that is lower than 44.1 kHz, it is possible that the audio output may have some imaging/aliasing components in it that are noticeable.

CTL_LATCH

There is no CTL_LATCH on the Rev. A NuBus Sound Accelerator.

Sound Accelerator SE Rev. A

The Mac SE version of the Sound Accelerator is a slave-only SE-bus card, plus a separate analog card which is installed at the expansion slot connector opening at the rear of the Mac SE. The card cannot initiate any transactions, and sits as memory-mapped I/O in the SE's address space. An 8-position DIP switch on the card allows the selection of the board's base address. Of the 256 possible values, only 72 are valid within the SE's current memory mapping scheme: $50-$57,$60-$8F and $C0-CF. These 8 bits are decoded as address lines A23-A16.

Memory Space

On a Mac SE, the board's registers are memory-mapped into the SE's RAM and I/O space, with the 8-position DIP switch on the card determining address bits A23-A16 of the board's base address. All board addresses mentioned will be written as an offset from the board's base address. The board supports only byte-wide transactions on the lower data bus (byte-wide accesses on odd addresses).

DSP Registers

The only 56000 registers accessible from Nubus are the registers in its Host Port. These are 8-bit registers, sequentially numbered (in ByteLane 3 addressing) starting at an offset of $00080003. The registers and their offsets are listed below:

Register offsets for Mac SE Boards:

Register Name	Mac SE Address Offset
Reset (write-only)	$000003
ICR (r/w)	$004001
CVR (r/w)	$004003
ISR (read-only)	$004005
IVR (r/w)	$004007
RXH (read-only)	$00400B
RXM (read-only)	$00400D
RXL (read-only)	$00400F
TXH (write-only)	$00400B
TXM (write-only)	$00400D
TXL (write-only)	$00400F

On a Mac SE, no additional wait states are introduced by any access of the Sound Accelerator.

Board Interrupts

On a Mac SE, the board can issue a level-1 interrupt to the main processor on the motherboard. The HREQ* line of the DSP's host port is attached to the IPL0 line of the SE bus. The user can program the DSP's host port to activate the HREQ* pin (driving it low) whenever the the host port's data register is empty or full. Thus, the DSP program can request service via a level-1 interrupt just by writing to its host port.

CTL_LATCH

There is no CTL_LATCH on the Rev. A SE Sound Accelerator.

Sound Accelerator Rev. B

The Sound Accelerator Rev. B NuBus card provides additional I/O interface abilities while remaining compatible with the Sound Accelerator Rev. A card. The primary differences between A and B are as follows:

- The expansion port connector is a 25 pin D type connector. The additional pins provide added control over external devices as well as increased flexibility in terms of clock and frame sync signals. The details of the added expansion port control signals are outlined in the 'CTL_LATCH' section.
- There is a set of edge connector pins along the upper edge of the card which allow direct access to the 56000's address and data lines as well as the complete I/O interface. These may be used in future products, hence their use will not be detailed in this document. See the appendices for a complete pinout of the edge connector.
- The extra bank of X/Y data RAM is 300-mil sockets, similar to the program memory sockets on the Rev. A card.
- The 56000 case style is a SLAM package.
- The analog audio output filter circuitry is improved over the Rev. A card.

Otherwise, the two cards are identical. Software written to run on the Rev. A board should run without modification on the Rev. B board as long as the additional capabilities of the Rev. B board (ie CTL_LATCH) are not utilized.

CTL_LATCH

See the CTL_LATCH section for details on the Rev. B Sound Accelerator.

Ad In Analog to Digital Converter

The Ad In is a two-channel analog to digital converter which attaches to the Sound Accelerator's expansion connector. It acts as the recording end of the system. The data it generates is received by the 56000's SSI port and generally read in during a SSI receive interrupt. See the file USampler.asm for details on receiving data from the Ad In.

The front panel switch selects whether the digitized data will be stereo or only the left or right channel. In the left or right position, the Ad In digitizes at a rate of 88.2 kHz, so you should only store every other sample in your programs. You could store the entire signal bandwidth, but the input anti-aliasing filters on the Ad In are optimized for a 44.1kHz signal. Channel selection can also be controlled by software using the CTL_LATCH register. See the Sound Accelerator Rev. B section for more information.

Because the Ad In has a 25 pin connector while the Sound Accelerator Rev. A has a 15 pin connector, a special conversion cable is required. See the appendices for a description of the signals and connector pinouts.

C-Bits

The following details the meaning of C-Bit settings and their associated signal directions/meanings for the Ad In. The signal lines such as 'Bit Clock' refer to lines on the 25 pin connector used on all Digidesign equipment. Most of the settings require the associated clock signal bits in the CTL_LATCH to be set appropriately. For example, if you set the C-Bits for the Ad In to send the L/R signal, you must set bit 1 in the CTL_LATCH to 0 so that the Sound Accelerator receives the L/R signal.

✔ WARNING: Use the following information with care and at your own risk. Every possible combination of settings has not been tested. Only the standard settings in the example code are guranteed to work.

Bit 11..10: Not Used
These bits are reserved for other present or future products. They should always be set to 0.

Bit 9..8: Ad In channel and clock control

00 Ad In generates bit clock, frame sync, and left/right signals with the 56k slaving. The Ad In's input is selected via the front panel switch.

01 Causes the Ad In to digitize and send only the left channel of its inputs at 88.2 kHz regardless of the front panel setting. This allows the Mac/56k to select whether to record mono or stereo signals without the user having to use the front panel switch. This would be achieved by always leaving the front panel switch in the stereo position. The 56k could then select stereo (00 or 11) or mono (01) input.

10 Turns the Ad In off by tri-stating the output buffers.

11 Ad In generates bit clock while the 56k generates frame synch and left/right signals. The Ad In's input is selected via the front panel switch.

Serial connector signal summary

- S = Send, R = Receive, Z = high impedance

Mode Name	Bit 9-8	Bit Clk	L/R	Data
NEW MODE STEREO	00	S	S	S
NEW MODE LEFT	01	S	S	S
MUTE	10	Z	R	Z
OLD MODE STEREO	11	S	R	S

Sound Accelerator SE/30 Rev. A

The Mac SE/30 version of the Sound Accelerator is a slave-only SE direct slot card, plus a separate analog card which is installed at the expansion slot connector opening at the rear of the Mac SE/30. The card cannot initiate any transactions, and sits as memory-mapped I/O in the SE/30's address space. The architecture of the board is virtually identical to the Sound Accelerator Rev. B NuBus board. In fact, the operating system on an SE/30 simulates the slot manager on a Mac II machine so software running on the SE/30 can be written as if it is running on a Mac II class machine.

CTL_LATCH

See the CTL_LATCH section for details on the Rev. B Sound Accelerator.

DAT I/O

The DAT I/O is an external digital audio interface that attaches to the Sound Accelerator's expansion connector. It allows you to receive and send AES/EBU and S/PDIF format serial audio data at 32, 44.1 and 48kHz data rates. It sends and receives data through the SSI port on the 56000 in much the same manner as the Ad In. In fact, if you don't utilize the added C-bits control, the Ad In and DAT I/O look identical from a software point of view. Like the Ad In, it can generate bit clock, frame sync, and left/right signals as well as various combinations of receiving and generating. Unlike the Ad In, the DAT I/O can slave completely to the card. This means that it can receive bit clock, frame sync, and left/right signals from the card while generating and receiving data.

C-Bits

The following details the meaning of C-Bit settings and their associated signal directions/meanings for the DAT I/O. Because the DAT I/O operates at multiple speeds, and it has the ability to both generate and receive bit clock, frame sync, and L/R signal, as well as concurrently receive and send data, its C-Bits are a bit (pun intended) complicated. The signal lines such as 'Bit Clock Out' refer to lines on the 25 pin connector used on all Digidesign equipment. Most of the settings require the associated clock signal bits in the CTL_LATCH to be set appropriately. For example, if you set the C-Bits for the DAT I/O to send the L/R signal, you must set bit 1 in the CTL_LATCH to 0 so that the Sound Accelerator receives the L/R signal.

✔ WARNING: Use the following information with care and at your own risk. Every possible combination of settings has not been tested. Only the standard settings in the example code are guranteed to work.

- S = Send, R = Receive, Z = high impedance
- BC Out = Bit Clock Out
- CB Out = C-Bits Out ie Passed to Ad In connector on back of DAT I/O

Record Source: Ad In

		Receive Mode					Transmit Mode			
Mode Name	Bits 11-8	BC Out	L/R	Data	CB Out		BC Out	L/R	Data	CB Out
PRO 32	0000	Z	R	Z	0000		Z	R	Z	0000
UNDEFINED	0001	S	R	S	0001		S	R	S	0001
DAT 32	0010	S	S	S	0010		S	S	S	0010
OLD LOCAL	0011	Z	R	Z	0011		Z	R	Z	**0010**
PRO 44.1	0100	Z	R	Z	0100		Z	R	Z	0100
UNDEFINED	0101	S	R	S	0101		S	R	S	0101
DAT 48	0110	S	S	S	0110		S	S	S	0110
DAT RCV	0111	S	S	S	0111		S	S	S	**0110**
PRO 44.056	1000	Z	R	Z	1000		Z	R	Z	1000
UNDEFINED	1001	S	R	S	1001		S	R	S	1001
DAT 44.1	1010	S	S	S	1010		S	S	S	1010
NEW LOCAL	1011	Z	R	Z	1011		Z	R	Z	**1010**
PRO 48	1100	Z	R	Z	1100		Z	R	Z	1100
UNDEFINED	1101	S	R	S	1101		S	R	S	1101
DAT EXT.	1110	S	S	S	1110		S	S	S	1110
OLD LOCAL	1111	Z	R	Z	1111		Z	R	Z	**1110**

Record Source: DAT I/O

		Receive Mode					Transmit Mode			
Mode Name	Bits 11-8	BC Out	L/R	Data	CB Out		BC Out	L/R	Data	CB Out
PRO 32	0000	Z	R	Z	0000		Z	R	Z	0000
UNDEFINED	0001	S	R	S	0001		S	R	S	0001
DAT 32	0010	S	S	S	0010		S	S	S	0010
OLD LOCAL	0011	S	R	S	**0010**		S	R	S	**0010**
PRO 44.1	0100	Z	R	Z	0100		Z	R	Z	0100
UNDEFINED	0101	S	R	S	0101		S	R	S	0101
DAT 48	0110	S	S	S	0110		S	S	S	0110
DAT RCV	0111	S	S	S	**0110**		S	S	S	**0110**
PRO 44.056	1000	Z	R	Z	1000		Z	R	Z	1000
UNDEFINED	1001	S	R	S	1001		S	R	S	1001
DAT 44.1	1010	S	S	S	1010		S	S	S	1010
NEW LOCAL	1011	S	S	S	**1010**		S	S	S	**1010**
PRO 48	1100	Z	R	Z	1100		Z	R	Z	1100
UNDEFINED	1101	S	R	S	1101		S	R	S	1101
DAT EXT.	1110	S	S	S	1110		S	S	S	1110
OLD LOCAL	1111	S	R	S	**1110**		S	R	S	**1110**

Audiomedia Rev. A and B

The Audiomedia card is a NuBus card which is very similar to the Rev. B Sound Accelerator. The primary differences are as follows:

- On-card stereo A/D converters as well as stereo D/A converters.
- Consumer level RCA-type audio inputs and outputs.
- High impedance microphone input.
- No digital expansion port connector.
- Faster DSP: 22.578MHz vs 19.7568MHz on a Rev. B Sound Accelerator.
- DMA Circuitry for improved performance.
- Software-selectable Mic/Line switching and input level control.
- New and improved device driver.
- DSP RAM configurations are different.

Analog I/O

The Audiomedia card, unlike the other Sound Accelerator cards, has no digital expansion port or connector. As such, it cannot be hooked up to an AD-In, DAT I/O or other digital peripherals. It does, however, have both analog inputs and outputs. These are in the form of RCA-type connectors on the back panel, 2 for input and 2 for output. The inputs and outputs are adjusted to consumer (-10 dB) levels. It also has a low impedance 1/4" phono jack microphone input.

The DSP has control over the input level (8 selectable attenuation levels), and whether or not the Mic input is enabled. The line input is always enabled, so if the Mic input is also enabled, the Mic signal sums equally to both the left and right analog inputs. The card's Control Latch controls these functions.

The card has a stereo A/D converter that is connected to the analog inputs. It uses an on-card 44.1 kHz clock as its sample rate, like the AD-In. Any DSP recording code written for the AD-In should work for the Audiomedia card as well. The Audiomedia card's Control Latch ignores any references to on-card or off-card clocking, since all clocking is on-card.

Digital Signal Processor

The card has the same Motorola 56000 used on other Sound Accelerators, except that it is running at about 22.578 mHz, or about 15% faster. This translates into about 256 real 56000 instructions (ie MAC X0,Y0,A X:(R5)-,X0 Y:(R2)+N2,Y0 is one instruction) per stereo sample pair at 44.1kHz.

The card is shipped with 8 kwords of external data RAM (4k X memory and 4k Y memory), with expansion to 32 kwords possible (16k X memory and 16k Y memory). It also has sockets for either 8 or 32 kwords of external program RAM. Rev. A Audio Media cards were shipped with 8 kwords of external program RAM. Rev. B Audio Media cards have an additional jumper which allows them to use 2 kword external program RAM chips which they are shipped with. From a developer viewpoint, the Sound Accelerator and Audio Media both come with the same base configuration of memory (4k X, 4k Y, 2k P), but the Sound Accelerator can be expanded to the full capacity of the

56000 (64k X, 64k Y, 64k P) while the Audio Media card can only use a quarter of the full capacity of the 56000 (16k X, 16k Y, 16k P).

DMA Circuitry

To reduce the bus bandwidth that transferring digital audio data requires, Audiomedia has a special DMA circuit on-card, to allow direct access of all external DSP RAM from the NuBus. 16 and 32-bit read and write transfers are supported:

32-bit Write:
The 24 msbits are written to the 24-bit word in DSP RAM.

32-bit Read:
The 24 bits of the word in DSP RAM are transferred to the 24 msbits of the 32-bit word. The 8 lsbits are indeterminate.

16-bit Write:
The 16-bit word is transferred to the 16 msbits of the 24-bit DSP RAM word. The 8 lsbits of the DSP RAM word are cleared to zero.

16-bit Read:
The 16 msbits of the DSP RAM word are transferred to the 16-bit NuBus word.

Consecutive DSP RAM addresses are actually 4 addresses apart in the NuBus address space. This conforms to 32-bit style addressing, although in our case we have only 24 bits, and the extra byte is ignored. Furthermore, Audiomedia does not support byte transfers to the DSP RAM via NuBus. Under ideal conditions, we have been able to transfer 5 channels of 44.1kHz 16 bit digital audio from or to a Macintosh hard disk (Quantum 80 ie about 12-18 msec access time) in real time. This translates to about 4 channels of digital audio accross the DMA circuit under real world conditions.

✔ Note: When using auto-increment mode on the 680x0 to transfer data to or from Audiomedia using DMA, the Audiomedia RAM address must increment by 4 for each transfer, regardless of whether it is a 16 or 32-bit transfer! See the file USACard.p and USACard.a for examples of DMA I/O using the Audiomedia card.

CTL_LATCH Bit Descriptions

The control latch on Audiomedia has extra bits defined for A/D and input level and switching control. All bits are backward-compatible with all other Sound Accelerator cards. For example, setting level control bits on a card without on-board A/D will not hurt anything.

✔ Note: On reset (either a NuBus reset or software generated reset) all data bits will be set to 0. The driver sets the control latch to $001FF4 on power-up ie this will be the state of the bits when your application receives a card. Bits that are marked "not used" should always be set to zero.

Bits 23-20: Not Used
These bits are reserved for other present or future products.

Bit 19: Microphone Switch

1 Enable Mic input which sums the microphone's input mono signal across the stereo line-level analog inputs before the signals reach the ADC's.

0 Disables the Mic input. Only the line level inputs are fed to the ADC's.

Bits 18-16: Input Level Control
These three bits select one of 8 input attenuation levels. A value of 0 gives no attenuation and a value of 7 gives about 18 dB of attenuation. Intermediate values are somewhat linear, with the exception that a value of 3 has less attenuation than a value of 4! This is an unusual side effect of the attenuation circuit. The attenuation circuit is just before the ADC's so both the Mic input signal (if it is enabled) and the line input signals will be attenuated. The approximate attenuation levels are as follows:

D18	D17	D16	Level relative to Full Scale = 0 dB (in dB)
1	1	1	0
1	1	0	-4.86
1	0	1	-7.97
0	1	1	-10.25
1	0	0	-11.63
0	1	0	-14.20
0	0	1	-15.41
0	0	0	-18.85

✔ Note: The attenuation levels are subject to change but the relative order with relation to attenuation will remain the same.

Bit 15: Not Used
This bit is reserved for other present or future products.

Bit 14: DSP RAM NuBus Lockout
When set, NuBus accesses of external DSP RAM via the DMA circuit will be held off. This can be used to synchronize shared data between the card and the Mac ie when implementing semaphores.

1 Lockout the NuBus from accessing the DSP RAM. Any NuBus accesses to the DSP's external RAM will be held off until the bit is set to 0. This can be useful when implementing semaphores between the two processors (56k and 68k) allowing shared data to be accessed atomically.

0 Allow the NuBus to access the DSP RAM. This is the normal mode of operation for DMA access. The NuBus and the DSP arbitrate access to the card's RAM.

✔ WARNING: do not set this bit for more than 256 NuBus clock cycles, or a NuBus access of DSP RAM will result in a Bus Error!

Bit 13: A/D Calibrate
Setting this bit will cause the A/D to go into Power Down mode. Clearing the bit will take the A/D out of power down mode and initiate an internal calibration cycle. The calibration period is 4096 X Left/Right clocks (92.88 ms @ 44.1 kHz). The generation of the Left/Right clock is dependent on the Frame Sync from the 56000 so the 56000 must be in an active transmit and/or receive mode in order to complete the calibration. The minimum time that the bit must remain in power down is 150 ns. An application should not normally need to use this function because the driver on Audio Media cards automatically calibrates the A/D converter when the card is powered up (see below).

Bits 12..8: External device control bits:
The control bits (or C-bits) are 4 general purpose output lines used by cards to control external devices such as the Ad In or the DAT I/O. Because the Audiomedia card lacks an expansion port, the bits have no specific meaning or use. If you are writing Audiomedia specific software, you can ignore their settings. If you are writing software that must run on the Sound Accelerator also, set the bits as described in the C-Bits section of the appropriate external device.

Bit 7-4: Clock control bits:
Bits 7-4 are used on Sound Accelerators to control various clock and expansion interface signals.

Bit 7	-	Bit 4		Description of state
0	0	0	0	audio output off, 56k internal on-chip clock selected
0	0	0	1	reserved
0	0	1	0	mono audio, 56k internal on-chip SSI clock selected
0	0	1	1	stereo audio, 56k internal on-chip SSI clock selected
0	1	0	0	audio output off, on-card clock selected (88.2 kHz)
0	1	0	1	reserved
0	1	1	0	mono audio, on-card off chip clock selected (88.2 kHz mono)
0	1	1	1	stereo audio, on-card off chip clock selected (44.1kHz stereo)
1	0	0	0	reserved
1	0	0	1	reserved
1	0	1	0	reserved
1	0	1	1	reserved
1	1	0	0	audio off, external (expansion port) clock selected
1	1	0	1	reserved
1	1	1	0	mono audio, external (expansion port) clock ÷ 2 selected
1	1	1	1	stereo audio, external (expansion port) clock selected

✔ D7 is not actually hooked up. It is shown here for convenience with its meaning on a Sound Accelerator. On the Sound Accelerator, D7 controls access to the external clock, a function that the Audio Media board does not have. In other words D7 is a don't care on this board, any code written using the external clock will default to the PAL clock.

Bits 3-0: Not Used
These bits are reserved for other present or future products.

Device Driver

The config ROM on the Audiomedia card contains a new version of the Sound Accelerator device driver. Make sure that any Audiomedia card in a multi-card system are installed in the lowest-numbered NuBus slot so that its driver is the one loaded into the system at boot time.

The new driver is 32-bit clean. It goes into 32-bit mode to access any card, and restores the MMU mode afterward. The card base address returned by the driver's Allocate control call is still in 24-bit mode. The translation between addresses in the two modes is as follows:

24-bit mode: $00s0 0000 (s = slot number)
32-bit mode: $Fs00 0000

✔ Note: The files USACard.a and USACard.p in DSPWorkshop contain details on accessing the card in 32 bit mode. In addition, the file 'USACard.p' in DSPWorkshop contains up to date driver interface definitions and examples.

The new driver adds power-on test support for the A/D and DMA circuitry on the Audiomedia card, and its Allocate control call now returns a full flags word, indicating the features an allocated card has, such as A/D, D/A, DMA, etc. The flags word bit definitions are now as follows:

csAllocFlags bit fields for ctlAllocCard driver control call

```
GetAnyAlgorithm     EQU   0     1 = ignore card's signature/algorithm
GetAnyMemSize       EQU   1     1 = get card regardless of mem sizes
GetAnyRefNum        EQU   2     0 = get card regardless of refNum (Rev 4+)
HasDMA              EQU   4     0 = no DMA circuitry
```

```
HasAD                EQU   5      0 = no A/D converters
HasNoDA              EQU   6      1 = no D/A converters
HasRequestedAlg      EQU   15     1 = card has requested sig/alg in it
```

Memory Map

The current Audiomedia memory map (relative to board base address ie $Fs00 0000 in 32 bit mode) is:

Audio Media Rev. A and B Memory Map

Start Addr.	End Addr.	Use
$xxx0 0000		DSP and Board Reset
$xxx4 0000	$xxx5 FFFF	External P Memory (32 kwords available - 2kwords standard)
$xxx6 0000	$xxx6 FFFF	External X Memory (16 kwords available - 4kwords standard)
$xxx7 0000	$xxx7 FFFF	External Y Memory (16 kwords available - 4kwords standard)
$xxx8 0000	$xxx8 001F	56000 Host Port Registers

Appendices

The following pages contain an assortment of diagrams, additional information, tables, and other random details. Most of the information involves specialized advanced uses of the card that go beyond the standard example programs. If you have questions about their meanings, or need advice on advanced applications, please call Digidesign for specific information.

Appendices

Connector is a female 15 pin D-type connector with pinout and signal definitions as shown and described below.

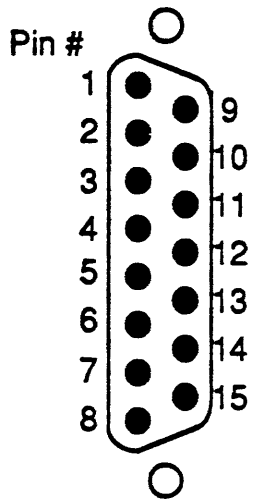


Figure 4. 15-Pin connector as viewed from the back when installed in a Macintosh II computer.

Signals are differential balanced.

Pin #

1. Reserved (Don't Use)
2. Reserved (Don't Use)
3. Left/Right+
4. Bit Clock Out+
5. STD+
6. SRD+
7. Bit Clock In+
8. Sample Sync+
9. Ground
10. Left/Right-
11. Bit Clock Out-
12. STD-
13. SRD-
14. Bit Clock In-
15. Sample Sync-

Explanation of signals:

Left/Right:

Output. A high level indicates that left channel data is being transmitted and a low level indicates that right channel data is being transmitted. For mono this signal should be ignored.

Bit Clock Out:

Output. This clocks the serial audio data at thirty-two times the sampling frequency for stereo and sixteen times the sampling frequency for mono. The data is valid on the falling edge of the clock. This signal is analogous to the TXC signal in the Motorola 56001 data manual.

STD:

Output: Serial output for sixteen bit audio data. The most significant bit is transmitted first.

Bit Clock In:

Input. This input may be used just as an external sample clock or it may be used to clock in serial 16-bit audio samples. Data must be valid on the falling edge of the bit clock. This signal is analogous to the RXC signal in the Motorola 56001 data manual.

SRD:

Input. Serial 16-bit audio data may be input on this pin. The most significant bit should be first.

Sample sync:

Output. A falling edge indicates the completion of the transmission of one sample, or 16-bits, of data regardless of whether it is the left or right channel. This signal may be used to latch audio data or start transmission of the next sample of audio. This signal is analogous to the frame sync signal of the Motorola 56001.

All timing is per the Motorola 56001 technical document ADI1290 Rev. 1, 1988 with the exception of the left/right clock, which is shown on the next page.

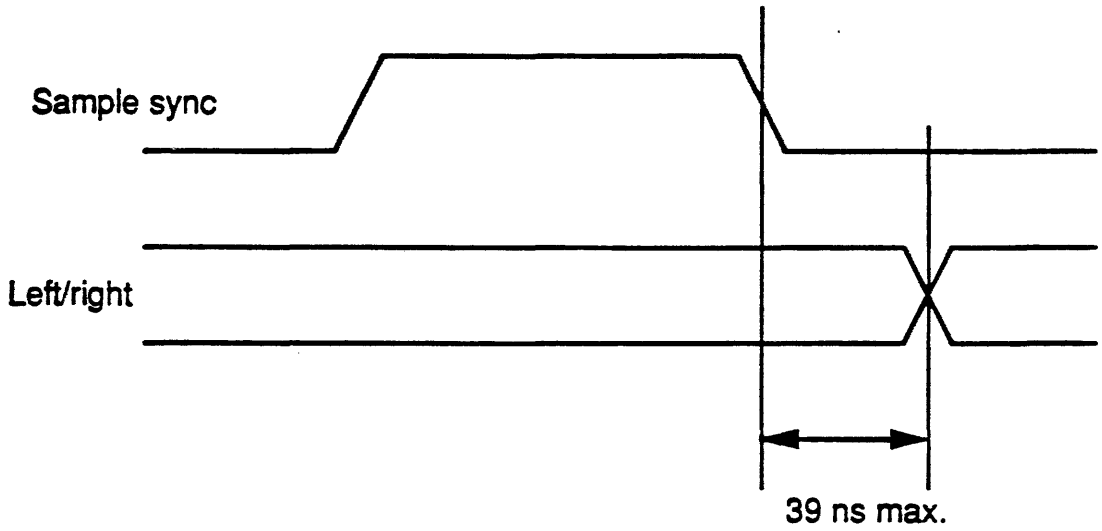


Figure 5. Timing for left/right signal.

Sound Accelerator Rev. A RAM Placement

Installation of static RAM (SRAM):
The board comes from the factory with 8K X 8 SRAMs installed in sockets U1-U3 and 2k X 8 SRAMs in sockets U7-U9.The board can have four different configurations of data SRAM and three different configurations of program SRAM. These configurations are listed below along with the minimum address access time that the parts are required to have. Refer to Figures 2 and 3 for clarification. When changing the configuration of SRAM the corresponding jumpers must be set accordingly. The jumper settings are shown on the following page.

Data SRAM: 1) Bank 1 has 55 nanosecond 8k X 8 SRAMs. Bank 2 is empty. (factory configuration)
2) Bank 1 has 55 nanosecond 8k X 8 SRAMs. Bank 2 has 45 nanosecond 8k X 8 SRAMs.
3) Bank 1 has 55 nanosecond 32k X 8 SRAMs. Bank 2 is empty.
4) Bank 1 has 55 nanosecond 32k X 8 SRAMs. Bank 2 has 45 nanosecond 32k X 8 SRAMs.

Program SRAM: All SRAMs must have 45 nanosecond address access times.
1) U7-U9 have 2K X 8 SRAMs. (factory configuration)
2) U7-U9 have 8k X 8 SRAMs.
3) U7-U9 have 32k X 8 SRAMs.

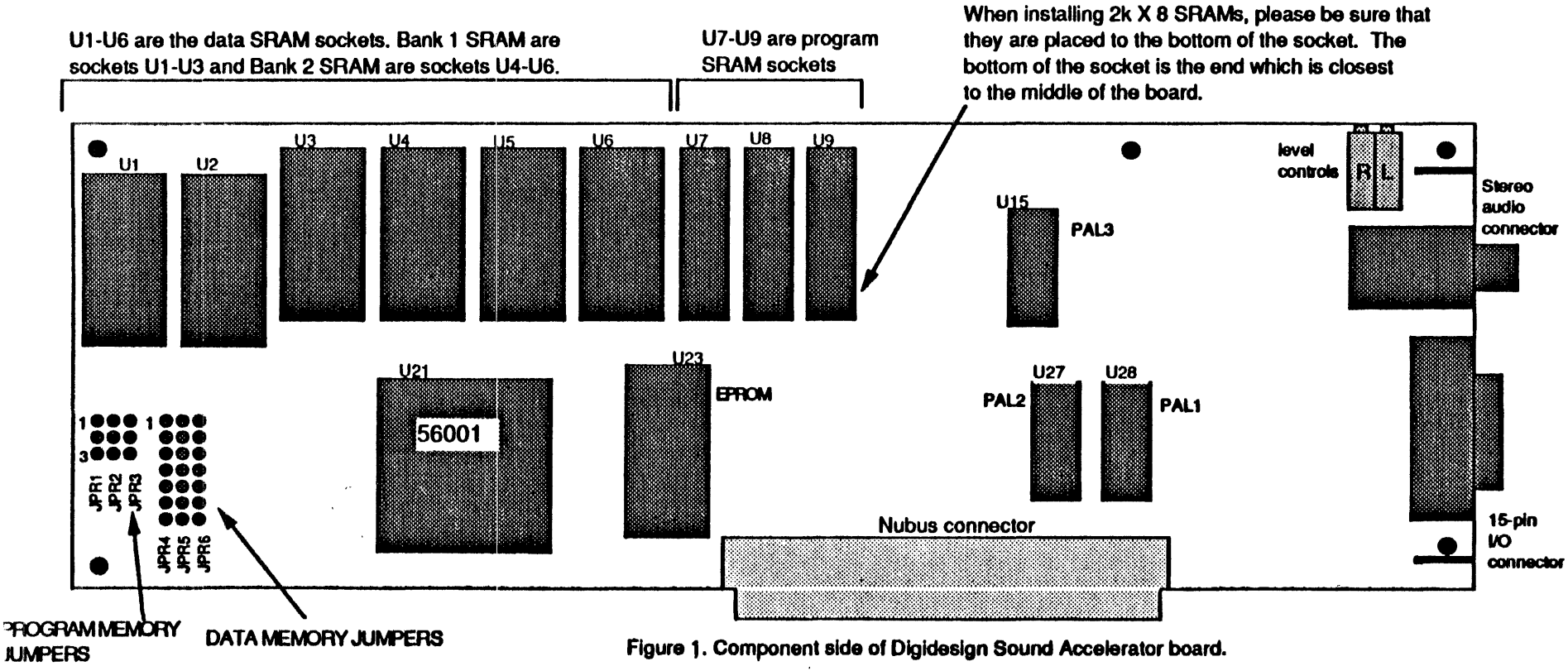
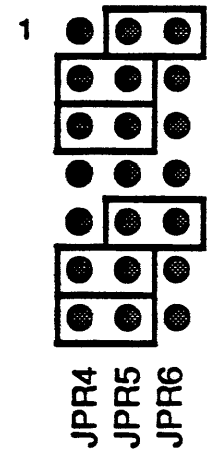
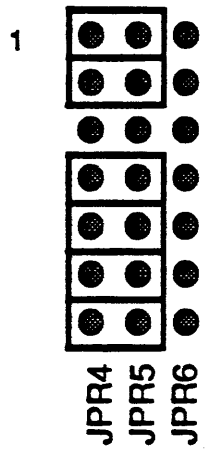


Figure 1. Component side of Digidesign Sound Accelerator board.

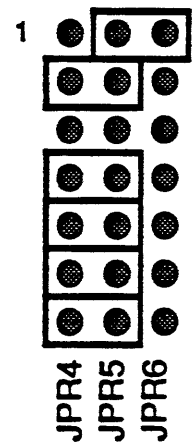
BANK 1: 8k X 8 SRAMs
BANK 2: Empty



BANK 1: 32k X 8 SRAMs
BANK 2: Empty



BANK 1: 8k X 8 SRAMs
BANK 2: 8k X 8 SRAMs



BANK 1: 32k X 8 SRAMs
BANK 2: 32k X 8 SRAMs

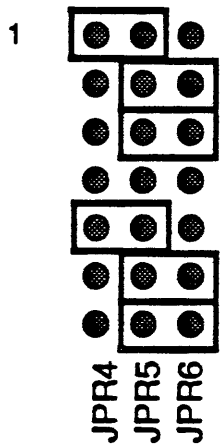
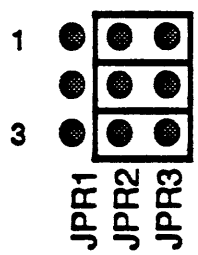
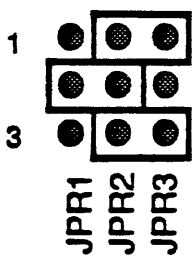


Figure 2. Data SRAM jumper settings

2k X 8 SRAMs



8k X 8 SRAMs



32k X 8 SRAMs

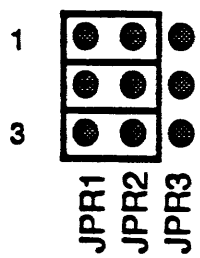


Figure 3. Program SRAM jumper settings

Connector is a female 15 pin D-type connector with pinout and signal definitions as shown and described below.

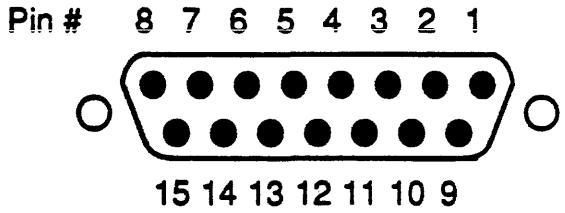


Figure 4. 15-Pin connector as viewed from the back when installed in a Macintosh SE computer.

Signals are differential balanced.

Pin #

1. Reserved (Don't Use)
2. Reserved (Don't Use)
3. Left/Right+
4. Bit Clock Out+
5. STD+
6. SRD+
7. Bit Clock In+
8. Sample Sync+
9. Ground
10. Left/Right-
11. Bit Clock Out-
12. STD-
13. SRD-
14. Bit Clock In-
15. Sample Sync-

Explanation of signals:

Left/Right:

Output. A high level indicates that left channel data is being transmitted and a low level indicates that right channel data is being transmitted. For mono this signal should be ignored.

Bit Clock Out:

Output. This clocks the serial audio data at thirty-two times the sampling frequency for stereo and sixteen times the sampling frequency for mono. The data is valid on the falling edge of the clock. This signal is analogous to the TXC signal in the Motorola 56001 data manual.

STD:

Output: Serial output for sixteen bit audio data. The most significant bit is transmitted first.

Bit Clock In:

Input. This input may be used just as an external sample clock or it may be used to clock in serial 16-bit audio samples. Data must be valid on the falling edge of the bit clock. This signal is analogous to the RXC signal in the Motorola 56001 data manual.

SRD:

Input. Serial 16-bit audio data may be input on this pin. The most significant bit should be first.

Sample sync:

Output. A falling edge indicates the completion of the transmission of one sample, or 16-bits, of data regardless of whether it is the left or right channel. This signal may be used to latch audio data or start transmission of the next sample of audio. This signal is analogous to the frame sync signal of the Motorola 56001.

All timing is per the Motorola 56001 technical document ADI1290 Rev. 1, 1988 with the exception of the left/right clock, which is shown on the next page.

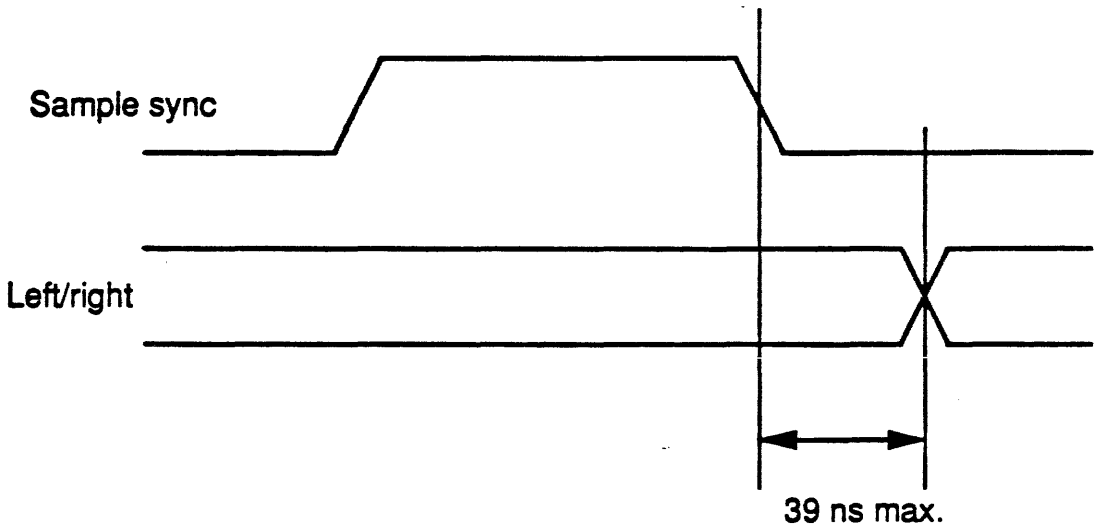


Figure 5. Timing for left/right signal.

Sound Accelerator Rev. B Serial Pinout and Signals

Connector is a female 25 pin D-type connector with pinout and signal definitions as shown and described below.

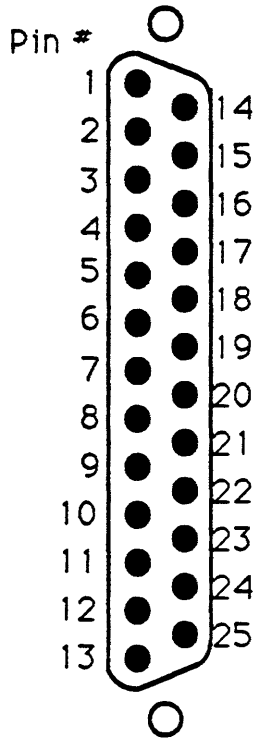


Figure 1. 15-Pin connector as viewed from the back when installed in a Macintosh II computer.

Signals are differential balanced.

Pin #		Pin #	
1.	+5 V	14.	Ground
2.	Control BIT 3+	15.	Control BIT 3-
3.	Control BIT 2+	16.	Control BIT 2-
4.	Serial Data In+	17	Serial Data In-
5.	Bit Clock In+	18.	Bit Clock In-
6.	Left/Right In+	19.	Left/Right In-
7.	Ground		
8.	Sample Sync+	20.	Sample Sync-
9.	Serial Data Out+	21.	Serial Data Out-
10.	Bit Clock Out+	22.	Bit Clock Out-
11.	Control BIT 1+	23.	Control BIT 1-
12.	Control BIT 0+	24.	Control BIT 0-
13.	+12 V	25.	-12 V

Explanation of signals:

Left/Right±:

Bidirectional. A high level indicates that left channel data is being transmitted and a low level indicates that right channel data is being transmitted. For mono this signal should be ignored.

Bit Clock Out±:

Output. This clocks the serial audio data at thirty-two times the sampling frequency for stereo and sixteen times the sampling frequency for mono. The data is valid on the falling edge of the clock. This signal is analogous to the TXC signal in the Motorola 56001 data manual.

Serial Data Out±:

Output: Serial output for sixteen bit audio data. The most significant bit is transmitted first.

Bit Clock In±:

Input. This input may be used just as an external sample clock or it may be used to clock in serial 16-bit audio samples. Data must be valid on the falling edge of the bit clock. This signal is analogous to the RXC signal in the Motorola 56001 data manual.

Serial Data In±:

Input. Serial 16-bit audio data may be input on this pin. The most significant bit should be first.

Sample sync±:

Bidirectional. A falling edge indicates the completion of the transmission of one sample, or 16-bits, of data regardless of whether it is the left or right channel. This signal may be used to latch audio data or start transmission of the next sample of audio. This signal is analogous to the frame sync signal of the Motorola 56001.

Control Bits0-3±:

Outputs. Used to control various states in the Ad In. May be used for other general purpose control.

+5V,-12V,+12V

Voltage outputs from the Macintosh power supply. These pins should not be used or connected to any external device. Use or connection of these pins may cause permanent damage to the Sound Accelerator Card and/or the Macintosh computer.

All timing is per the Motorola 56001 technical document ADI1290 Rev. 1, 1988 with the exception of the signals shown below:

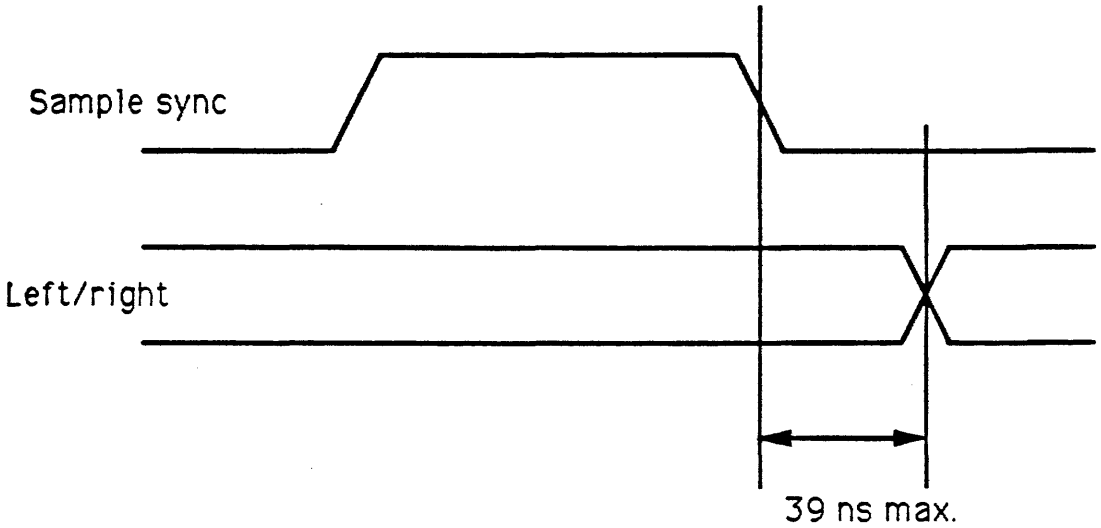


Figure ?. Timing for left/right signal.

When using the external clock please allow for an extra 25 nanoseconds (maximum) of propagation delay before arriving at the 56000 relative to the external Sample sync (Frame Sync) input and Serial Data Input.

Sound Accelerator Rev. B RAM Placement

Installation of static RAM (SRAM):
The board comes from the factory with 8K X 8 SRAMs installed in sockets U1-U3 and 2k X 8 SRAMs in sockets U7-U9.The board can have four different configurations of data SRAM and three different configurations of program SRAM. These configurations are listed below along with the minimum address access time that the parts are required to have. Refer to Figures 2 and 3 for clarification. When changing the configuration of SRAM the corresponding jumpers must be set accordingly. The jumper settings are shown on the following page.

Data SRAM: 1) Bank 1 has 55 nanosecond 8k X 8 SRAMs. Bank 2 is empty. (factory configuration)
2) Bank 1 has 55 nanosecond 8k X 8 SRAMs. Bank 2 has 45 nanosecond 8k X 8 SRAMs.
3) Bank 1 has 55 nanosecond 32k X 8 SRAMs. Bank 2 is empty.
4) Bank 1 has 55 nanosecond 32k X 8 SRAMs. Bank 2 has 45 nanosecond 32k X 8 SRAMs.

Program SRAM: All SRAMs must have 45 nanosecond address access times.
1) U7-U9 have 2K X 8 SRAMs. (factory configuration)
2) U7-U9 have 8k X 8 SRAMs.
3) U7-U9 have 32k X 8 SRAMs.

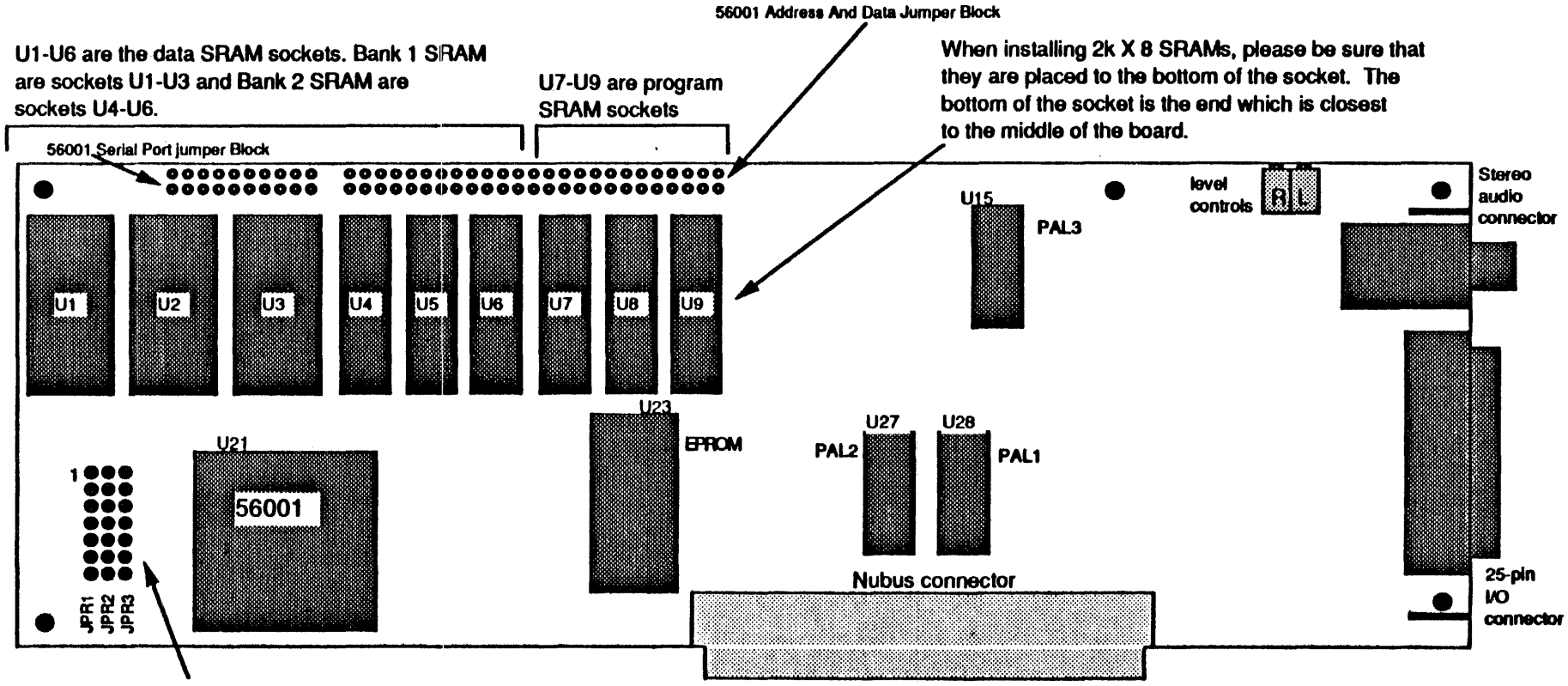
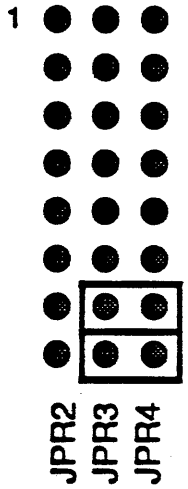


Figure 1. Component side of Digidesign Sound Accelerator Rev B board.

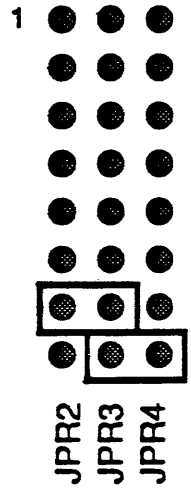
Sound Accelerator Rev. B RAM Jumper Information

Program SRAM Jumper Configurations

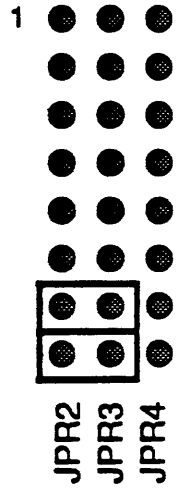
2k X 8 SRAM



8k X 8 SRAM

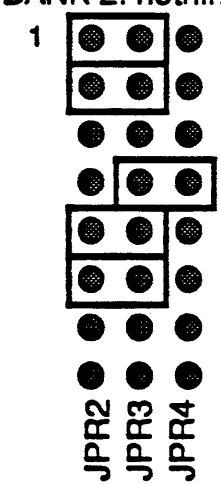


32k X 8 SRAM

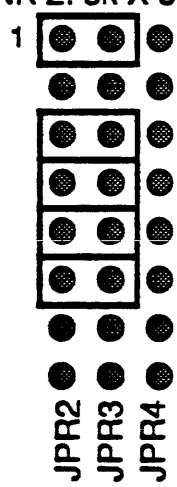


Data SRAM Jumper Configurations

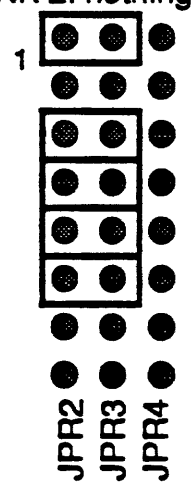
BANK 1: 8k X 8 SRAM
BANK 2: nothing



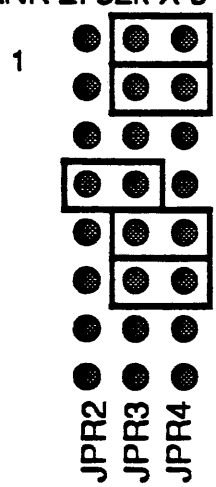
BANK 1: 8k X 8 SRAM
BANK 2: 8k X 8 SRAM



BANK 1: 32k X 8 SRAM
BANK 2: nothing

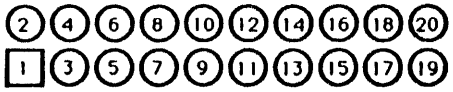


BANK 1: 32k X 8 SRAM
BANK 2: 32k X 8 SRAM



Sound Accelerator Rev. B Header Information

56001 SERIAL PORT JUMPER BLOCK PINOUT AND SIGNAL DESCRIPTION.



1: BITCLK (SCK Pin on 56000)
3: SSI TXDATA (STD Pin on 56000)
5: SSI RXDATA (SRD Pin on 56000)
7: Frame Sync (SC2 Pin on 56000)
9: SC1
11: SC0
13: No Connect
15: SCLK
17: L/R* (left/right signal, TXD Pin on 56000)
19: RXD
2-20: Ground

56001 ADDRESS AND DATA JUMPER BLOCK PINOUT AND DESCRIPTION

1: A0	2: Ground	27: D2	28: D1
3: A2	4: A1	29: D4	30: D3
5: A4	6: A3	31: D6	32: D5
7: A6	8: A5	33: D8	34: D7
9: A8	10: A7	35: D10	36: D9
11: A10	12: A9	37: D12	38: D11
13: A12	14: A11	39: D14	40: D13
15: A14	16: A13	41: D16	42: D15
17: WR*	18: A15	43: D18	44: D17
19: X/Y*	20: RD*	45: D20	46: D19
21: NC (see note)	22: DS*	47: D22	48: D21
23: BR*	24: BG*	49: Ground	50: D23
25: D0	26: Ground		

Note: Should have been PS* but due to PC fab error pin not connected.

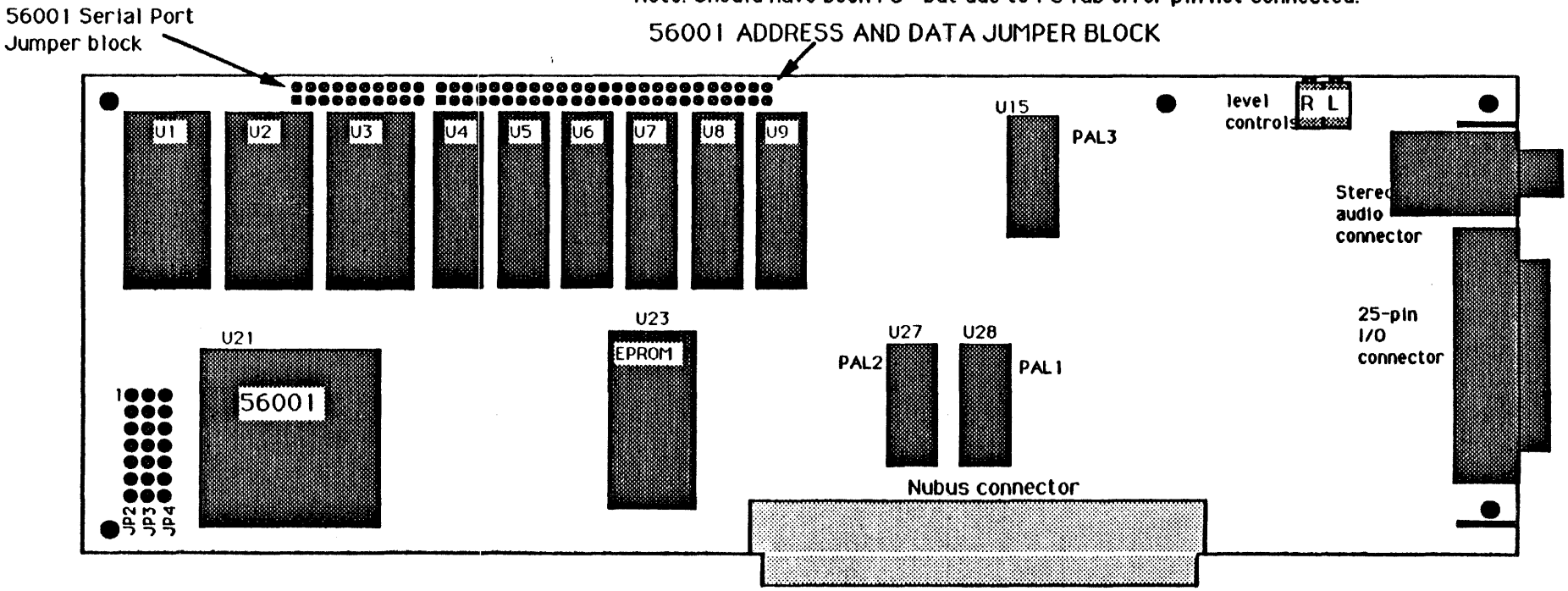
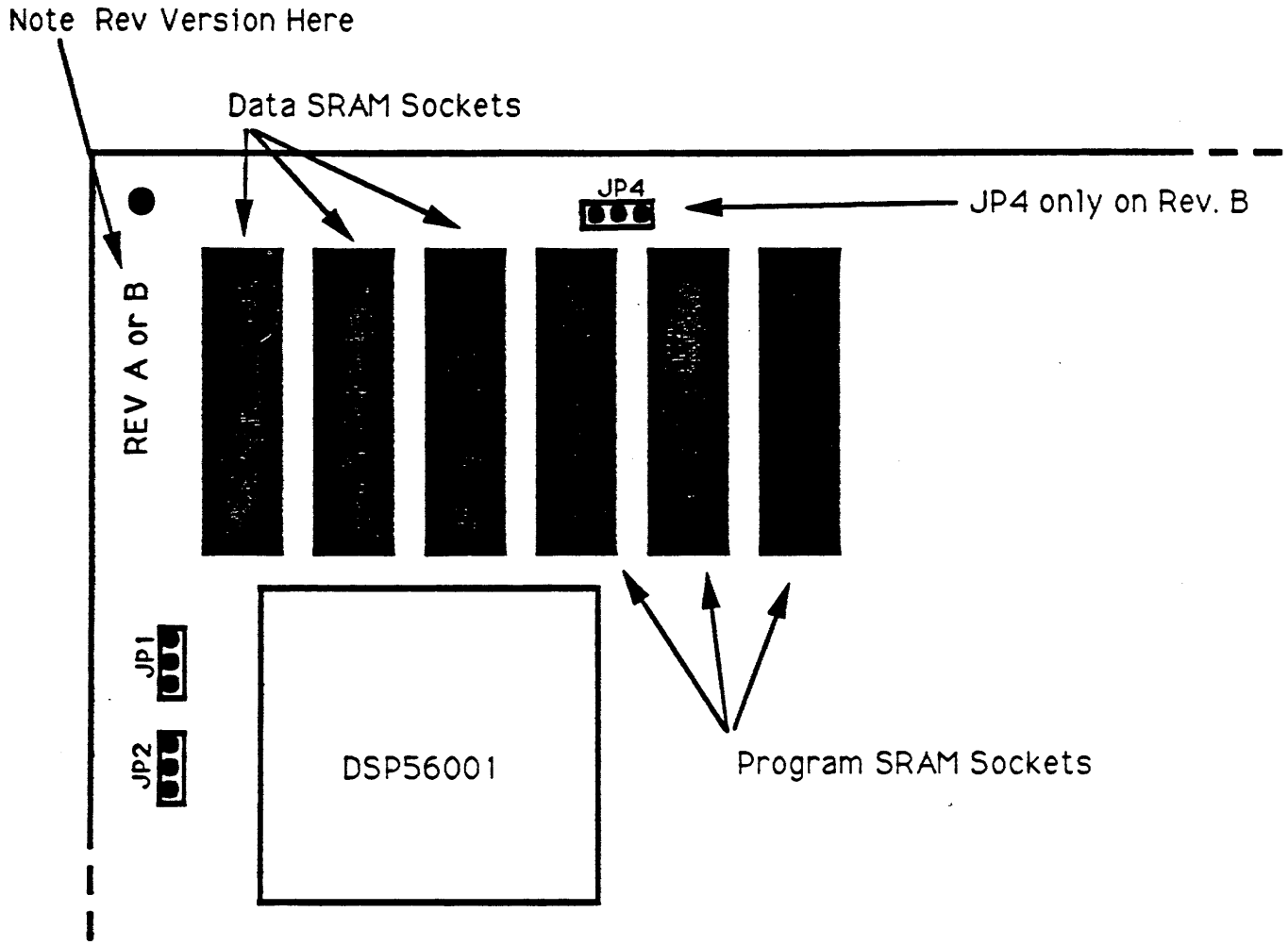
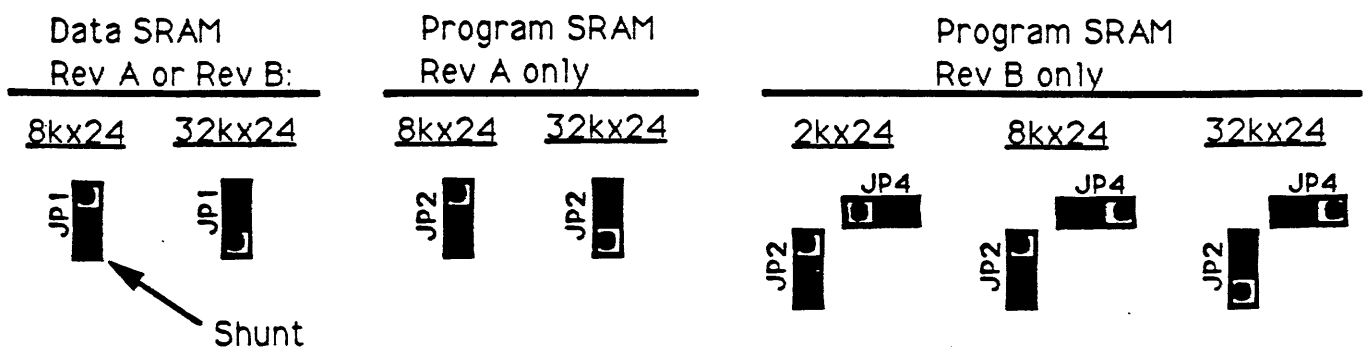


Figure 7. Component side of Digidesign Sound Accelerator board, Revision B.

Audiomedia Memory Upgrade Jumper Configuration



On Rev. A Audio Media boards both Data and Program SRAM may only be configured as either 8k X 24 or 32k X 24.
On the Rev B. Audio Media boards Program SRAM may also be configured as 2k X 24.
The jumper configurations for various SRAM configurations are shown below:



Memory Upgrades

The Sound Accelerator (all models) and Audiomedia cards support a variety of memory configurations. The following are suggested configurations and part numbers. Digidesign no longer supplies memory upgrades due to the volatile nature of the memory market. Most of the following parts are available at reduced cost from national mail order parts companies or from local electronic and computer retailers. The upgrade paths are described in the form of 'kits' that you need to upgrade to a certain configuration.

Upgrade kit #1: Three 8kX8 SRAMs, 600 mil, 45 ns
Upgrade kit #2: Three 32kX8 SRAMs, 600 mil, 55 ns
Upgrade kit #3: Three 32kX8 SRAMs, 600 mil, 45 ns
Upgrade kit #4: Three 8kX8 SRAMs, 300 mil, 45 ns
Upgrade kit #5: Three 32kX8 SRAMs, 300 mil, 45 ns

Memory	Data SRAM		Program	upgrade kits
Upgrade	Bank 0	Bank1	SRAM	needed
A:	8kX8†	-	8kX8	4
B:	8kX8†	-	32kX8	5
C:	8kX8†	8kx8	2kX8	1
E:	8kX8†	8kX8	8kX8	1,4
F:	8kX8†	8kX8	32kX8	1,5
G:	32kX8	-	2kX8	2
H:	32kX8	-	8kX8	2,4
I:	32kX8	-	32kX8	2,5
J:	32kX8	32kX8	2kX8	2,3*
K:	32kX8	32kX8	8kX8	2,3*,4
L:	32kX8	32kX8	32kX8	2,3*,5

* 55 ns parts are placed in Bank 0 and 45 ns parts are placed in Bank 1.
† Comes standard with board

Upgrade kit #1: Three 8kX8 SRAMs, 600 mil, 45 ns

Vendor	Vendor #
Motorola	MCM6164C45 or MCM6164P45
Cypress	CY7C186-45PC or CY7C186L-45PC
IDT	IDT7164S45P or IDT7164L45P

Upgrade kit #2/3: Three 32kX8 SRAMs, 600 mil, 55/45 ns

Vendor	Vendor #
Motorola	MCM6206P55 (55 ns part)
Motorola	MCM6206P45 (45 ns part)
Cypress	CY7C199-55PC (55 ns part)
Cypress	CY7C199-45PC (45 ns part)
IDT	IDT71256S55P or IDT7164L55P (55 ns part)
IDT	IDT71256S45P or IDT7164L45P (45 ns part)

Upgrade kit #4: Three 8kX8 SRAMs, 300 mil, 45 ns

Vendor	Vendor #
Motorola	MCM6264P45
Toshiba	TMM2088P-45
Cypress	CY7C185-45PC or CY7C185L-45PC
IDT	IDT7164S45TC or IDT7164L45TC

Upgrade kit #5: Three 32kX8 SRAMs, 300 mil, 45 ns

Vendor	Vendor #
Cypress	CY7C198-45PC
Toshiba	TC55328P-35

Note: These vendors and other vendors (Hitachi, for example) may have new product out that matches our needs.

FSynch Mod Notes

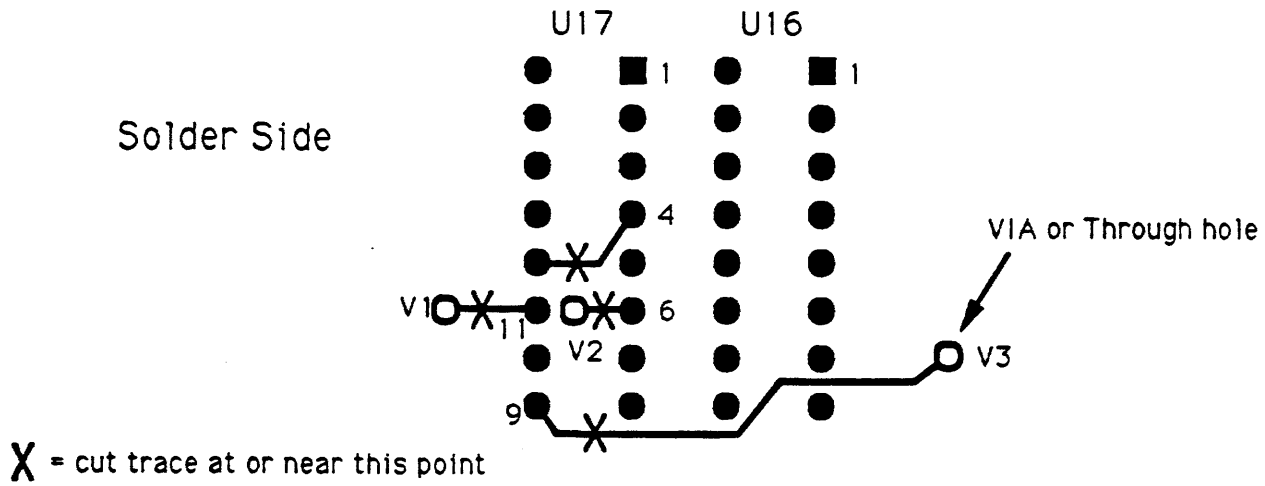
The Revision A Sound Accelerator© (SA) board (Macintosh II board only) will not allow a user to bring digital data into the board via the 15-pin external connector using an external Frame Sync. The solution to this problem can be solved in one of two ways: 1) Enable the 56000 internal Frame Sync and use it with either an external clock or the SA internal clock. With either choice of clock the Frame sync will be generated by the 56000 (thus synchronized to your choice of clock) and can then be brought off board via the 15-pin connector and used to trigger an A/D or other device according to your application. 2) By modifying the hardware. If the Frame Sync can only be generated externally then extensive hardware modifications are necessary. Listed below are the modifications necessary to provide a bi-directional (but enabled for one direction at a time) Frame Sync signal:

These modifications should only be performed by a qualified individual. Please follow all necessary precautions when working with static sensitive devices. **Note that any modification to the board will void the warranty.**

Make the PCB trace cuts as shown in Figure 1.

Make the following jumper connections (Reference Fig. 1 for V connections):
V4 to U17-pin 9
V5 to U17-pin 11
V2 to U17-pin 10
U17-pin 12 to U17-pin 16
V1 to U17-pin 5 and U16-pin 9
V6 to U17-pin 6 and U16-pin 10
V3 to U17-pin 7 and U16-pin 11
U17-pin 4 to U10-pin 3
U16-pin 12 to U10-pin 4

These modifications will provide you with a bidirectional buffer whose direction is controlled by the RXD pin of the SCI port (configured as general purpose I/O). When the RXD pin is not configured, such as after a 56000 hardware reset, then the RXD pin will be tri-stated and the buffer direction signal will be pulled low enabling the Frame Sync input; this is also true when RXD is configured as an output and is deasserted (active high), the input Frame Sync buffer will be enabled and the output Frame Sync buffer disabled. Asserting RXD will enable the output Frame Sync buffer and disable the input Frame Sync buffer. After these modifications the output data and clock buffers will always be active.


Solder Side
U17
U16
1
1
4
VIA or Through hole
V1
11
6
V2
V3
9
X = cut trace at or near this point

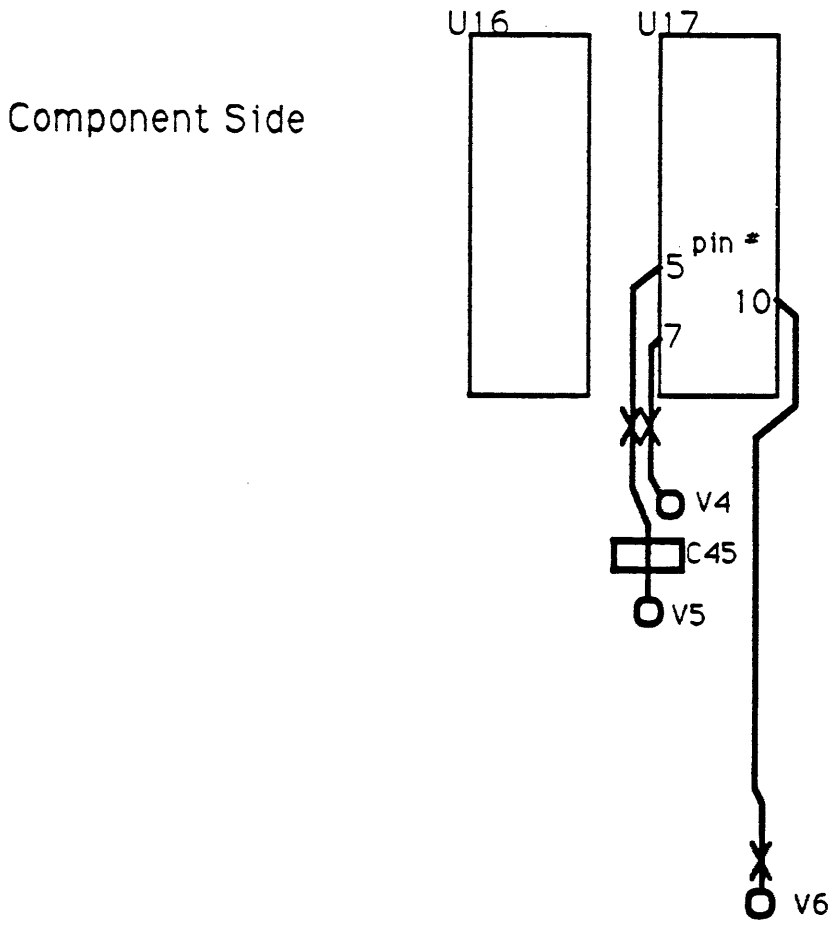

Component Side
U16
U17
pin #
5
10
7
V4
C45
V5
V6

Figure 1

Sound Tools DB-25 Pinout

The Sound Tools system (Sound Accelerator Rev. B SE30 cards, DAT I/O, and Ad In) uses a common 25 pin interface cable. The following line description applies to all hardware in the system.

PIN	SIGNAL	DIRECTION
1	N.C.	
2	CBIT3+	OUT
3	CBIT2+	OUT
4	RXDATA+	IN
5	BITCLKIN+	IN
6	L*/R+	IN/OUT
7	GND	
8	FSYNC+	IN/OUT
9	TXDATA+	OUT
10	BITCLKOUT+	OUT
11	CBIT1+	OUT
12	CBIT0+	OUT
13	N.C.	
14	GND	
15	CBIT3-	OUT
16	CBIT2-	OUT
17	RXDATA-	IN
18	BITCLKIN-	IN
19	L*/R-	IN/OUT
20	FSYNC-	IN/OUT
21	TXDATA-	OUT
22	BITCLKOUT-	OUT
23	CBIT1-	OUT
24	CBIY0-	OUT
25	N.C.	

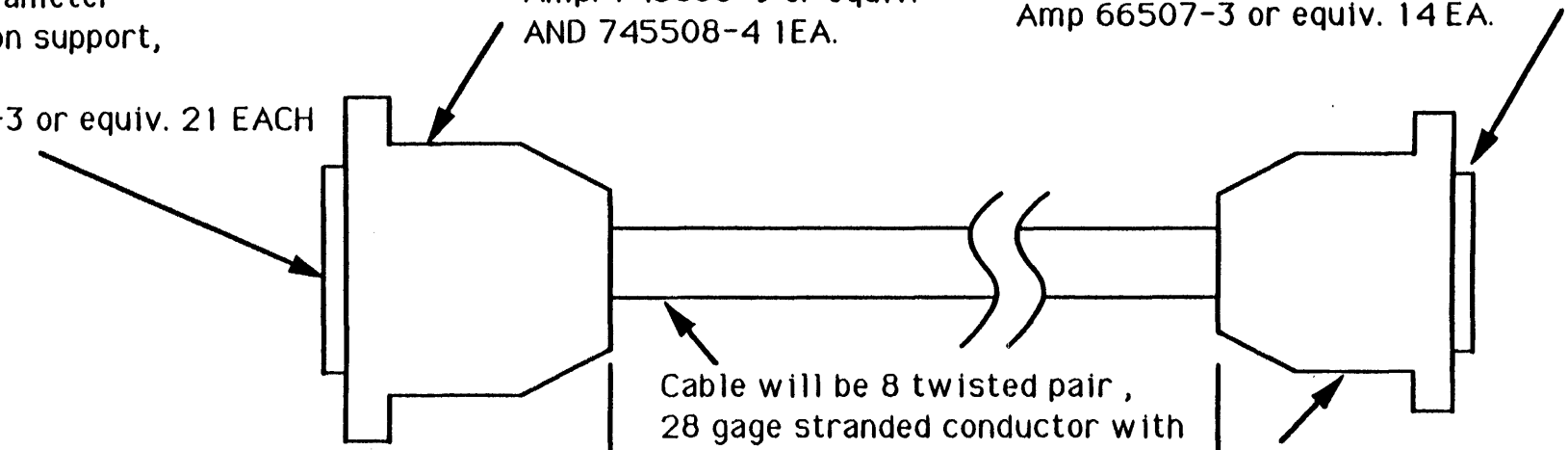
All signals are RS-422 pairs:
"+" indicates non-inverting input/output
"-" indicates inverting input/output
CBIT signals are static control bits used to configure peripheral equipment, and to determine bidirectional signal directions
N.C. = No Connection
GND = 0 volt reference

DAT I/O Rev C 25 Pin to Sound Accelerator Rev. A (II and SE) Cable

25-Pin D-type Male Tin plated shell
with grounding indents.
AMP 207464-2 or equiv. 1 EA.
Pins: 0.04 diameter
w/ insulation support,
strip form.
Amp 66507-3 or equiv. 21 EACH

25 Conductor Shield
(metal plated plastic ok)
Amp: 745833-9 or equiv.
AND 745508-4 1EA.

15-Pin D-type Male Tin plated shell
with grounding indents.
AMP 205206-3 or equiv. 1 EA.
Pins: 0.04 diameter w/ insulation
support, strip form.
Amp 66507-3 or equiv. 14 EA.



Cable will be 8 twisted pair ,
28 gage stranded conductor with
SR-PVC insulation, foil shield,
and chrome PVC jacket.
Belden: 8138
Alpha: 3478
Carol: C0671

6 feet

15 Conductor Shield
(metal plated plastic ok)
Amp: 747099-5 or equiv.
AND 745508-4 1 EA.

The twisted pairs shall be connected as follows:

a-b x-y

(means a is connected to x and b is connected to y, and that this connection is a twisted pair.)

25 Pin	15 Pin	Name
4-17	6-13	RXDATA
5-18	7-14	BC-IN
6-19	3-10	L/R*
8-20	8-15	FSYNC
9-21	5-12	TXDATA
10-22	4-11	BC-OUT
2,3,11,12	1	+12V
14,15,16, 23,24	9	GROUND

Cable shields will be grounded to connector shell at both ends.

Useful Programming Information

Hex	Bin	Hex	Bin
0	0 0 0 0	8	1 0 0 0
1	0 0 0 1	9	1 0 0 1
2	0 0 1 0	A	1 0 1 0
3	0 0 1 1	B	1 0 1 1
4	0 1 0 0	C	1 1 0 0
5	0 1 0 1	D	1 1 0 1
6	0 1 1 0	E	1 1 1 0
7	0 1 1 1	F	1 1 1 1

Mac II Register Offsets:

```
ICR SET $03
CVR SET $07
ISR SET $0B
IVR SET $0F
TXH SET $17
TXM SET $1B
TXL SET $1F
```

Mac SE Register Offsets:

```
ICR = $01
CVR = $03
ISR = $05
IVR = $07
TXH = $0B
TXM = $0D
TXL = $0F
```

```
PCDDR: 0005
PCC: 01F8
```

CTL_LATCH

```
$001327  Mono on chip clock
$001377  Stereo on-card clock
$0013F7  Stereo off-card clock
```

56001 Registers

CRB

Bit 15 … Bit 8

RIE	TIE	RE	TE	MOD	GCK	SYN	FSL
				0	0		1

Bit 7 … Bit 0

		SCKD	SCD2	SCD1	SCD0	OF1	OF0
0	0	0 = ext clk 1 = int clk	1	1	1	0 = mono 1 = stereo	0 = ext clk 1 = int clk

PCD

Bit 7 … Bit 0

					PC2	PC1	PC0
					0 = on card clock source 1 = off card clock source ie enable ext data and clock input	channel flag: 0=play right receive left 1=play left receive right	0 = disable exp. port 1 = enable exp. port

Mac Registers

ICR (R/W)

Bit 7 … Bit 0

INIT	HM1	HM0	HF1	HF0	0	TREQ	RREQ
			host flags set by Mac			1=56k causes slot interrupt when sending	1=56k causes slot interrupt after reading from RX

CVR (R/W)

Bit 7 … Bit 0

HC	0	0	HV
1 = host cmd pending 0 = host cmd acknowledged			code of the current or last host command

ISR (Read-only)

Bit 7 … Bit 0

HREQ	DMA	0	HF3	HF2	TRDY	TXDE	RXDF
1=56k TX register is full or RX register is empty ie generates a slot interrupt			host flags set by 56k		1=both buckets are empty	1=56k is ready to accept some data	1=56k is sending something

AIFF File Format

Audio Interchange File Format: "AIFF"

A Standard for Sampled Sound Files
Version 1.2
Apple Computer, Inc.

The Audio Interchange File Format (Audio IFF) provides a standard for storing sampled sounds. The format is quite flexible, allowing for the storage of monaural or multichannel sampled sounds at a variety of sample rates and sample widths.

Audio IFF conforms to the *"EA IFF 85" Standard for Interchange Format Files* developed by Electronic Arts.

Audio IFF is primarily an **interchange** format, although application designers should find it flexible enough to use as a data storage format as well. If an application does choose to use a different storage format, it should be able to convert to and from the format defined in this document. This will facilitate the sharing of sound data between applications.

Audio IFF is the result of several meetings held with music developers over a period of ten months in 1987-88. Apple Computer greatly appreciates the comments and cooperation provided by all developers who helped define this standard.

Another "EA IFF 85" sound storage format is *"8SVX" IFF 8-bit Sampled Voice*, by Electronic Arts. "8SVX", which handles 8-bit monaural samples, is intended mainly for storing sound for playback on personal computers. Audio IFF is intended for use with a larger variety of computers, sampled sound instruments, sound software applications, and high fidelity recording devices.

Data types

A C-like language will be used to describe data structures in this document. The data types used are listed below:

`char:`	8 bits, signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).
`unsigned char:`	8 bits, unsigned. Contains any number from zero to 255 (inclusive).
`short:`	16 bits, signed. Contains any number from -32,768 to 32,767 (inclusive).
`unsigned short:`	16 bits, unsigned. Contains any number from zero to 65,535 (inclusive).
`long:`	32 bits, signed. Contains any number from -2,147,483,648 to 2,147,483,647 (inclusive).
`unsigned long:`	32 bits, unsigned. Contains any number from zero to 4,294,967,295 (inclusive).
`extended:`	80 bit IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type *Extended*).
`pstring:`	Pascal-style string, a one byte count followed by text bytes. The total number of bytes in this data type should be even. A pad byte can be added at the end of the text to accomplish this. This pad byte is not reflected in the count.
`ID:`	32 bits, the concatenation of four printable ASCII character in the range ' ' (SP, 0x20) through '~' (0x7E). Spaces (0x20) cannot precede printing characters; trailing spaces are allowed. Control characters are forbidden.

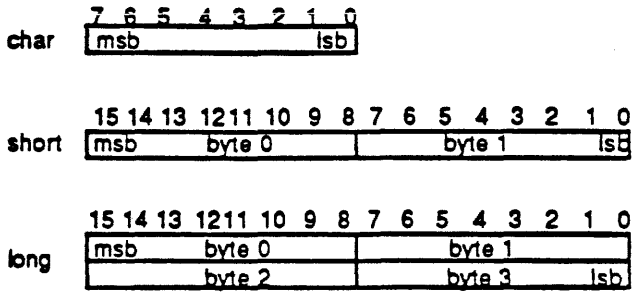
`OSType`: 32 bits. A concatenation of four characters, as defined in *Inside Macintosh, vol II*.

Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g. 0x0A12, 0x1, 0x64.

Data Organization

All data is stored in Motorola 68000 format. Data is organized as follows:



Referring to Audio IFF

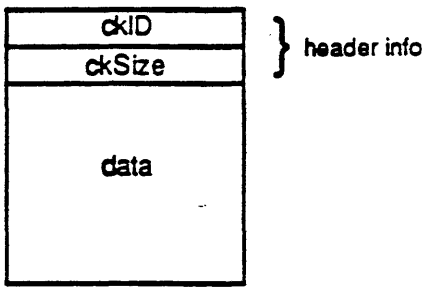
The official name for this standard is *Audio Interchange File Format*. If an application program needs to present the name of this format to a user, such as in a "Save as..." dialog box, the name can be abbreviated to *Audio IFF*.

File Structure

The *"EA IFF 85" Standard for Interchange Format Files* defines an overall structure for storing data in files. Audio IFF conforms to the "EA IFF 85" standard. This document will describe those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85", please refer to the document *"EA IFF 85" Standard for Interchange Format Files*.

An "EA IFF 85" file is made up of a number of *chunks* of data. Chunks are the building blocks of "EA IFF 85" files. A chunk consists of some header information followed by data:

A chunk.



A chunk can be represented using our C-like language in the following manner:

```
typedef struct {

     ID             ckID;          /* chunk ID            */
     long           ckSize;        /* chunk Size          */

     char           ckData[];      /* data                */

} Chunk;
```

ckID describes the format of the *data* portion a chunk. A program can determine how to interpret the chunk data by examining *ckID*.

ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

ckData contains the data stored in the chunk. The format of this data is determined by *ckID*. If the data is an odd number of bytes in length, a zero pad byte must be added at the end. The pad byte is not included in *ckSize* .

Note that an array with no size specification (e.g. `char ckData[];`) indicates a variable-sized array in our C-like language. This differs from standard C.

An Audio IFF file is a collection of a number of different types of chunks. There is a *Common Chunk* which contains important parameters describing the sampled sound, such as it's length and sample rate. There is a *Sound Data Chunk* that contains the actual audio samples. There are several other optional chunks that define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in later sections of this document.

The chunks in a Audio IFF file are grouped together in a container chunk. "EA IFF 85" defines a

number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

```
typedef struct {

    ID              ckID;
    long            ckSize;

    ID              formType;
    char            chunks [];

} Chunk;
```

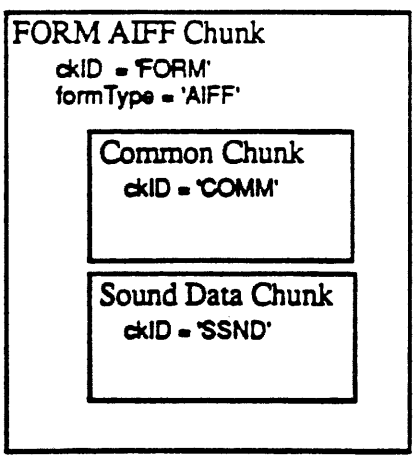
ckID is always 'FORM'. This indicates that this is a FORM chunk.

ckSize contains the size of data portion of the 'FORM' chunk. Note that the data portion has been broken into two parts, *formType* and *chunks*[].

formType describes what's in the 'FORM' chunk. For Audio IFF files, *formType* is always 'AIFF'. This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of *formType* 'AIFF' is called a *FORM AIFF*.

chunks are the chunks contained within the FORM. These chunks are called *local chunks*. A FORM AIFF along with its local chunks make up an Audio IFF file.

Here is an example of a simple Audio IFF file. It consists of a file containing single FORM AIFF which contains two local chunks, a Common Chunk and a Sound Data Chunk.



There are no restrictions on the ordering of local chunks within a FORM AIFF.

On an Apple //, the FORM AIFF is stored in a PRO DOS file. The file type is 0xCB and the aux type is 0x0000.

On a Macintosh, the FORM AIFF is stored in the data fork of an Audio IFF file. The Macintosh file type of an Audio IFF file is 'AIFF'. This is the same as the *formType* of the FORM AIFF.

Macintosh applications should not store any information in Audio IFF file's resource fork, as this information may not be preserved by all applications. Applications can use the *Application Specific Chunk*, defined later in this document, to store extra information specific to their application.

On an operating system that uses file extensions, such as MS-DOS or UNIX, it is recommended that Audio IFF file names have a ".AIF" extension.

A more detailed example of an Audio IFF file can be found in Appendix A. Please refer to this example as often as necessary while reading the remainder of this document.

Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections. The *ckID*s for each chunk are also defined.

There are two types of chunks, those that are required and those that are optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has greater than zero length. All other chunks are optional. All applications that use FORM AIFF must be able to read the required chunks, and can choose to selectively ignore the optional chunks. A program that copies a FORM AIFF should copy all of the chunks in the FORM AIFF.

To insure that this standard remains usable by all developers, only Apple Computer, Inc. should define new chunk types for FORM AIFF. If you have suggestions for new chunk types, Apple is happy to listen! Please refer to Appendix B for instructions on how to send comments to Apple.

Common Chunk

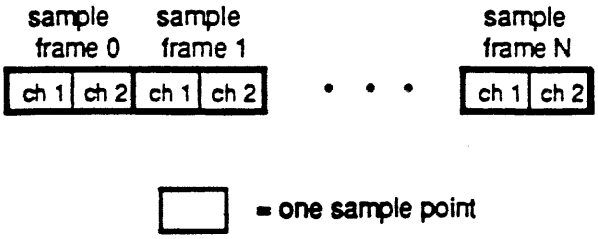
The Common Chunk describes fundamental parameters of the sampled sound.

```
#define    CommonID  'COMM'    /* ckID for Common Chunk */

typedef struct {

     ID             ckID;
     long           ckSize;

     short          numChannels;
     unsigned long  numSampleFrames;
     short          sampleSize;
     extended       sampleRate;

} CommonChunk;
```

ckID is always 'COMM'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*. For the Common Chunk, *ckSize* is always 18.

numChannels contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, and 4 means four channel sound, etc. Any number of audio channels may be represented.

The actual sound samples are stored in another chunk, the *Sound Data Chunk*, which will be described shortly. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a *sample frame*. This is illustrated below for the stereo case.



For monophonic sound, a sample frame is a single sample point.

For multichannel sounds, the following conventions should be observed:

	channel 1	2	3	4	5	6
stereo	left	right				
3 channel	left	right	center			
quad	front left	front right	rear left	rear right		
4 channel	left	center	right	surround		
6 channel	left	left center	center	right	right center	surround

numSampleFrames contains the number of sample frames in the *Sound Data Chunk*. Note that *numSampleFrames* is the number of sample frames, not the number of bytes nor the number of sample points in the *Sound Data Chunk*. The total number of sample points in the file is *numSampleFrames* times *numChannels*.

sampleSize is the number of bits in each sample point. It can be any number from 1 to 32. The format of a sample point will be described in the next section, the *Sound Data Chunk*.

sampleRate is the sample rate at which the sound is to be played back, in *sample frames* per second.

One and only one Common Chunk is required in every FORM AIFF.

Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

```
#define    SoundDataID    'SSND'    /* ckID for Sound Data Chunk */

typedef struct {

    ID                  ckID;
    long                ckSize;

    unsigned long       offset;
    unsigned long       blockSize;
    unsigned char       soundData[];

} SoundDataChunk;
```

ckID is always 'SSND'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

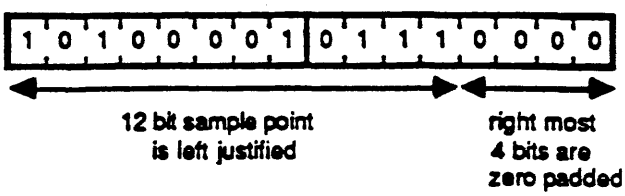
offset determines where the first sample frame in the *soundData* starts. *offset* is in bytes. Most applications won't use *offset* and should set it to zero. Use for a non-zero *offset* is explained in the *Block-Aligning Sound Data* section below.

blockSize is used in conjunction with *offset* for block-aligning sound data. It contains the size in bytes of the blocks that sound data is aligned to. As with *offset*, most applications won't use *blockSize* and should set it to zero. More information on *blockSize* is in the *Block-Aligning Sound Data* section below.

soundData contains the sample frames that make up the sound. The number of sample frames in the *soundData* is determined by the *numSampleFrames* parameter in the *Common Chunk*.

Sample Points

Each sample point in a sample frame is a linear, 2's complement value. The sample points are from 1 to 32 bits wide, as determined by the *sampleSize* parameter in the *Common Chunk*. Sample points are stored in an integral number of contiguous bytes. One to 8 bit wide sample points are stored in one byte, 9 to 16 bit wide sample points are stored in two bytes, 17 to 24 bit wide sample points are stored in 3 bytes, and 25 to 32 bit wide samples are stored in 4 bytes. When the width of a sample point is less than a multiple of 8 bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated below. A 12 bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.



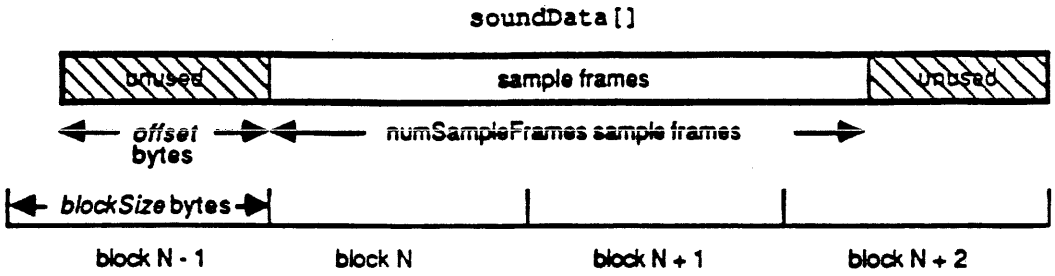
Sample Frames

Sample frames are stored contiguously in order of increasing time. The sample points within a

sample frame are packed together, there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

Block-Aligning Sound Data

There may be some applications that, to insure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This can be accomplished with the *offset* and *blockSize* parameters, as shown below.



Block-aligned sound data

In the above figure, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first *offset* bytes of the *soundData*. Note too that the *soundData* array can extend beyond valid sample frames, allowing the *soundData* array to end on a block boundary.

blockSize specifies the size in bytes of the block that is to be aligned to. A *blockSize* of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set *blockSize* and *offset* to zero when writing Audio IFF files. Applications that write block-aligned sound data should set *blockSize* to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application doesn't preserve alignment, it should set *blockSize* and *offset* to zero. If an application needs to realign sound data to a different sized block, it should update *blockSize* and *offset* accordingly.

The Sound Data Chunk is required unless the *numSampleFrames* field in the *Common Chunk* is zero. A maximum of one Sound Data Chunk can appear in a FORM AIFF.

Marker Chunk

The Marker Chunk contains markers that point to positions in the sound data. Markers can be used for whatever purposes an application desires. The *Instrument Chunk*, defined later in this document, uses markers to mark loop beginning and end points, for example.

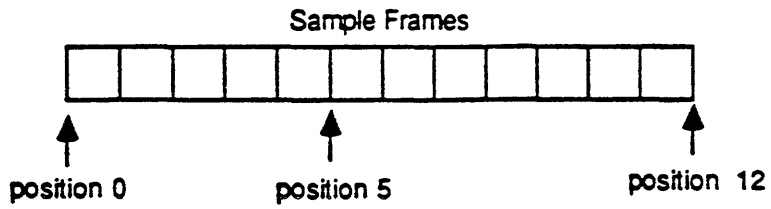
Markers

A marker has the following format.

```
typedef    short     MarkerId;

typedef struct {

     MarkerId              id;
     unsigned long         position;
     pstring               markerName;

} Marker;
```

id is a number that uniquely identifies the marker within a FORM AIFF. The id can be any positive non-zero integer, as long as no other marker within the same FORM AIFF has the same id.

The marker's position in the sound data is determined by *position*. Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at position zero, while a marker that falls between the first and second sample frame in the sound data is at position 1. Note that the units for *position* are sample frames, not bytes nor sample points.



markerName is a Pascal-style text string containing the name of the mark.

Note: Some "EA IFF 85" files store strings as C-strings (text bytes followed by a null terminating character) instead of Pascal-style strings. Audio IFF uses `pstrings` because they are more efficiently skipped over when scanning through chunks. Using `pstrings`, a program can skip over a string by adding the string count to the address of the first character. C strings require that each character in the string be examined for the null terminator.

Marker Chunk Format

The format for the data within a Marker Chunk is shown below.

```
#define    MarkerID  'MARK'    /* ckID for Marker Chunk */

typedef struct {

     ID                  ckID;
     long                ckSize;

     unsigned short      numMarkers;
     Marker              Markers[];

} MarkerChunk;
```

ckID is always 'MARK'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

numMarkers is the number of markers in the Marker Chunk.

numMarkers, if non-zero, it is followed by the markers themselves. Because all fields in a marker are an even number of bytes in length, the length of any marker will always be even. Thus, markers are packed together with no unused bytes between them. The markers need not be ordered in any particular manner.

The Marker Chunk is optional. No more than one Marker Chunk can appear in a FORM AIFF.

Instrument Chunk

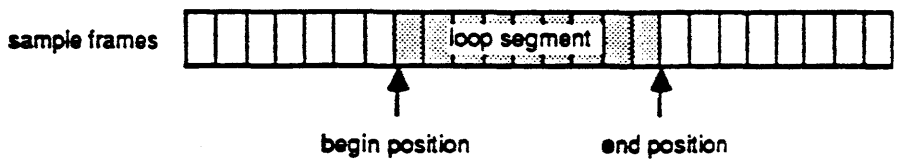
The Instrument Chunk defines basic parameters that an instrument, such as a sampler, could use to play back the sound data.

Looping

Sound data can be looped, allowing a portion of the sound to be repeated in order to lengthen the sound. The structure below describes a loop:

```
typedef struct {

    short          playMode;
    MarkerId       beginLoop;
    MarkerId       endLoop;

} Loop;
```

A loop is marked with two points, a begin position and an end position. There are two ways to play a loop, forward looping and forward/backward looping. In the case of forward looping, playback begins at the beginning of the sound, continues past the begin position and continues to the end position, at which point playback restarts again at the begin position. The segment between the begin and end positions, called the *loop segment*, is played over and over again, until interrupted by something, such as the release of a key on a sampling instrument, for example.



With forward/backward looping, the loop segment is first played from the begin position to the end position, and then played *backwards* from the end position back to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

playMode specifies which type of looping is to be performed.

```
#define    NoLooping              0
#define    ForwardLooping         1
#define    ForwardBackwardLooping 2
```

If `NoLooping` is specified, then the loop points are ignored during playback.

beginLoop is a the marker id that marks the begin position of the loop segment.

endLoop marks the end position of a loop. The begin position must be less than the end position. If this is not the case, then the loop segment has zero or negative length and no looping takes place.

Instrument Chunk Format

The format of the data within an Instrument Chunk is described below.

```
#define    InstrumentID   'INST'    /* ckID for Instrument Chunk */

typedef struct {

     ID             ckID;
     long           ckSize;

     char           baseNote;
     char           detune;
     char           lowNote;
     char           highNote;
     char           lowVelocity;
     char           highVelocity;
     short          gain;
     Loop           sustainLoop;
     Loop           releaseLoop;

} InstrumentChunk;
```

ckID is always 'INST'. *ckSize* is the size of the data portion of the chunk, in bytes. For the Instrument Chunk, *ckSize* is always 20.

baseNote is the note at which the instrument plays back the sound data without pitch modification. Units are MIDI (MIDI is an acronym for Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.

detune determines how much the instrument should alter the pitch of the sound when it is played back. Units are in *cents* (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised.

lowNote and *highNote* specify the suggested range on a keyboard for playback of the sound data. The sound data should be played if the instrument is requested to play a note between the low and high notes, inclusive. The base note does not have to be within this range. Units for *lowNote* and *highNote* are MIDI note values.

lowVelocity and *highVelocity* specify the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is is between low and high velocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

gain is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0 db means no change, 6 db means double the value of each sample point, while -6 db means halve the value of each sample point.

sustainLoop specifies a loop that is to be played when an instrument is sustaining a sound.

releaseLoop specifies a loop that is to be played when an instrument is in the release phase of playing back a sound. The release phase usually occurs after a key on an instrument is released.

The Instrument Chunk is optional. No more than one Instrument Chunk can appear in a FORM AIFF.

MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data (please refer to *Musical Instrument Digital Interface Specification 1.0*, available from the International MIDI Association, for more details on MIDI).

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in this block as well. As more instruments come on the market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the *Instrument Chunk*. For example, a new sampling instrument may have more than the two loops defined in the *Instrument Chunk*. These loops will likely be represented in the MIDI System Exclusive message for the new machine. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

```
#define    MIDIDataID  'MIDI'  /* ckID for MIDI Data Chunk */

typedef struct {

     ID                  ckID;
     long                ckSize;

     unsigned char       MIDIdata[];

} MIDIDataChunk;
```

ckID is always '`MIDI`'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

MIDIData contains a stream of MIDI data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a FORM AIFF. If MIDI System Exclusive messages for several instruments are to be stored in a FORM AIFF, it is better to use one MIDI Data Chunk per instrument than one big MIDI Data Chunk for all of the instruments.

Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

```
#define    AudioRecordingID  'AESD'       /* ckID for Audio Recording */
                                          /*   Chunk.                 */

typedef struct {

     ID                  ckID;
     long                ckSize;

     unsigned char       AESChannelStatusData[24];

} AudioRecordingChunk;
```

ckID is always 'AESD'. *ckSize* is the size of the data portion of the chunk, in bytes. For the Audio Recording Chunk, *ckSize* is always 24.

The 24 bytes of *AESChannelStatusData* are specified in the *AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data,* section 7.1, Channel Status Data. That document describes a format for real-time digital transmission of digital audio between audio devices. This information is duplicated in the Audio Recording Chunk for convenience. Of general interest would be bits 2, 3, and 4 of byte 0, which describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a FORM AIFF.

Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by manufacturers of applications. For example, an application that edits sounds might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, and the like.

```
#define    ApplicationSpecificID  'APPL'  /* ckID for Application */
                                          /*  Specific Chunk.    */

typedef struct {

     ID             ckID;
     long           ckSize;

     OSType         applicationSignature;
     char           data[];

} ApplicationSpecificChunk;
```

ckID is always 'APPL'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

applicationSignature identifies a particular application. For Macintosh applications, this will be the application's four character signature.

data is the data specific to the application.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EA IFF 85" has an *Annotation Chunk* that can be used for comments, but the Comments Chunk has two features not found in the "EA IFF 85" chunk. They are: 1) a timestamp for the comment; and 2) a link to a marker.

Comment

A comment consists of a time stamp, marker id, and a text count followed by text.

```
typedef struct {

    unsigned long       timeStamp;
    MarkerID            marker;
    unsigned short      count;
    char                text;

} Comment;
```

timeStamp indicates when the comment was created. Units are the number of seconds since January 1, 1904. (This time convention is the one used by the Macintosh. For procedures that manipulate the time stamp, see The Operating System Utilities chapter in *Inside Macintosh, vol II*).

A comment can be linked to a marker. This allows applications to store long descriptions of markers as a comment. If the comment is referring to a marker, then *marker* is the ID of that marker. Otherwise, *marker* is zero, indicating that this comment is not linked to a marker.

count is the length of the text that makes up the comment. This is a 16 bit quantity, allowing much longer comments than would be available with a `pstring`.

text contains the comment itself. This text must be padded with a byte at the end to insure that it is an even number of bytes in length. This pad byte, if present, is not included in *count*.

Comments Chunk Format

```
#define    CommentID        'COMT'    /* ckID for Comments Chunk.  */

typedef struct {

    ID                  ckID;
    long                ckSize;

    unsigned short      numComments;
    Comment             comments[];

} CommentsChunk;
```

ckID is always 'COMT'. *ckSize* is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by *ckID* and *ckSize*.

numComments contains the number of comments in the Comments Chunk. This is followed by

the comments themselves. Comments are always an even number of bytes in length, so there is no padding between comments in the Comments Chunk.

The Comments Chunk is optional. No more than one Comments Chunk may appear in a single FORM AIFF.

Text Chunks - Name, Author, Copyright, Annotation

These four chunks are included in the definition of every "EA IFF 85" file. All are text chunks; their data portion consists solely of text. Each of these chunks is optional.

```
#define    NameID          'NAME'    /* ckID for Name Chunk.  */
#define    AuthorID        'AUTH'    /* ckID for Author Chunk.  */
#define    CopyrightID     '(c) '    /* ckID for Copyright Chunk.  */
#define    AnnotationID    'ANNO'    /* ckID for Annotation Chunk.  */

typedef struct {

     ID                    ckID;
     long                  ckSize;

     char                  text[];

} TextChunk;
```

ckID is either 'NAME', 'AUTH', '(c) ', or 'ANNO', depending on whether the chunk as a Name Chunk, Author Chunk, Copyright Chunk, or Annotation Chunk, respectively. For the Copyright Chunk, the 'c' is lowercase and there is a space (0x20) after the close parenthesis.

ckSize is the size of the data portion of the chunk, in this case the *text*.

text contains pure ASCII characters. It is not a `pstring` nor a C string. The number of characters in *text* is determined by *ckSize*. The contents of *text* depend on the chunk, as described below:

Name Chunk

text contains the name of the sampled sound. The Name Chunk is optional. No more than one Name Chunk may exist within a FORM AIFF.

Author Chunk

text contains one or more author names. An author in this case is the creator of a sampled sound. The Author Chunk is optional. No more than one Author Chunk may exist within a FORM AIFF.

Copyright Chunk

The Copyright Chunk contains a copyright notice for the sound. *text* contains a date followed by the copyright owner. The chunk ID '(c) ' serves as the copyright characters '©'. For example, a Copyright Chunk containing the text "1988 Apple Computer, Inc." means "© 1988 Apple Computer, Inc."

The Copyright Chunk is optional. No more than one Copyright Chunk may exist within a FORM AIFF.

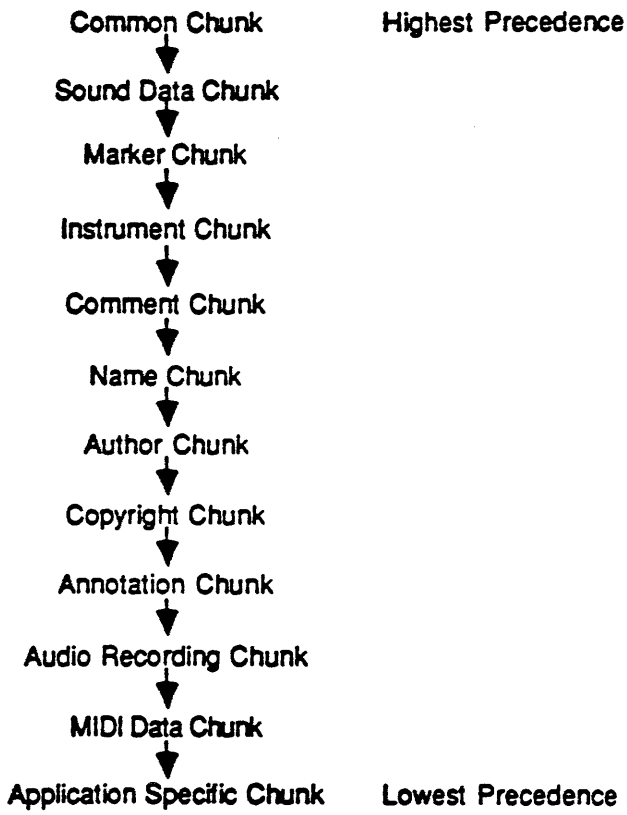
Annotation Chunk

text contains a comment. Use of this chunk is discouraged within FORM AIFF. The more powerful *Comments Chunk* should be used instead. The Annotation Chunk is optional. Many Annotation Chunks may exist within a FORM AIFF.

Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the *Instrument Chunk* defines loop points and MIDI system exclusive data in the *MIDI Data Chunk* may also define loop points. What happens if these loop points are different? How is an application supposed to loop the sound?

Such conflicts are resolved by defining a precedence for chunks:

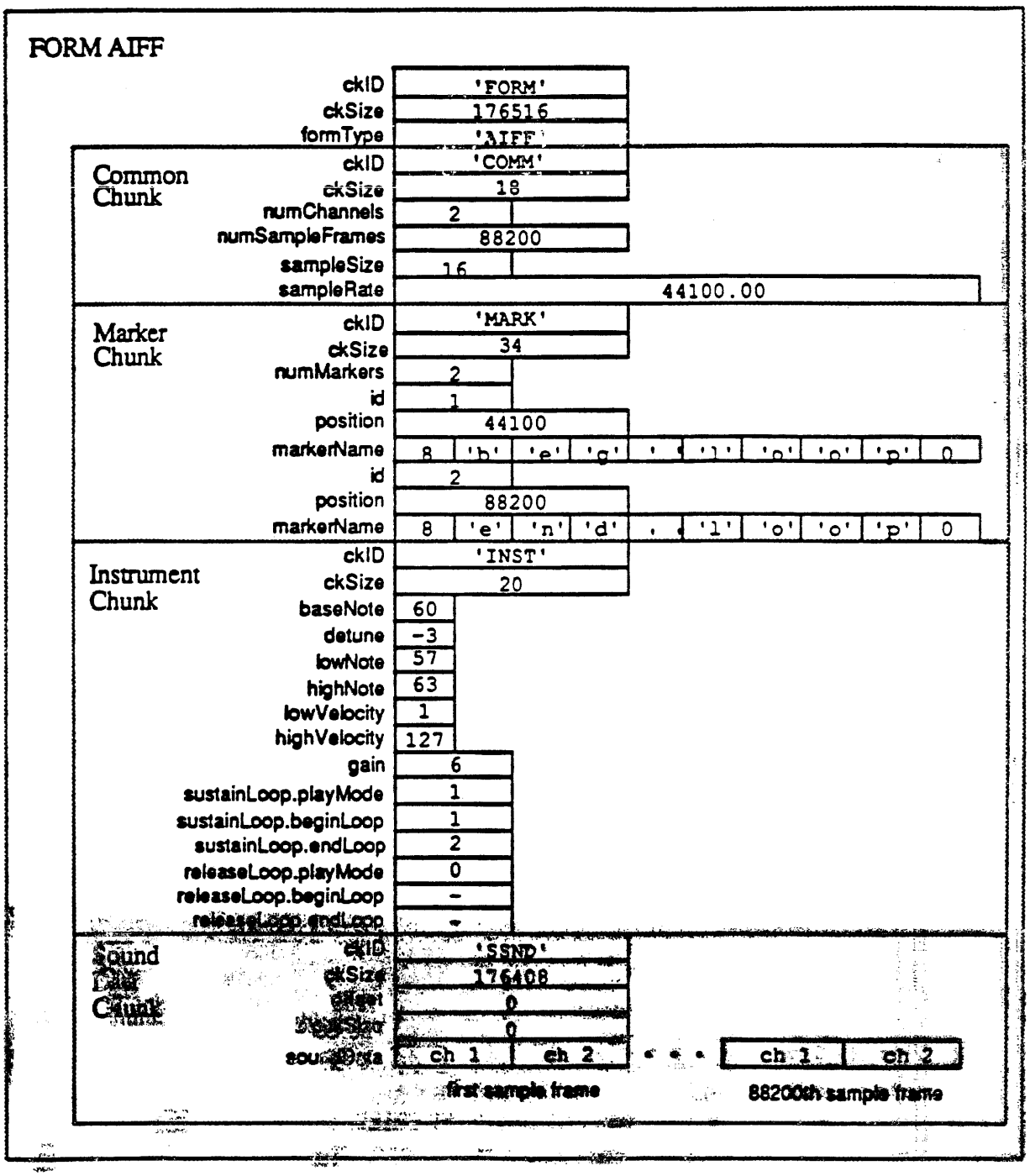


The *Common Chunk* has the highest precedence, while the *Application Specific Chunk* has the lowest. Information in the *Common Chunk* always takes precedence over conflicting information in any other chunk. The *Application Specific Chunk* always loses in conflicts with other chunks. By looking at the chunk hierarchy, for example, one sees that the loop points in the *Instrument Chunk* take precedence over conflicting loop points found in the *MIDI Data Chunk*.

It is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly.

Appendix A - An Example

Illustrated below is an example of a FORM AIFF. An Audio IFF file is simply a file containing a single FORM AIFF. On a Macintosh, the FORM AIFF is stored in the data fork of a file and the file type is 'AIFF'.



Appendix B - Sending comments to Apple Computer, Inc.

If you have suggestions for new chunks to be added to the Audio Interchange File Format, please describe the chunk in as much detail as possible, and give an example of its use. Suggestions for new FORMs, ways to group FORM AIFF's into a bank, and new local chunks are welcome. When sending in suggestions, be sure to mention that your comment refers to the *Audio Interchange File Format: "AIFF"* document.

Send comments to:

Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, MS: 27-T
Cupertino, CA 95014 USA

References

AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data, Audio Engineering Society, 60 East 42nd Street, New York, New York 10165

MIDI: Musical Instrument Digital Interface, Specification 1.0, the International MIDI Association.

"EA IFF 85" Standard for Interchange Format Files. Electronic Arts.

"8SVX" IFF 8-Bit Sampled Voice. Electronic Arts.

Inside Macintosh, Volume II. Apple Computer, Inc., Addison-Wesley Publishing Company, Inc., 1986.

Apple® Numerics Manual, Addison Wesley Publishing Company, Inc., 1986.