**rPod10.2, rPod8.4, MS-II, and MS-IIE**

**Software Description Manual**

**Simon-Kaloi Engineering, LTD**

# Table of Contents

[Table of Contents 2](#_Toc143486175)

[Command Entry 7](#_Toc143486176)

[Sequence Program Development 7](#_Toc143486177)

[Execution Modes: 8](#_Toc143486178)

[Sequence Structure: 9](#_Toc143486179)

[Programs: 9](#_Toc143486180)

[Group files: 9](#_Toc143486181)

[Events Lists: 9](#_Toc143486182)

[Cue Lists: 9](#_Toc143486183)

[Layout: 10](#_Toc143486184)

[Key points: 10](#_Toc143486185)

[Math and Logical Operations: 11](#_Toc143486186)

[Decision Operations: 12](#_Toc143486187)

[Looping Operations: 13](#_Toc143486188)

[Constants: 13](#_Toc143486189)

[Common Variables: 14](#_Toc143486190)

[Trigger Variables: 15](#_Toc143486191)

[Passing Processor Data (rPod8.4 only): 15](#_Toc143486192)

[String Operations: 16](#_Toc143486193)

[String Concatenation: 16](#_Toc143486194)

[String Comparison: 16](#_Toc143486195)

[String to Number: 16](#_Toc143486196)

[Number to String: 17](#_Toc143486197)

[String Length: 17](#_Toc143486198)

[Search String: 17](#_Toc143486199)

[Left String: 18](#_Toc143486200)

[Right String: 18](#_Toc143486201)

[Mid String: 19](#_Toc143486202)

[Sound Channel Information; 20](#_Toc143486203)

[Sound Channel Output Information 20](#_Toc143486204)

[Sound Channel Input Gain Information: 20](#_Toc143486205)

[Sound Track Operations: 21](#_Toc143486206)

[Single Track Information: 21](#_Toc143486207)

[Single Track Progress: 22](#_Toc143486208)

[Track Lists: 23](#_Toc143486209)

[System variables: 24](#_Toc143486210)

[Audio and Control Status: 24](#_Toc143486211)

[Accessing Clip Parameters: 25](#_Toc143486212)

[System Information: 26](#_Toc143486213)

[Ethernet: 27](#_Toc143486214)

[Serial Communications: 27](#_Toc143486215)

[Triggers: 28](#_Toc143486216)

[Expansion Triggers: 30](#_Toc143486217)

[DMX Input: 31](#_Toc143486218)

[Compact Flash and File Status: 31](#_Toc143486219)

[Random Numbers: 32](#_Toc143486220)

[Real Time Clock: 33](#_Toc143486221)

[System Timer: 34](#_Toc143486222)

[General Purpose Timers: 36](#_Toc143486223)

[Timer Based Actions: 37](#_Toc143486224)

[Event Lists: 37](#_Toc143486225)

[Event List behavior: 38](#_Toc143486226)

[Embedded Timer Points: 38](#_Toc143486227)

[Cue Lists 40](#_Toc143486228)

[rPod8.4 Considerations: 41](#_Toc143486229)

[Quick Reference Guide: 42](#_Toc143486230)

[File access: 42](#_Toc143486231)

[Sound File Playback: 42](#_Toc143486232)

[Sound Volume Control: 43](#_Toc143486233)

[Clips: 43](#_Toc143486234)

[DMX/RS485 and Output Control Commands: 43](#_Toc143486235)

[DMX File Playback: 43](#_Toc143486236)

[DMX File Capture: 44](#_Toc143486237)

[Ethernet: 44](#_Toc143486238)

[System Commands: 44](#_Toc143486239)

[Logging 45](#_Toc143486240)

[Sequence Programming Direct Commands: 45](#_Toc143486241)

[Program Flow: 46](#_Toc143486242)

[Event Programming: 47](#_Toc143486243)

[Timer Synchronization: 47](#_Toc143486244)

[Reserved words: 48](#_Toc143486245)

[Table of Operations: 50](#_Toc143486246)

[Table of Operators: 50](#_Toc143486247)

[Table of System Constants and Subroutines: 50](#_Toc143486248)

[Command Description Index 51](#_Toc143486249)

[File Access Commands 51](#_Toc143486250)

[Change drive: 51](#_Toc143486251)

[Directory: 51](#_Toc143486252)

[Change Directory: 52](#_Toc143486253)

[Make Directory: 52](#_Toc143486254)

[Remove Directory: 52](#_Toc143486255)

[Delete: 53](#_Toc143486256)

[Rename: 53](#_Toc143486257)

[Openfile: 54](#_Toc143486258)

[Closefile: 55](#_Toc143486259)

[File Configuration: 55](#_Toc143486260)

[Write File: 56](#_Toc143486261)

[Read File: 57](#_Toc143486262)

[Seek File: 59](#_Toc143486263)

[Sound File Playback 60](#_Toc143486264)

[Pre-loading a Sound: 60](#_Toc143486265)

[Unloading a Sound Track: 61](#_Toc143486266)

[Playing Sounds: 62](#_Toc143486267)

[Looping: 63](#_Toc143486268)

[Un-Looping: 64](#_Toc143486269)

[Convert to Looping 64](#_Toc143486270)

[Appending: 64](#_Toc143486271)

[Stopping Sound Tracks: 64](#_Toc143486272)

[Pausing Sound Tracks: 65](#_Toc143486273)

[Mirroring Sound Tracks (rPod only): 66](#_Toc143486274)

[Positioning Sound Tracks: 66](#_Toc143486275)

[Frequency Pitching Sound Tracks: 67](#_Toc143486276)

[Setting the Global Playback Sample Rate: 67](#_Toc143486277)

[Sound Volume Control 68](#_Toc143486278)

[Channel Volume Control: 68](#_Toc143486279)

[Track Volume Control: 69](#_Toc143486280)

[Ducking Volume Setting: 69](#_Toc143486281)

[Duck Track Control: 70](#_Toc143486282)

[Input Volume Setting: 70](#_Toc143486283)

[Patch Assignment: (rPod10.2, rPod8.4 only) 71](#_Toc143486284)

[Input Mapping: 72](#_Toc143486285)

[Input Control: (rPod10.2, rPod8.4 only) 72](#_Toc143486286)

[Input Control: 73](#_Toc143486287)

[Boost Microphone Gain: 74](#_Toc143486288)

[Mute: 74](#_Toc143486289)

[Un-Mute: 74](#_Toc143486290)

[Track Playback and Ducking Attack Time: 75](#_Toc143486291)

[Track Playback and Ducking Decay Time: 75](#_Toc143486292)

[Amplifier Control: (MSII only) 76](#_Toc143486293)

[Clip Based Metaphors 77](#_Toc143486294)

[Load Clip Command: 77](#_Toc143486295)

[Clip Command: 78](#_Toc143486296)

[Un-loading a Clip from memory: 78](#_Toc143486297)

[Clip Metaphor Data Base: 79](#_Toc143486298)

[Clip Manager Program: 80](#_Toc143486299)

[Connecting to a Clip Database: 80](#_Toc143486300)

[Navigating the Clip Database 82](#_Toc143486301)

[Clip Database Fields 83](#_Toc143486302)

[Track Database Fields 84](#_Toc143486303)

[Exporting Clips 85](#_Toc143486304)

[DMX/RS485 and Output Control Commands 86](#_Toc143486305)

[Bit Level Control: 86](#_Toc143486306)

[Byte Level Control: 87](#_Toc143486307)

[Dimming Control: 87](#_Toc143486308)

[DMX/RS485 Output Driver Control: 88](#_Toc143486309)

[DMX File Playback 88](#_Toc143486310)

[DMX file Pre-loading: 89](#_Toc143486311)

[Unloading DMX track: 90](#_Toc143486312)

[Play DMX File: 90](#_Toc143486313)

[DMX Looping: 91](#_Toc143486314)

[DMX File Mounting: 91](#_Toc143486315)

[Un-mounting DMX Files: 92](#_Toc143486316)

[Stopping DMX Tracks: 92](#_Toc143486317)

[Stopping all DMX Tracks: 92](#_Toc143486318)

[Positioning DMX Playback: 93](#_Toc143486319)

[DMX File Capture 94](#_Toc143486320)

[Record DMX: 94](#_Toc143486321)

[Pre-load Record DMX: 94](#_Toc143486322)

[Starting Pre-loading Record DMX action: 95](#_Toc143486323)

[Stop Recording DMX: 95](#_Toc143486324)

[Ethernet 96](#_Toc143486325)

[Configuring Ethernet: 96](#_Toc143486326)

[Ethernet Command Status Reporting: 96](#_Toc143486327)

[Configuring FTP Password: 96](#_Toc143486328)

[Configuring UDP: 97](#_Toc143486329)

[IP Referencing: 97](#_Toc143486330)

[UDP Control: 98](#_Toc143486331)

[UDP Sending Data: 99](#_Toc143486332)

[Receiving Data using UDP Protocol: 100](#_Toc143486333)

[System Commands 102](#_Toc143486334)

[Re-Boot: 102](#_Toc143486335)

[Setting the Time Base: 102](#_Toc143486336)

[Cosmic variable commands: 103](#_Toc143486337)

[Start Playback: 103](#_Toc143486338)

[Query Playing Tracks: 104](#_Toc143486339)

[Query Track Status: 105](#_Toc143486340)

[Printing: 105](#_Toc143486341)

[Monitor Serial Port: 107](#_Toc143486342)

[Configure Serial Port: 107](#_Toc143486343)

[Null Character Substitution: 108](#_Toc143486344)

[Read Serial Port: 109](#_Toc143486345)

[Peek Serial Port: 110](#_Toc143486346)

[Silent Mode: 110](#_Toc143486347)

[Command feedback Enhancement: 110](#_Toc143486348)

[Install Operating System: 111](#_Toc143486349)

[Controlling Status LEDs: 111](#_Toc143486350)

[Configuring Triggers: 111](#_Toc143486351)

[Logging 112](#_Toc143486352)

[Log Activation: 112](#_Toc143486353)

[Log Deactivation: 112](#_Toc143486354)

[Log Update: 112](#_Toc143486355)

[Log File Clearing: 113](#_Toc143486356)

[Log Listing: 113](#_Toc143486357)

[Log Mode: 113](#_Toc143486358)

[Log file handling: 113](#_Toc143486359)

[Log timestamp: 113](#_Toc143486360)

[Sequence Programming Commands: 114](#_Toc143486361)

[Setting Sequence Context: 115](#_Toc143486362)

[Local Integer and String Variable Definition: 115](#_Toc143486363)

[Global Integer and String Variable Definition: 116](#_Toc143486364)

[Group Enumeration: 116](#_Toc143486365)

[Group Playback: 117](#_Toc143486366)

[Loading a Sequence File: 117](#_Toc143486367)

[Play Sequence Program: 118](#_Toc143486368)

[Stopping Running Sequences: 119](#_Toc143486369)

[Stepping through a sequence: 119](#_Toc143486370)

[Setting program breakpoints: 120](#_Toc143486371)

[Listing a Sequence: 121](#_Toc143486372)

[Program Flow 122](#_Toc143486373)

[Timer Delay: 122](#_Toc143486374)

[Branch immediate: 122](#_Toc143486375)

[Branch conditional construction: 123](#_Toc143486376)

[Branch on Selection construction: 123](#_Toc143486377)

[do loop construction: 124](#_Toc143486378)

[For loop construction: 125](#_Toc143486379)

[Event Programming: 125](#_Toc143486380)

[Event List Header and Footer: 125](#_Toc143486381)

[Event Command: 126](#_Toc143486382)

[Event Sensitivity: 127](#_Toc143486383)

[Cue List Commands: 128](#_Toc143486384)

[Cue relative: 129](#_Toc143486385)

[Cue Clear: 129](#_Toc143486386)

[Cue print: 129](#_Toc143486387)

[Timer Synchronization: 130](#_Toc143486388)

[Timer Sync Commands: 130](#_Toc143486389)

[Synchronizing Audio to the Master Timer 131](#_Toc143486390)

[Locking Sound Tracks: 131](#_Toc143486391)

[Custom Audio Processing 132](#_Toc143486392)

[System Description: 132](#_Toc143486393)

[Using the Default Model: 133](#_Toc143486394)

[Creating a Custom Design: 133](#_Toc143486395)

[Input DSP Processing 134](#_Toc143486396)

[Sigma Studio Setup: 135](#_Toc143486397)

[Ethernet Control of DSP Processing 136](#_Toc143486398)

[ASCII Table 138](#_Toc143486399)

[FAQ 140](#_Toc143486400)

[How do I read an input trigger and play a sound when the trigger occurs? 140](#_Toc143486401)

[How do I cross fade two sound files? 140](#_Toc143486402)

[How do I start a sound file 10 seconds into the file? 141](#_Toc143486403)

[How do I patch microphone inputs to analog outputs on the rPod10.2 and rPod8.4 . 141](#_Toc143486404)

[How do I play random shows using a single trigger? 144](#_Toc143486405)

[Strings:  Can any 8bit value be stored in a string variable? 145](#_Toc143486406)

[Comments?  What are the comment delimiters? 146](#_Toc143486407)

[Does VAL convert the string “HB” into  0x8A? (0x48 + 0x42) 146](#_Toc143486408)

[Will VAL convert the string “SC11104” into 0x18D? 146](#_Toc143486409)

[Is $a = $a + $b legal? 146](#_Toc143486410)

This manual provides a software reference for using the rPod10.2, rPod8.4 and MSII family (including the MSIIE) sound and motion control units. The scope of this document covers both direct and automated control of the units using the Sequence Programming Language. It’s broken down into four sections:

1. Operation and Use.
2. Quick reference guide.
3. Command description index
4. Frequently asked questions (FAQ)

This document will be updated as new features are added and software bugs are fixed. Please check with the SKE website or contact SKE directly for recent updates. For additional information on hardware setup and operation, please refer to the rPod10.2, rPod8.4, MSII, and MSIIE Hardware manuals. For simplicity, the references to MSII will include the MSIIE that is the MSII with Ethernet capability.

# Command Entry

Commands can be entered through the console or DTE serial ports using a standard terminal program. Pressing the Enter key completes a command. The Enter key generates the ‘carriage return’, line feed ASCII characters 13 (0x0d) and 10(0x0a). This is indicated with a ‘ ‘throughout the document. Note that all commands are case sensitive and followed by a <space> character. The default port characteristics are 115,200 baud, 8 bits, 1 stop, and no parity. The console port on-board connector is a DCE type, 9 pin female.

# Sequence Program Development

This section provides some general guidelines for developing and using sequences. Sequences are small programs, written in the SKE proprietary programming language and called by the unit’s operating system. Up to 16 programs can be run concurrently. They execute independent of one another but may share information through a set of user defined global variables.

The unit operating system has a built-in compiler that accepts source text files created with any standard text editor on a PC. Program source files are then transferred to a Compact Flash card and installed in the unit.

## Execution Modes:

There are three execution modes:

* 1. **Manual mode** - Started through command entered from a control port such as the Console or DTE interface.
  2. **Boot mode** - Automatically executed at power-up.
  3. **Program mode** – Started through command instructions issued from within another sequence program.

Generally speaking, the **Manual mode** is used during initial development or if the unit is being controlled by another device such as a PLC (programmable logic controller) or external computer. The commands “Ldseq” and “Playseq” are entered through the Console or DTE serial ports to load and execute the programs.

The **Boot mode** is used when the unit is going to operate as a stand-alone controller. After the program has been designed and tested, the unit can be configured to automatically execute the program at startup. The system recognizes a special file name, “startup.seq”, as the startup program. It will automatically load (Ldseq) and execute (Playseq) this file as part of the power-up process.

*Note: During development and testing of sequences, it is recommended that the startup file be removed from the system to prevent unexpected operation of the unit.*

The **Program mode** allows for the loading and playing of sequences from another executing sequence program. Including the “Ldseq” and “Playseq” commands in the main or startup sequence easily accomplishes this.

## Sequence Structure:

### Programs:

The sequence programming system allows for concurrent operation of 16 sequence programs. The programs are text files that reside on the CF card. Prior to execution, they may be pre-loaded and compiled to resident memory. Programs and subprograms may then be executed, stopped, or restarted without going through the load or compile process. A variety of debug facilities are available to the programmer to test and evaluate a program.

The main sequence is almost always designed to be a continuously looping program using of one of the loop mechanisms (for, do, or Goto). Subprograms generally execute to completion. However, subprograms can also be continuous. This is useful for polling status of inputs, tracking system health, or maintaining the state of the system.

### Group files:

Sequences that start, execute, and then end without interruption are group types. They are almost always called from a main program but may be executed independent of the main program. Group files are loaded, compiled, and executed at the time they are called. Debug features are not available for group sequences. Care should be taken when using this feature because timing may be affected by the extra file accesses. They are used to group common show segments and provide common library for initialization routines or other batch type operations.

### Events Lists:

Timer event lists can be embedded in sequence for fixed show control based on the internal timer. Only one event list may be active at a time. Events can be controlled by other programs, subprograms, group files or from the Console port. The rPod10.2 and rPod 8.4 allows for MIDI triggering and the rPod8.4 provides event triggering based on an external SMPTE generated clock when the SMPTE add-on board is attached.

### Cue Lists:

Another way to create Timer events is using Cue lists. Cue lists can be generated spontaneously or may be embedded in sequence files. They are used primarily for dynamic show control and, like event lists, are based on the internal timer. Reference groups are established to enable relative time line execution.

### Layout:

The system provides for a large variety of operands including constants, variables, timers, clocks, and hardware configuration and status. The following suggests a general structure for a typical main program.

Description Example

1. Variable definitions and Group enumerations Define x
2. Variable initialization x = 0
3. Port initialization Config Console = 9600
4. Control File Mounting Mount light1 on 1
5. Subprogram loading Ldseq Show1 on 2
6. Main loop start main Playseq 1
7. Program body x = x + 1
8. Main loop end Goto main
9. Event Lists Eventlist Eventlist1

### Key points:

1. Port initialization and subprogram loading should not be called, repeatedly, in the main loop.
2. The main loop start line consists of a jump label, “main” in the example, and may be followed any valid command.
3. The program body consists of one or more valid commands.
4. The main loop end is always a jump back to the main loop start line. Note that knowing the program line number is not necessary for any “Goto” instruction.
5. The main loop section can be alternately written with the “do” command.

|  |
| --- |
| forever = 1  main do  x = x + 1  while forever == 1 |

## Math and Logical Operations:

They provide a method for processing, calculating and evaluating system and program resources for the purpose of program control. Statements consist of a wide variety and combination of operands and operators. But, the system currently has a limit of one operator per line. An operation such as “x = x + y \* z” is INVALID. The four major types of operations are “assignment”, “if”, “while”, and “for” as shown in the Table of Operations.

The assignment group has three basic types; direct, logical, and arithmetic. The assignment function always stores operators from the right side of the “=” sign into the operand on the left side of the “=” sign. This operand must be writable variable, timer, or clock. Examples of these statements are:

|  |  |  |
| --- | --- | --- |
| Direct examples: | x = 25 |  *Store 25 into x* |
| y = z |  *Store the contents of z into y* |
| x = Trig1 |  *Store the state of external trigger 1 into x* |
|  | $a = x |  *Store the character value of x in $a* |
|  | x = $a |  *Store the ASCII value or sum-check of $a into x* |
| Logical examples: | y = ! x |  *Store the logical NOT of x into y* |
| x = x & 5 |  *Store the binary AND of x and 5 into y* |
| Increment/decrement | x ++ |  *Increments x. x = x + 1* |
|  | y = x -- |  *Set y=x then decrement x.* |
|  | y = ++ x |  *increment x then Set y=x.* |
| Arithmetic example: | x = y – 5 |  *Store the result of y minus 5 into x* |

## Decision Operations:

The method for making logical decisions in sequence programs is the “if” statement. It provides for conditional processing of two logical states, true (non-zero) and false (zero). Operations are constructed with a combination of operators and/or operands to the right of the “if”. If the operation is evaluated as true then the first block of commands before the corresponding “else” are executed. If it is false then the block of commands between the “else” and the “endif” are executed. See the Branch conditional construction section for a further description. Here are some examples of this operation type:

If example 1:

|  |
| --- |
| if Close1 == 1 ;*Was*‘*Trigger 1’ asserted (leading edge)?*  Play Flim on 1 ;*Yes, play the sound*  Dmx 1-1 on ;*Yes, assert the control output*  endif ;*End of if case* |

If example 2:

|  |
| --- |
| if x == 5 ;*Is the contents of variable x equal to 5?*  Play Flim on 1 ;*Yes, play the sound*  x = 0 ;*Clear x*  else ;*No, x is not equal to 5*  x = x + 1 ;*increment x*  endif ;*End of if case* |

There are two range checking operators that return true(1) or false(0). They are the “includes” and “excludes” operator. These operators use three operands “low range”, “high range” and “check value”. Here are some examples showing use of these operators:

|  |
| --- |
| Define xmin = 3 *;low range value*  Define xmax = 10 ;*high range value*  Define x *;check value*  if xmin xmax includes x ;*Is the contents of variable x between xmin and xmax?*  Print ”x>=3 and x<=10”  else ;*No, x is not equal to 5*  if 5 8 excludes x ;*Is the contents of x>8 or x<5*  Print “x is out of range”  endif  endif ;*End of if case* |

## Looping Operations:

The first of two looping methods is the “do” construct. With this construct, the block of commands between the “do” and the “while” is executed prior the evaluation of a logical statement. If the statement is evaluated to be true (non-zero) then processing continues at the beginning of the block of commands. Otherwise, processing continues at the line following the “while” statement. See the Branch conditional construction section for a further description. Here is an example of this operation form.

|  |
| --- |
| do ;*Start of loop*  count = count + 1 *;Increment a counter*  while Trig1 <> 0 ;*Continue looping while Trigger 1 is asserted* |

The other looping method is the “for” construct. Evaluation of the logical statement occurs prior to executing the block of commands between the “for” and “next” statements. The “incrementing” operand is compared with the “compare” operand. If the result is true (non-zero), then the block is executed. Otherwise, the program continues at the line after the corresponding “next” statement.

See the Branch conditional construction section for a further description. Here is an example of this operation form.

|  |
| --- |
| for I = 1 to 10 *;Counter I is initialized to 1. It is then compared with*  x = x << 1  *;the constant 10. Upon each subsequent pass, I is then*  next  *;incremented before the next evaluation.* |

## Constants:

The first operand type is the constant. Sequence programming constants are defined as any signed 32 bit integer value. Constants may also be strings that are enclosed by double quotes. Finally, you can use 32 bit floating point values.

## Common Variables:

Sequence programs can have nine types of variables; cosmic integers, public integers, global integers, local integers, and floating points. Cosmic strings, public strings global strings, and local strings. They may be created from within a sequence or through direct entry using the “Define” command. The Public variables are only available on the rPods. Public definitions are Global variables defined on both processors. Assignments made to a Public variable are implemented on both processors and are therefore visible to all sequences. Cosmic variables are saved to the compact flash card and will be automatically created and their values restored when the unit is re-powered. This Cosmic variable list is stored in the file is named “boot.ini” and may be edited with a text editor. Floating point variables are global to all sequences.

|  |  |
| --- | --- |
| Define x   Define Global y  Define $a   Define Global $b  Define Cosmic s1  Define Cosmic $c  Define Public z   Define Public $s   Define Float d | * Create a local variable x*  * Create a global variable y*  * Create a local string $a*  * Create a global string $b*  * Create a cosmic variable s1*  * Create a cosmic string $c*  * (rPod 8.4 only) Create a Public variable z*  * (rPod 8.4 only) Create a Public string $s*  * Create a global floating point variable d* |

Variable definitions are processed during compilation and not at program execution. Local variables are destructed when the sequence program ends or when loading over an existing sequence occurs. Global variables can’t be destructed except through re-booting the unit. There are a total of 64 global, public, floating point, and cosmic integers that can be shared by the 16 available sequences. Each sequence has a total of 64 local variables which are not shared between sequences.

String variable are always preceded by a ‘$’ symbol. The maximum string length is 256 characters. Also, only 64 local strings per sequence and 64 global/public/cosmic strings can be defined.

Here are some suggestions for using variables:

1. Variable names may consist of any combination of numbers or letters and may not include spaces or ASCII control characters. The underscore character may be used to separate words. Ex. Counter\_1, Left\_eye, $my\_string, etc.
2. Variables may not use names that are on the reserved words list.
3. Variables do not need to be defined before they are used. It’s a good idea to define them at the beginning of the program but its not required.
4. Local variables are only available to the sequence that created it.
5. Global/Public/Cosmic variables are available to all sequences.
6. Global, Public,Cosmic and local variables should not be given the same name. Unexpected operation may occur due to variable shadowing. Multiple sequences that operate concurrently may have local variables with the same name.

Variables may also be viewed with the “Print” command and modified through assignment from the command entry mode. See the “Update, “Define” and “Context” commands for more information on cosmic variables.

## Trigger Variables:

Trigger variables act like Public variables but use a numerical reference instead of a variable reference. Because of this, the definition form and access has a different set of rules. The use forms are as follows:

|  |  |
| --- | --- |
| Define ETRIG 2   Define ETRIG 300 = 20  ETRIG 50 = 5  If ETRIG 2 == 1   ETRIG 10 = 50 | * Creates trigger 2 without initialization*  * Creates trigger 300 and initializes it to 20*  * Sets the trigger 50 to 5*  * Conditional: is trigger 2 equal to 1*  * Assigns the value 50 to trigger 10* |

Trigger variable rules:

1. There can be a total of 256 trigger variables defined.
2. The range of trigger numbers is 1 to 65535. ETRIG1 – ETRIG65535.
3. All trigger values are signed 32 bit numbers. From −(231) to 231– 1.
4. Like Public variables, triggers are common to both processors and all sequences. They should only be defined in one sequence and one processor. Any subsequent definitions are redundant and will be ignored.
5. All triggers must be allocated with a Define statement.\

## Passing Processor Data (rPod8.4 only):

Because the rPod 8.4 has dual processors, it has two separate and distinct scripting engines. All global and local variables are also separate from each other. Public variables do not need to be passed as this happens automatically. The pass command facilitates variable assignment in the other processor and pass operator enables variable retrieval from the other processor. Examples are:

Direct examples:

|  |  |
| --- | --- |
| Pass x to y   Pass 25 to y   Pass $b to $a   Pass “Hello” to $a   *x =* Pass y   $a *=* Pass $b  | * Store variable x on Ps to variable y on Pd*  * Store constant 25 to y on Pd*  * Store string $b on Ps to string $a on Pd*  * Store constant “Hello”on Ps to string $a on Pd*  * Store variable y on Pd to variable x on Ps*  * Store string $b on Pd to string $a on Ps* |

*Note: Ps = Source Processor, Pd = Destination processor*

# String Operations:

The functions in this section provide a means to manipulate local and global strings. The strings are read in through one of the serial ports from another device or generated inside a sequence program. The functions can be called by entering them through a serial port, executed in a sequence program, or called in an event list. The system currently has a limit of one operator per line. An operation such as “x = Left a 3 + Right a 3” is INVALID. The following is a list of the functions:

### String Concatenation:

*Syntax- “destination string” =* *“string1” + “string2”* 

*Parameters* --*“destination string”, “string1”, “string2” =* a defined local or global string variable, string constant or system string variable.

This operation concatenates two ASCII strings and stores the result in the destination string. To append a string type:

|  |  |
| --- | --- |
| $a = $b + $c   $a = “String = ” + $b |  Example 1:*variable string version*   Example 2:*constant string version* |

### String Comparison:

*Syntax-*  *“string1” == “string2”* 

*“string1” <> “string2”* 

*Parameters* -- *“string1”, “string2” =* a defined local or global string variable, string constant or system string variable.

This operation compares two ASCII strings. The operation returns a one when the compare operation is true. A zero is returned when it is false. To compare a string type:

|  |  |
| --- | --- |
| if $b == $c   x = $a <> “Foobar”  |  Example 1:*variable string version*   Example 2:*constant string version* |

### String to Number:

*Syntax- “operand” =* Val *“string”* 

*Parameters* --*“ operand” =* local or global integer variable.

*“string” =* a defined local or global string variable, string constant or system string variable.

This function converts an ASCII string to its equivalent numeric integer value. To convert a string type:

|  |  |
| --- | --- |
| x = Val $a   x = Val “32”   x = Val “0x10”   x = Val Console  |  Example 1: *variable string version*   Example 2: *constant string version*   Example 2: *Hexadecimal string version*   Example 4: *string received on the Console port* |

### Number to String:

*Syntax- “string” =* Str *“operand”* 

*Parameters* --*“ operand” =* local or global integer variable.

*“string” =* a defined local or global string variable, string constant or system string variable.

This function converts an integer variable or constant to an ASCII string. To convert a number to an ASCII string type:

|  |  |
| --- | --- |
| $a = Str 10   $a = Str count  |  Example 1:*constant value version*   Example 2:*variable value version* |

### String Length:

*Syntax- “operand” =* Len *“string”* 

*Parameters* --*“ operand” =* local or global integer variable.

*“string” =* a valid local or global string variable, string constant or system string variable.

The Len function calculates the length of the specified string:

|  |
| --- |
| x = Len $a  *; variable string version* |

### Search String:

*Syntax- “operand” =* Instring *“string1” “string2”* 

*Parameters* --*“ operand” =* local or global integer variable.

*“string1” =* source string. Any defined local or global string variable.

*“string2” =* search string. Any defined local or global string variable.

Searches string1 for the first occurrence of string2 and returns the position to operand1. 0 = not found.. To search a string type:

|  |
| --- |
| $a = “Say Hello”  x = Instring $a “Hello”  Print x  5 |

### Left String:

*Syntax- “string1” =* Left *“string2” “count”* 

*“operand” =* Left *“string2” “count”* 

*Parameters* --*“ count” =* number of characters to return.

*“string1” =* destination string. Any defined local or global string variable.

*“string2” =* source string. Any defined local or global string variable.

*“operand” =* destination integer. Any defined local or global variable.

This function extracts the left “count” characters from “string2” to “string1”. To get the left four characters of string $a to string $b type:

|  |
| --- |
| $a = “Fire Truck”  $b = Left $a 4  Print $b  Fire |

When the destination is an integer variable, this function sums the left “count” ascii values of the characters in “string2”. In this example, x is assigned the ASCII value of the “a” character:

|  |
| --- |
| $a = “ab”  x = Left $a 1  Print x  97 |

### Right String:

*Syntax- “string1” =* Right *“string2” “count”* 

*“operand” =* Right *“string2” “count”* 

*Parameters* --*“ count” =* number of characters to return.

*“string1” =* destination string. Any defined local or global string variable.

*“string2” =* source string. Any defined local or global string variable.

*“operand” =* destination integer. Any defined local or global variable.

This function extracts the right “count” characters from “string2” to “string1”. To get the right 8 characters of string $a to string $b type:

|  |
| --- |
| $b = Right $a 8 |

When the destination is an integer variable, this function sums the right “count” ascii values of the characters in “string2”. In this example, x is assigned the ASCII value of the “b” character:

|  |
| --- |
| $a = “ab”  x = Right $a 1  Print x  98 |

### Mid String:

*Syntax- “string1” =* Mid *“string2” “start” “count”* 

*“operand” =* Mid *“string2” “start” “count”* 

*Parameters* --*“ start” =* character position to start the extraction.

*“count” =* number of characters to return.

*“string1” =* destination string. Any defined local or global string variable.

*“string2” =* source string. Any defined local or global string variable.

*“operand” =* destination integer. Any defined local or global variable.

This function extracts the “count” characters starting at “start” from string2 to string1. To get the 10 characters of string $a starting at the fifth character type:

|  |
| --- |
| $b = Mid $a 5 10 |

When the destination is an integer variable, this function sums the middle count ascii values of the characters in string2 starting at start. In this example, x is assigned the sum of the ASCII value of the “a” and “b” characters:

|  |
| --- |
| $a = “1ab2”  x = Mid $a 2 2  Print x  195 |

# Sound Channel Information;

## Sound Channel Output Information

The functions in this section provide a means to access sound channel output setting for status and reporting. The functions can be called by entering them through a serial port, executed in a sequence program, or called in an event list. The system currently has a limit of one operator per line.

*Syntax- “operand” =* Cvol *“ Channel #”*

*“operand2” =* Cvold *“Channel #”*

*Parameters* -- *“operand” =* a destination integer. Any defined local or global variable.

*“operand2” =* a destination floating point or integer variable. Any defined local or global variable.

*“Channel #” =* Channel number may specified by a constant, local or global variable.

The ‘Cvol’ operation obtains channel volume setting as either 0-127 where 0=off and 127 = max volume. The ‘Cvold’ operation obtains the channel volume setting in decibels ranging from -90 dB – 0 dB. Both operations store the result in the destination variable operand.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x = Cvol 1   |  |  |  | | --- | --- | --- | | f = Cvold 3 |  |  | | x will contain the volume setting for channel 1 from 0 to 127  f will contain the volume setting for channel 3 from -90 to 0db |

## Sound Channel Input Gain Information:

The functions in this section provide a means to access sound channel input gain for status and reporting. The functions can be called by entering them through a serial port, executed in a sequence program, or called in an event list. The system currently has a limit of one operator per line.

*Syntax- “operand” =* Ivol *“ Channel”*

*“operand2” =* Ivold *“Channel ”*

*Parameters* -- *“operand” =* a destination integer. Any defined local or global variable.

*“operand2” =* a destination floating point or integer variable. Any defined local or global variable.

*“Channel” =* Channel is either MICL,MICR,LINL, or LINR.

These operations get the input sound gain setting for the specified channel. The Ivol operation obtains channel input gain setting as either 0-127 where 0=min gain and 127 = max gain. The ‘Ivold’ operation obtains the input channel gain setting in decibels ranging from -90 dB – 0 dB . Both operations store the result in the destination variable operand.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x = Ivol LINL   |  |  |  | | --- | --- | --- | | f = Ivold MICR |  |  | | x will contain the volume setting for left line in from 0 to 127  f will contain the volume setting for right mic in from -90 to 0db |

# Sound Track Operations:

The functions in this section provide a means to access track information for status and reporting. The functions can be called by entering them through a serial port, executed in a sequence program, or called in an event list. The system currently has a limit of one operator per line.

The commands use numeric and text sub-elements to identify specific sound track parameters. The following is an enumerated list of these parameters:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Format Index | Descriptor | Numeric | String | Description |
| 0 | 'label' |  |  | format text |
| 1 | pitch |  |  | track number |
| 2 | name |  |  | track name |
| 3 | channel |  |  | channel number |
| 4 | status |  |  | playback status |
| 5 | percent |  |  | percent complete |
| 6 | length(1-5) |  |  | sound length |
| 7 | tvol |  |  | track volume |
| 8 | atk |  |  | attack time |
| 9 | dek |  |  | decay time |
| 10 | dvol |  |  | duck volume |
| 11 | form |  |  | mono or stereo file |
| 12 | mode |  |  | mono or stereo playback |
| 13 | sample |  |  | sound sample rate |
| 14 | loopcount |  |  | audio loop playback count |
| 15 | tvold |  |  | track volume in decibels |
| 16 | dvold |  |  | duck volume in decibels |
| 17 | datk |  |  | duck attack time |
| 18 | ddek |  |  | duck decay time |

## Single Track Information:

*Syntax- “operand” =* Tvar *“Track #” “Fmt index/descriptor”*

*“string” =* Tvar *“Track #” “Fmt index/descriptor”*

*Parameters* -- *“operand” =* a destination integer. Any defined local or global variable.

*“string” =* a defined local or global string variable, string constant or system string variable.

*“track #” =* Track number may be specified by a constant, local or global variable.

*“Fmt index” =* SoundTrack parameter index specified by either the descriptor label or the numeric format index. This can be a constant, local variable or global variable.

This operation obtains track information specified by the format index number and stores the result in the destination variable or string.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x = Tvar 2 13    |  |  |  | | --- | --- | --- | | $a = Tvar 3 name |  |  | | x will contain the numeric sample rate of track 2  $a will contain the name of the sound file assigned to track 3 |

## Single Track Progress:

*Syntax- “operand” =* Progress *“Track #” “mode”*

*Parameters* -- *“operand” =* a destination integer. Any defined local or global variable.

*“track #” =* Track number may be specified by a constant, local or global variable.

*“mode” =* SoundTrack mode where:

*mode=0 - Time length of sound*

*mode=1 - Time Played so far*

*mode=2 - Time remaining to play*

This can be a constant, local variable or global variable.

This operation obtains track information specified by the mode index number and stores the result in the destination variable.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x = Progress 2 0    |  |  |  | | --- | --- | --- | | y = Progress 3 1 |  |  | | x will contain the time length of the sound on track 2  y will contain the time played so far on track 3 |

## 

## Track Lists:

*Syntax- “string” =* Tlist *“Format string” “Track list string”*

*Parameters* -- *“string” =* a defined local or global string variable, string constant or system string variable.

*“format string” =* a defined local or global string variable, string constant defining the output report format.

*“track list string” =* a defined local or global string variable, string constant specifying the tracks to include in the report list.

This operation creates a report based on the “format string” for all tracks specified by in the “track list string”. The “format string” is a comma-delimited list of sound track parameters in the table above and/or labels. Labels are defined as any set text characters enclosed by single quote characters. These may be considered as sub-strings. Labels are used to make the list report more readable.

The following “format string” example has two labels. The first is ‘Track #’ and the second is ‘ is playing on ch# ‘. There are also two report variables, track and channel. The Tlist command will replace track and channel variables with the current assigned values for each track called the Track list string. In this example, they are tracks 1 and 3. Here is the “format string”:

|  |  |
| --- | --- |
| $b = "'Track #',track,' is playing on ch# ' ,channel " |  |

The track list string can be a constant or string variable. Next is the function call:

|  |  |
| --- | --- |
| $a = Tlist $b “1,3”  ; or  $a = Tlist $b $c ; where $c = “1,3” |  |

$a will contain the track list report. A carriage return and line feed character is place between each track line. Printing $a results in the following output:

|  |  |
| --- | --- |
| Print $a   Track #1 is playing on ch# 1  Track #3 is playing on ch# 3 |  |

The format strings adhere to the “C” language string convention. Therefore, care must be used when including some of the special string format characters like ‘%’. In this case, use two consecutive characters, ie. “%%” to produce a single “%” character. Also, note that the track list must be contained a string. $a = Tlist $b 1,3 is INVALID.

# System variables:

These special variables are pre-defined by the system and may be referred to in sequences. These variables names are reserved and may not be redefined using the “Define” command.

## Audio and Control Status:

The variables ?Pn, ?Tn, and ?Dn return the current playback status of audio and control files. Their return values are as follows:

|  |  |
| --- | --- |
| ?Pn  | 0 = Audio Track n is not playing. |
| 1 = Audio Track n is playing. |
| 2 = Audio Track n is looping. |
| 3 = Audio Track n is pre-loaded in buffer |
| 4 = Audio Track n is waiting for buffer closure |
|  | 5 = Audio Track n is paused |

|  |  |
| --- | --- |
| ?Tn  | Audio Track n completion where 0 = start and 100 = end. |

|  |  |
| --- | --- |
| ?Dn  | 0 = DMX Track n is not playing. |
| 1 = DMX Track n is playing. |
| 2 = DMX Track n is looping. |
| 3 = DMX Track n is pre-loaded in buffer |
| 4 = DMX Track n is waiting for buffer closure |

The variable “Source” determines the Codec source for the rPod8.4 only. For more information, refer to the rPod 8.4 Considerations section.

Source -> 0 = Split Channels 1-4, 9 assigned to P1. Channels 5-8, 10 to P2

1 = P1 Channels 1-10 controlled by processor 1/CF1

2 = P2 Channels 1-10 controlled by processor 2/CF2

## Accessing Clip Parameters:

The variables in this section provide a means to access and/or change information for Clips that have already been loaded to the unit’s memory from a CF card using “Clip” or “Ldclip” command. Access is available from a command interface, sequence program, or event list. Operations that modify Clip parameters do not affect the file located on the CF card. Therefore, the changes are temporary and will be restored if the Clip is re-loaded from the original file.

There are two types of variables that can be used. The first variable references Clip information. The only parameter that must be provided is the Clip Name. This is followed by a ‘.’ delimiter and then the “Clip Parameter” to be referenced. The example below shows how to retrieve the wave sound file name:

|  |  |
| --- | --- |
| Define $n  Ldclip cApollo ;Load the clip cApollo.clip  $n = Apollo.audiofilename ;retrieve the wave file name |  |

The following table shows a list of available parameters. The Clip Name and Audio File Name are Strings and can only be assigned to string variables. The rest of the parameter can only be assigned to numeric variables and not strings.

|  |  |  |  |
| --- | --- | --- | --- |
| Clip Parameter | Numeric | String | Description |
| .name  .id |    |    | Clip Name (Main reference for clip)  Clip Identification number |
| .audiofilename |  |  | Name of the Audio File |
| .channels |  |  | Number of channels in the Audio File |
| .trackcount |  |  | Number of associated tracks |

The second type of variable references Track information. Because Clips can have multiple tracks, both the Clip name and the Track name must be known. A ‘.’ is used to separate “Clip Name” and “Track Name”. This is followed by another ‘.’ and then the “Track Parameter to be referenced. The following example shows how to temporarily change the pre-determined track volume:

|  |  |
| --- | --- |
| Ldclip cApollo ;Load the clip cApollo.clip  Apollo.apollo1.tvol = 80 ;change the track volume to 80  Apollo.apollo1.tvolflg = 1 ;Enable the track change |  |

Note that in the example above, the track change will not occur until the next “Start C cApollo” command is issued. Since a “Start” command does not re-load a Clip, it will not change the Clip file in any way. If the Clip is loaded again, the previous track number and track volume flag will be restored. To change the track volume of an actively playing sound enter:

|  |  |
| --- | --- |
| Tvol Apollo.apollo1.track = 80 ;Change an active clips track volume |  |

The following are identifiers for accessing Track Parameters in a Clip:

|  |  |  |  |
| --- | --- | --- | --- |
| Track Parameter | Numeric | String | Description |
| .name  .id  .track  .tvol  .atk  .dek  .dvol  .channel  .cvol  .crosstrack  .startoffset  .duration  .tvolflg  .atkflg  .dekflg  .dvolflg  .stereo  .startmode  .startoffsetfmt  .startoffsetflg  .durationfmt  .crossflg  .endmode |    |    | Track name  Track identification number  Track number  Track sound volume  Track sound volume attack time  Track sound volume decay time  Track duck volume  Channel assignment  Channel volume setting  Crossfade track to play  Start time offset  Time length playback duration limit  Track volume change enable flag  Track attack time enable flag  Track decay time enable flag  Track duck volume enable flag  Track playback mode  Start behavior  Start time format  Start time offset enable flag  Duration time format  Crossfade enable flag  End behavior |
|  |  |
|  |  |
|  |  |
|                                    |                                    |

## System Information:

The system version, manufacturing date and Ethernet MAC address can be obtained by accessing the system variable INFO. Printing INFO give the following sample response:

a:>Print INFO

Revision 1.56, File:rpodP1a056 26JUN2014 44.1K/16bit

a:>cd b

b:>Print INFO

Revision 1.55, File:rpodP2a055 10JUN2014 44.1K/16bit

MFG: Fri 05/16/14 03:54:41 PM Serial #13351307290023 F4:1E:26:04:00:44

The ?DPM variable maintains a count of the dual port memory data transfer misses. After reading this variable the error count is automatically reset to zero.

## Ethernet:

There are three categories of status variables that can be accessed through a sequence program or directly from the Console or DTE serial ports. The first set of status variables report the Ethernet system characteristics. They are:

IPADD = IP address of the unit. This can be assigned to a string or number variable.

IPPORT = IP Port of the unit

IPMASK = IP subnet mask of the unit.

UDP = String variable containing the contents of the active UDP packet.

INFO = Unit version, mfg date, and MAC address.

SIGPORT = Port assignment for Sigma DSP programming over ethernet

The second set of variables are used to facilitate UDP data exchanges. The following variables are used with UDP Sendv and Recv commands:

?ETYPE = User defined configuration of variable list. 0 - 65535

?EORIG = Source bind number from 1 to 255.

?ECOUNT = Number of string or number variables received

?EPKT = Packet number from the sender. 0 to 65535

?ENET = UDP buffer ready flag. 0=no data, 1=Receive variables, 2=raw UDP data?

The third, EENABLE, enables or disables UDP data reception. Here is an example of its use:

|  |
| --- |
| if EENABLE == 0 ;*is UDP reception disabled?*  EENABLE = 1 ;*yes, then enable UDP reception*  *else*  EENABLE = 0 ;*no, then disable UDP reception*  endif |

## Serial Communications:

Two serial communication variables, “?Console”, “?DTE”,?DMX, are available to flag Read completion of the Console, DTE or DMX serial ports. When the port termination is on, the read only flags return 1 if the Read operation has completed and 0 if it has not. The example provided next illustrates its use:

|  |
| --- |
| M0 Read DTE ;*Commands DTE port to look for a string*  if ?DTE == 1 ;*Checks to see if a string has been received*  if DTE == “Play” ;*Compares the received string to “Play”*  Play Flim on 1 ;*if it is then it plays the sound file “Flim”*  endif  endif  Goto M0 |

In the example above, the string variable “DTE” is used. It holds the input string captured as a result of a “Read DTE” command. Likewise, the variable “Console” and DMX will hold the string captured from a “Read Console” and Read DMX command. Notice that it is compared to the user-defined string “Play”. The DTE, Console, DMX and user-defined strings can be used by the “Print” command. This is very useful tool for debugging a sequence program.

If port termination is off then the “?DTE”, “?Console”, and “?DMX” variables return the number of character in the receive buffer. Issuing a “Read Console”, “Read DTE”, or “Read DMX” will read all the characters in the buffer. Read Console 5 will read up to 5 characters from the Console buffer.

## Triggers:

Four system variables can be used to determine the status of the Input Trigger Port. The Trig, Close, and Open variables are subject to the de-bounce settings. See the Configuring Triggers section.

|  |  |
| --- | --- |
| Variable name | Description |
| Inputs  Trig1  Close1  Open1 | *Trigger Word*  *Trigger Input*  *Trigger Assertion*  *Trigger Removal* |

The “Inputs” variable returns the current state of all inputs as a binary weighted code. Bit 0 is assigned to Input 1 and bit 7 to Input 8. A “1” indicates that the trigger is asserted and a “0” is no assertion.

|  |  |
| --- | --- |
| Trigger # | Weighting |
| 1  2  3  4  5  6  7  8 | 1  2  4  8  16  32  64  128 |

For example, the following shows trigger 1, 2 and 4 are asserted:

|  |  |
| --- | --- |
| x = Inputs   Print x   *11* |  *Read the input triggers*   *Print the result*   *Trigger 1,2 & 4 are asserted* |

The “Trig” variable returns the current state of a single input. A “1” indicates that the trigger is asserted and a “0” that it is not asserted. For example, the following reads the current state of trigger 5:

|  |  |
| --- | --- |
| x = Trig5   Print x   *1* |  *Read trigger 5*   *Print the result*   *Trigger 5 is asserted* |

The “Close” and “Open” variables are “sticky” variables and can be used for edge trigger detection. The system logs all trigger transitions until the next time the associated variable is polled. Therefore, they should be included as part of the main polling loop. The “Close” command detects the initial assertion of a trigger and the “Open” command detects the removal of the trigger. The variables read back a “1” when an event is detected and a “0” when no event was detected. Reading these variables will re-enable edge detection processing and all subsequent reads will return “0” until another trigger is found. Here is an example:

|  |  |
| --- | --- |
| x = Close2   Print x   *1*  x = Close2   Print x   *0* |  *Read trigger 2*   *Print the result*   *Trigger 2 edge was detected*   *Read trigger 2*   *Print the result*   *Trigger 2 edge was not detected* |

## Expansion Triggers:

Triggering can be expanded by using one of the external inputs as a serial input connection. The transmitting device can be a PLC or other smart device. This connection accepts a serial word from 5 to 10 bits and can create up to 1024 unique triggers. Command examples are shown below:

Config TEXP <option> <parameter>

**Command Description**

Config TEXP on Enable serial trigger expansion mode

Config TEXP off disable serial trigger expansion mode

Config TEXP input1 n First Trigger input to use (1-8 ms3) and (1 - 12 rpod)

Config TEXP input2 n Second Trigger (mirror trigger) input to use (1-8 ms3) and (1 - 12 rpod)

Config TEXP bittime n Bit period in ms (1-200)

Config TEXP bits n Number of bits in serial code (4-10)

Config TEXP idle low/high Rest logic state when not transmitting

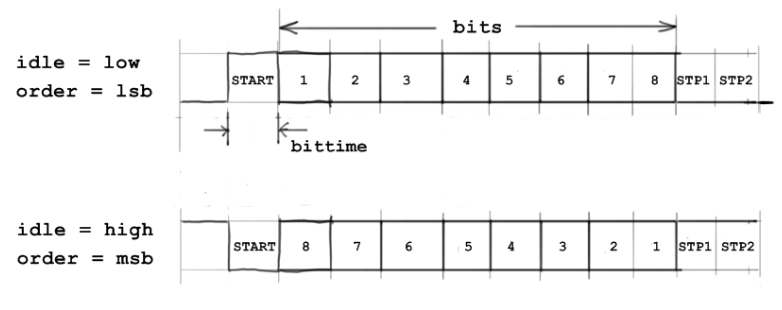
Config TEXP polarity low/high logic low state: low -> 0=low, 1=high, high -> 0=high, 1=low

Config TEXP stops n Number of stop bits (0-2)

Config TEXP order lsb/msb Transmit order: msb-most significant bit first, lsb-least significant bit first

**Defaults:**

TEXP is off, input1 =1, input2 = 1, bit time=50ms, bits=5, idle=low, polarity=low, stop=0 (none)



There are two system read/write variables that determine if a serial trigger has been detected and the value of the trigger received. These variables are:

XTRDY : Trigger received flag. 0 = not ready, 1 = ready

XTRDY should be cleared to 0 after ready is detected using XTRDY = 0.

XTRIG : This is the trigger number. It is valid after XTRDY has been read as a 1.

|  |
| --- |
| ;Trigger expansion example  Define trig  Config bits 8 ;Set the input word size to 8 bits  Config TEXP on ;Enable Trigger expansion mode  Main  if XTRDY == 1 ;Trigger received?  XTRDY = 0 ;Yes, clear trigger ready flag  trig = XTRIG ;Assign trigger to variable trig  if trig == 1 ;Was the trigger a 1?  Play sound1 on 1 T1 ;Yes, play something  endif  if trig == 2 ;Was the trigger a 2?  Play sound2 on 1 T1 ;Yes, play something else  endif  endif  Goto Main ;End of main loop |

## DMX Input:

The system variable “Dmxin “ is used to access any channel from 1 to 512. This read only variable has several optional that, can detect specific changes in state of a channel. The variable can be assigned to a variable. The table below shows the four options

|  |  |
| --- | --- |
| Variable name | Description |
| value  nonzero  zero  change | *Reads the instantaneous value*  *Detects an edge change from zero to nonzero*  *Detects an edge change from nonzero to zero*  *Detects any change in value* |

The “nonzero”, “zero”, and “change” options compare the previous read of a channel to its current value. This requires a baseline read to detect the change. The example below illustrates use of this variable.

|  |
| --- |
| Define x = 0 |
| x = Dmxin 1 value ;Read DMX ch 1 to establish baseline |
| x = Dmxin 2 value ;Read DMX ch 2 to establish baseline |
| x = Dmxin 3 value ;Read DMX ch 3 to establish baseline |
| J1 |
| x = Dmxin 1 nonzero ;Look for an edge change from zero to nonzero |
| if x <> 0 ;x returns 0 for no change and 1 for a change |
| Play flim on 1 T1 |
| endif |
| x = Dmxin 2 zero ;Look for an edge change from nonzero to zero |
| if x <> 0 ;x returns 0 for no change and 1 for a change |
| Play Apollo 13 on 1 T2 |
| endif |
| x = Dmxin 3 change ;Look for any change in value on DMX channel 3 |
| if x <> 0 ;x returns 0 for no change and 1 for a change |
| Play Sqam on 1 T3 |
| endif |
| x = Dmxin 4 ;read the current value of DMX channel 4 |
| if x == 128 ;is DMX channel 4 = 128? (mid range) |
| Play bell on 1 T4 |
| endif |
| Goto J1 |

## Compact Flash and File Status:

The status of a File on the CF card can be determined using the “?FILE” function. This function requires a file reference that may be a quote-delimited string, Console, or DTE. The function returns 0 if the file is not on the card, 1 if the file is there and it’s a sound file, and 2 if it’s a DMX file. The following shows an example of its use.

|  |
| --- |
| J1 Read Console ;*Put Console port in read mode*  if ?Console == 1 ;*Has string has been received;*  if ?FILE Console == 1 ;*Is the text from the console an*  *; actual sound file?*  Play Console on 1 ;*if it is then it plays the sound*  *;file from console*  else  if ?FILE “Flim” == 1 ;*Does the sound file Flim exist?*  Play Flim on 1 ;*yes, Play the file*  endif  endif  endif  goto J1 |

The system variables ?CF1 and ?CF2 return the status of the compact flash card. The read only variables can be used within a sequence program to tell if the card is installed or not installed. A 0 is returned if the card is missing and 1 if the card is installed. ?CF1 is used for the Mini Sam and rPod. ?CF2 is only applies to the rPod.

## Random Numbers:

*Syntax- “value” =* Rand *“start” “end”* 

*Parameters* --*“ start” =* start of random number range.

*“end” =* end of random number range.

The “Rand” variable returns a pseudo-random integral number in the range 0 to 32767. When the unit is powered up, the random generator is “seeded” using the current date and time from the real-time clock. The following example returns a random number between 1 and 10.

|  |
| --- |
| index = Rand 1 10  |

## Real Time Clock:

The real time clock (RTC) can be set and read using the “CLOCK” function. To set the RTC, enter the following command using the current date and time:

|  |
| --- |
| CLOCK = Jun 4, 2008 11:24:00  |

The format for entry is *“MMM DD, YYYY hh:mm:ss”* where:

1. *MMM* = 3 character month abbreviation. *{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}*
2. *DD* = 2 character day of the month number. *{01-31}*
3. *YYYY* = 4 character year number. *{1900-3000}*
4. *hh* = 2 character hour number. *{00-23 military format}*
5. *mm* = 2 character minute number. *{00-59}*
6. *ss* = 2 character seconds number. *{00-59}*

To print the current time, type the following command:

|  |  |
| --- | --- |
| Print CLOCK   *06/04/08 11:26:01 AM* |  *Displayed value from the unit* |

Adding any of the following extensions can individually access the date and time sub-fields:

1. CLOCK.MONTHmonth number. *{1=January … 12= December}*
2. CLOCK.DATE day of the month number. *{1-31}*
3. CLOCK.YEARyear number. *{1900-3000}*
4. CLOCK.HOUR hour number. *{00-23 military format}*
5. CLOCK.MINUTE minute number. *{00-59}*
6. CLOCK.SECONDSseconds number. *{00-59}*
7. CLOCK.DAYday of the week. *{1=Sunday … 7=Saturday}*
8. CLOCK.NUMBER numerical representation of CLOCK

Clock example 1: *(Entered through the console)*

|  |  |
| --- | --- |
| A:>Define x   A:>x = CLOCK.DAY   A:>Print x   *4* |  *Displayed value from the unit on a Wednesday* |

Clock example 2: (*Sequence program)*

The following example sequence will chime the sound file “Clarion” every hour on the hour:

|  |
| --- |
| Main   if CLOCK.MINUTE == 0   if CLOCK.SECONDS == 0   Play Clarion on 1   do   while ?P1 <> 0   endif   endif   Goto Main  |

## System Timer:

The timer is an incrementing time of day counter that is based on the system clock, MIDI, or SMPTE source. The timer operates independent of the real time clock (RTC) and is disabled and set to zero at startup. The timer resolution is 1/30th of a second or one frame when configured for frames, 1/1000th of a second when configured for ms, and 1/10 second for mixed. It is also common to all sequences. The constant format for the timer is *“hh:mm:ss.ff”or “hh:mm:ss.mmm”* where:

1. *hh* = 2 character hour number. *{00-23 military format}*
2. *mm* = 2 character minute number. *{00-59}*
3. *ss* = 2 character seconds number. *{00-59}*
4. *ff* = 2 character frames number. *{00-29}*
5. *mmm =* 3 character milliseconds. *{000-999}*

The system timer can be accessed using the “TIMER” system variable. To set the timer to five minutes thirty seconds and 20 frames, enter the following command:

|  |
| --- |
| TIMER = 5:30.20 *or*  TIMER = 5:30.660  |

Specific I/O references for source and destination ports are as follows:

|  |  |
| --- | --- |
| MIDI1  MIDI2  SMPTE | On-board MIDI port.  RASR daughter board MIDI port. (rPod8.4 only)  RASR daughter board SMPTE output. (rPod8.4 only) |

The timer can be synchronized to “LOCAL”, “MIDI1”, “MIDI2” or “SMPTE” using the system variable “Tsync”. The system default is “LOCAL”. To synch the timer to “SMPTE” type:

|  |
| --- |
| Tsync = SMPTE  |

The timer destination can be set to “NONE”, “MIDI1”, “MIDI2”, or “SMPTE” using the system variable “Tdest”. The system default is “NONE”.

To set the time destination to midi time code type:

|  |
| --- |
| Tdest = MIDI1 |

The timer has five modes; “on”, “off”, “hold”, “stop”, and “resume”.

1. “on” : The “on” mode sets the timer to increment based on the system clock, MIDI clock or, in the case of the rPod8.4, an external SMPTE reference. A “timer offset” parameter may be specified after the “on” flag. When a “timer offset” is specified, all Events are referenced from the timer offset.
2. “off”: This command stops the system timer and any event cueing. Changing the timer when it is “off” will cause a re-calculation of the event list.
3. “hold”: This command stops the system timer but allows for event cueing that may occur from manually changing the timer value. This allows for manual operation of the timer.
4. “stop”: Like the off command, this command stops the system timer and any event cueing. It also pauses all playing sounds and DMX control files. This is the “General Stop” command.
5. “resume”: This command performs the same function as the on command but also resumes playback of all paused sounds and DMX control files.

The following are examples of setting the timer mode:

1. *TIMER on ;start timer running*
2. *TIMER off ;halt timer running (default)*
3. *TIMER hold ;hold timer for manual operation*
4. *TIMER stop ;pause timer, sound, and DMX playback.*
5. *TIMER resume ;resume timer, sound, and DMX playback.*

The timer elements can be individually accessed for printing or variable assignment. When the timer source is set to the “LOCAL”, it may be changed by using the following:

|  |  |  |
| --- | --- | --- |
| **Timer Examples** | **Access** | **Assign** |
| HOUR | y = TIMER.HOUR | TIMER.HOUR = 12 |
| MINUTE | Print TIMER.MINUTE | TIMER.MINUTE = 30 |
| SECONDS | if TIMER.SECONDS == 20 | TIMER.SECONDS = x |
| FRAMES | while TIMER.FRAMES == 20 | TIMER.FRAMES = 0 |
| CLOCK | - | TIMER = CLOCK |

## General Purpose Timers:

Eight general-purpose timers, Tmr1 – Tmr8, are available. Just like the system timer, they use an incrementing time of day counter that is based on the system clock. However, each timer operates independent of the system timer, real time clock (RTC), or each other. They are disabled and set to zero at startup. The timer resolution is 1/30th of a second or one frame when configured for frames , 1/1000th of a second when configured for ms and 1/10 second for mixed. They are also common to all sequences. The constant format for the timer is *“hh:mm:ss.ff” or “hh:mm:ss.mmm”* where:

1. *hh* = 2 character hour number. *{00-23 military format}*
2. *mm* = 2 character minute number. *{00-59}*
3. *ss* = 2 character seconds number. *{00-59}*
4. *ff* = 2 character frames number. *{00-29}*
5. *mmm =* 3 character milliseconds. *{000-999}*

The timers can be accessed using the “Tmr” system variable. To set a timer to five minutes thirty seconds and 20 frames, enter the following command:

|  |
| --- |
| Tmr5 = 5:30.20  |

Each timer has two modes; “on” and “off”.

1. “on” : This sets a timer to increment.
2. “off”: Stops the system timer.

Timers can be printed, assigned to a variable, or another timer. All of the following are valid timer commands:

|  |
| --- |
| Tmr1 = 0   Tmr1 on   Tmr1 off  Print Tmr1  *00:00:00:05.22*  Tmr2 = Tmr1  Print Tmr2  *00:00:00:05.22*  x = Tmr1  Print x  *172*  Tmr2 = x + 5  Print Tmr2  *00:00:00:05.27* |

The main use for the general-purpose timers is for programming. The following show a sequence that uses timer 3:

|  |
| --- |
| ;10 second timer  Tmr3 = 0 ;initialize to 0  Tmr3 on ;turn timer on  J1  if Tmr3 >= 10.0 ;is timer > 10 seconds  Play bugle on 1 T1 ;yes, play sound  Tmr3 = 0 ;reset timer to 0  endif  Goto J1 |

# Timer Based Actions:

Timer events are based on the system timer (Internal, MIDI, SMPTE) and provide a means to specify a precise time to execute standard Sequence Programming commands, operations, and functions. A timer event consists of a cue point and a command, operation, or function. A cue point has a specific format that uses a bracketed timer constant in the form of [hh:mm:ss.ff] for frames mode and [hh:mm:ss.mmm] for frames and mixed mode . Timer events can be placed in the standard sequence program body or in an Event list.

## Event Lists:

The event list feature provides a linear show-control programming tool that works in conjunction with the standard sequence programming language. Each list entry must have a timer cue. All jump labels are ignored. Up to 64 event lists or shows may be dispersed among the 16 sequence programs. But, only one event list may be active at a given time. The format for an event list is as follows:

|  |
| --- |
| Eventlist <event label>  [hh:mm:ss.ff/mmm] <sequence instruction 1>  “” “”  [hh:mm:ss.ff/mmm] <sequence instruction n>  Endlist |

All conditional, looping, or branching instructions will be ignored. Only single line executable instructions may be included. Timer events must be sorted in the order that they occur. An example of an event list is shown:

|  |
| --- |
| Eventlist mainshow *;Event list header*  [10] Play Flim on 1 *;Play sound file Flim on channel 1*  [10] Dmx 1-1 on *;Turn on ctl out port 1 bit 1*  [10:35] Dmx 1-1 off *;Turn off ctl out port 1 bit 1*  [10:35] Play Sqam on 1 *;Play sound file Sqam on channel 1*  [10:35] Dmx 1-2 on *;Turn on ctl out port 1 bit 2*  [14:56] Dmx 1-2 off *;Turn off ctl out port 1 bit 2*  [14:56] TIMER = 0 *;Restart main show*  Endlist *;Event list footer* |

## Event List behavior:

The active event list maintains a pointer that increments through the list as the timer events are processed and will reposition based on timer mode and timer value changes. The following shows event list and timer actions based on timer mode changes:

|  |  |  |
| --- | --- | --- |
| **FROM** | **TO** | **Action** |
| TIMER off | TIMER on | Position list pointer, Enable timer and activate event list |
| TIMER off | TIMER hold | Position list pointer, Disable timer and activate event list |
| TIMER on | TIMER off | Disable timer and event list |
| TIMER on | TIMER hold | Disable timer, event list still active |
| TIMER hold | TIMER on | Activate timer, event list active |
| TIMER hold | TIMER off | Disable timer and event list |

The list pointer is **always** re-positioned anytime the timer is manually adjusted or the event source (*“EVENT = source”)* is changed regardless of the timer mode.

If the timer value is set to an earlier time than the current timer value then no timer events are performed.

When the timer value is set to a later time than the current timer then the behavior varies based on the timer mode. If the timer mode is set to “off” then no events are performed. However, if it is set to “on” or “hold” then all events between the old list pointer and the new list pointer are executed. This feature allows for manual/programmatic implementation of the system timer. Typically, the timer should be turned off before adjusting the timer forward.

## Embedded Timer Points:

Timer points can also be embedded in the sequence program body with other program instructions to provide pause points for a show. However, if the timer has passed the timer event before the line is executed, then the instruction is immediately performed and program execution continues as normal. Embedded timer points are not subject to Event List behavior. See the example sequence below:

|  |
| --- |
| TIMER = 0 ;Initalize the timer  TIMER on ;enable the timer  Main  [1:32.25] Play Flim on 1 T1 ;Play the sound “Flim”  [5:51.00] Play Apollo 13 on 1 T1 ;Play the sound “Apollo 13”  do ;Wait for “Apollo 13” to  while ?P1 <> 0 ;finish playing.  TIMER = 0 ;Reset the timer  Goto Main ;Loop to beginning |

This example waits for the timer to reach 1 minute, 32 seconds and 25 frames before playing the sound file “Flim”. The sequence then waits until the timer reaches 5 minutes and 51 seconds before playing the sound file “Apollo 13”. After “Apollo 13” finishes, the timer is reset and the show is repeated.

Here is another example that uses the time of day as a reference. This show uses the real-time clock as a timer reference and plays a show every day at Noon.

|  |
| --- |
| TIMER = CLOCK ;Initalize the timer  TIMER on 11:00:00 ;enable the timer with 11am offset  Main  [1:00:00] Play Flim on 1 T1 ;Play at 12 noon. 11:00:00+1:00:00  [1:10.00] Play Apollo 13 on 1 T1 ;Play at 12:10:00  Goto Main ;Loop to beginning |

Note: When the TIMER rolls over from 23:59:59.29 to 00:00:00.00 all events are re-initialized. The TIMER.DAYS field is ignored in all event list timing calculations.

Finally, this Event List shows an example of running output bits on DMX channel 1. Trigger one will pause the Event List and Trigger 2 resumes it. The sequence combines both the standard scripting and an event list in a single file:

|  |
| --- |
| EVENT = mainshow ;Select the “mainshow” event list  Tsync = LOCAL ;Set the timer source to onboard  TIMER = 0 ;Clear the timer  TIMER on ;Turn on the timer  EVENT on ;activate the event list  J1 if Close1 == 1 ;If trigger input 2 has closed then  TIMER off  Print # "Stop show", 13, 10  endif  if Close2 == 1 ;If trigger input 2 has opened then  TIMER on  Print # "Start show", 13, 10  endif  Goto J1  Eventlist mainshow ;Event list header  [1.00] Dmx 1-8 off  [1.00] Dmx 1-1 on  [2.00] Dmx 1-1 off  [2.00] Dmx 1-2 on  [3.00] Dmx 1-2 off  [3.00] Dmx 1-3 on  [4.00] Dmx 1-3 off  [4.00] Dmx 1-4 on  [5.00] Dmx 1-4 off  [5.00] Dmx 1-5 on  [6.00] Dmx 1-5 off  [6.00] Dmx 1-6 on  [7.00] Dmx 1-6 off  [7.00] Dmx 1-7 on  [8.00] Dmx 1-7 off  [8.00] Dmx 1-8 on  [8.00] TIMER = 0 ;Restart main show  Endlist ;Event list footer |

## Cue Lists

The Cue list feature provides another tool for linear show-control. Just like Event lists, the main timer provide the time basis but is independent and separate from the Event list system. They are an integral component to the timer synchronization system. A cue list consists of one or more cues. Cues are timed commands or actions that are dynamically added to the cue list. The actions can be created through Ethernet, Serial communications, external triggers, or other conditional criteria. Every Cue entry must have a timer reference. Once added to the list, it is continuously compared to the system timer and then executed when the system timer exceeds the specified cue time. The system has a maximum cue limit of 2048 concurrently loaded cues. The cue list will grow and shrink as cues are added, removed, or executed. Each cue can be associated with a group and each group can have a relative timer offset reference.

The example sequence below shows a typical use for cue lists. In this case, anytime a switch closure is detected a timer reference is set so that commands will be triggered 5 and 10 seconds later. Note that the cues are loaded “on the fly” and not pre-loaded. After the cues are loaded, the cue buffer is printed to the port the command originated from.

|  |
| --- |
| Define t1 = 300 ;10 seconds  TIMER on |
| J1 |
| if Close1 == 1 |
| Cue group 1 now |
| Cue group 1 [05.00] Dmx 1-1 on |
| Cue group 1 [05.00] Play Apollo 13 on 1 T1 |
| Cue group 1 [10.00] Dmx 1-1 off |
| Cue print |
| endif |
| if Close2 == 1 |
| Cue group 2 now |
| Cue group 2 [05.00] Dmx 1-2 on |
| Cue group 2 150 Play Sqam on 1 T2 |
| Cue group 2 t1 Dmx 1-2 off |
| Cue print |
| endif |
| Goto J1 |

# rPod8.4 Considerations:

The rPod8.4 has dual processors and can operate two separate and distinct scripting engines. Each engine is linked to a processor and compact flash card slot. The following table shows how the physical resources are connected:

|  |  |
| --- | --- |
| **Processor 1** | **Processor 2** |
| Compact Flash Slot 1 | Compact Flash Slot 2 |
| Console Serial Port | DTE Serial Port |
| DMX Lighting Control Interface | MIDI Control Interface |
| Triggers | Ethernet |
| Control Outputs | SMPTE (with RASR board) |
| Analog Input Control | ADAT (with RASR board) |
| System Clock Control |  |

A transparent communications interface is provided between the two processors. In general, cross usage of resources is provided by the operating system and only the following exceptions must be observed when programming the rPod 8.4:

1. Playback sound channels as defined by CMAP. The mode assignments are:

|  |  |
| --- | --- |
| CMAP  (Audio Channel Mapping) | Description |
| Split | P1: Channels 1-4, 9,10 P2:Channels 5-8,11,12 |
| P1 (processor 1) | P1: Channels 1-12 P2:NONE |
| P2 (processor 2) | P1: NONE P2: Channels 1-12 |

For example: When the system is in the **SPLIT** mode, sound files on a card in CF card slot 1 may only be played on channels 1-4 and subs 9 and 10. Likewise, files on a card in CF card slot 2 may only be played on channels 5-8 and subs 11 and 12.

1. Control files should only be loaded on CF card slot 1. However, sequences executed on processor 2 can load, play, and stop these files.
2. Global and Local variables defined by a processor are only visible to that processor. However, the “Pass” operator is provided to exchange constants and variables between the two processors.
3. To install a new operating system, the separate loader files (.ldr) for each processor must be resident in the associated CF card. CF slot 1 for Processor 1 and CF slot 2 for Processor 2.
4. Each processor maintains a separate system timer. However, they are derived from the same master clock source. The P1 timer can only be controlled from the Console port and P1 sequences. The P2 timer can only be controlled from the DTE Port and P2 sequences.

The two processors share the following common system resources:

1. Audio and DMX tracks: A common set of tracks is maintained for both processors.
2. Triggers and Output controls: Either processor may access these resources.
3. Console and DTE: Either processor may access these resources.
4. Analog inputs: They are mapped to both processors.
5. System Clock: Either processor may access the clock
6. Ethernet Data: Data is shared by both processors

# Quick Reference Guide:

**Command List index**

## File access:

1. Change directory cd “*drive name*” 
2. Display directory dir 
3. Change directory Chdir *“directory name”* 
4. Make directory Makedir *“directory name”*
5. Remove directory Remdir *“directory name”*
6. Delete file del *“file name”* 
7. Rename file Rename *“original file name”* as *“new file name”*
8. Open file *“filenumber”* = Openfile *“file name”* for *“filemode”*
9. Close file Closefile *“filenumber”*
10. File configuration Config FDELIM = *“character list”*
11. Write file with delim Writefile *“filenumber” “data list”*
12. Write file no delim Writefile # *“filenumber” “data list”*
13. Read file with delim Readfile *“filenumber”* *“variable list”*
14. Read file no delim Readfile # *“filenumber”* *“variable list”*
15. Read file to string Readfile $ *“filenumber”*  “*string variable”* *<count>*
16. Seek file *“position”* = Seekfile *“filenumber”* *“offset” <R>* *{CURR/BEG/END}*

## Sound File Playback:

1. Pre-Loading sounds stereo Ldsnd *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
2. Pre-Loading sounds mono Ldsndm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
3. Un-Loading sounds mono Unload *“track # list”* 
4. Playing sounds stereo Play *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
5. Playing sounds mono Playm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
6. Looping sounds stereo Loop *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
7. Looping sounds mono Loopm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 
8. Un-looping tracks Unloop *“track # list”* 
9. Convert to looping Makeloop *“track # list”* 
10. Appending sounds Append *“file name”* on *“track #”* 
11. Stopping sound tracks Stop *“track # list”* 
12. Stopping all tracks Stop all 
13. Pausing tracks Pause *“track # list”* 
14. Pausing all tracks Pause all
15. Resume tracks Resume *“track # list”* 
16. Resume all tracks Resume all
17. Mirror sound tracks Mirror *“track # list”* on/off
18. Set Play Position Pos *“track # list”* to F/S/T *“position reference”* 
19. Skip Play Position Skip *“track # list”* F/S/T *“position reference”* 
20. Pitch audio frequency Pitch *“track # list”* to “percent x 10” <in *“time (ms)”>* 
21. Set master sample rate Config SAMPLE *“sample rate”*

## Sound Volume Control:

1. Channel Volume Control Cvol *“ch # list”* = *“volume”* 
2. Track Volume Control Tvol *“track # list”* = *“volume”* 
3. Duck Volume Setting Dvol *“track # list”* = *“volume”* 
4. Duck Track Control Duck *“track # list”* 
5. Un-Duck Track Control Unduck *“track # list”* 
6. Input Volume Setting Ivol *“ch # list”* = *“volume”* 
7. Patch Assign (rPod only) Patch A/B= *“input ch list”* 
8. Input Mapping (rPod only) Patch A/Bto *“output ch# list”* 
9. Input Control (rPod only) Patch A/Bon/off 
10. Input Mapping (MS-II only) Ain MIC/ LIN/all 
11. Input Control (MS-II only) Ain on/off 
12. Boost Input Gain Boost *“input ch”* *“gain”* 
13. Mute All Mute 
14. Un-Mute All Unmute 
15. Track Attack Time Atk *“track # list”* = *“time”* 
16. Duck Track Attack Time Datk *“track # list”* = *“time”* 
17. Track Decay Time Dek *“track # list”* = *“time”* 
18. Duck Track Decay Time Ddek *“track # list”* = *“time”* 
19. Amplifier Control (MSII only) Amp on/off/stby/mute 

## Clips:

1. Load Clip Ldclip *“file name”*
2. Play Clip Clip *“file name”*
3. Un-load Clip Freeclip *“file name”*

## DMX/RS485 and Output Control Commands:

1. Bit Level Control Output *“control # list”* on/off/toggle 
2. Bit Level Control Dmx *“ch #”* - *“bit #”* on/off 
3. Byte Level Control Dmx *“ch #”* = *“output value”* 
4. Dimming Control Dimto *“output value”* in *“time (ms)”* on *“ch # list”* 
5. Output Driver Control Dmx DRIVE
6. Output Driver Control Dmx RELEASE

## DMX File Playback:

1. Pre-Loading DMX files Lddmx *“file name”* on *“track #”* 
2. Un-Loading DMX files Unloaddmx *“track #list”* 
3. Un-Loading DMX files Unloaddmx all 
4. Play DMX File Playdmx *“file name”* on *“track #”* 
5. Looping DMX File Loopdmx *“file name”* on *“track #”* 
6. Mounting DMX files Mount *“file name”* on *“track #”* 
7. Un-mounting DMX files Unmount *“track #”* 
8. Stopping DMX Channels Stopdmx *“ch # list”* 
9. Stopping all Channels Stopdmx all
10. Set Play Position Posdmx *“track # list”* to F/T *“position reference”* 
11. Skip Play Position Skipdmx *“track # list”* F/T *“position reference”* 

## DMX File Capture:

1. Record DMX files: Recdmx *“file name”* from *“ch # (start)”* to *“ch # (end)”* 
2. Pre-load Record DMX: Ldrecdmx *“file name”* from *“ch #* to *“ch #* T *“linked sound track #”* 
3. Starting Pre-loaded Record: Start 
4. Stop Recording DMX (1) Stop all 
5. Stop Recording DMX (2) Stopdmx all 

## Ethernet:

1. Configuring Config ENET *<“ip address”>* : <*“port”* >/<*”subnet address”>*
2. Cmd StatusReporting Verbose ENET on/off
3. Configuring FTP Password Config PASSWORD *“string”*
4. Configuring UDP Config UDP Manual/Auto(default)
5. IP Referencing Bind ENET *“ip address”* *to “bind number”* 
6. UDP Control Sendcmd *“bind number”* : *“constant string”*
7. UDP Control Sendcmds *“bind number”* : *”string variable”*
8. UDP Sending Data Send *“bind number”.<”packet number”>-<“ message type* “>|*“string or variable list”*
9. UDP Printing Data Printe *“ bind number”|”* *string or variable list”*
10. UDP Receive Data Recv *“string or variable list”*
11. UDP Reading Data Read UDP
12. UDP Peeking Data Peek UDP
13. UDP String termination Config ENET Term *“Term method”* 

## System Commands:

1. Re-Boot: Boot 
2. Setting the Time Base Config TIMEREF ms/frames
3. Cosmic Variable Commands Update
4. Cosmic Variable Commands Reload
5. Cosmic Variable Commands Remove *“variable list”/*all
6. Start Playback Start <T *“track # list”* D *“channel # list”*> 
7. Query Playing Tracks ?P 
8. Query Track Status ?T *“track # list”* 
9. Printing Print*(port)* *“variable list”* 
10. Print w/ control characters Print*(port)* **#** *“variable list”* 
11. Monitor Port Monitor Console/DMX/DTE/Midi on/off  *(rPod8.4 rev1.2 only)*
12. Monitor Port Monitor Console/DMX/DTE on/off  *(rPod8.4 rev1.0a only)*
13. Monitor Port Monitor Console/DMX/ on/off  *(MS2e)1*
14. Monitor Port Monitor all off 
15. Configure Serial Port: Config Console/DTE/DMX/Midi = *“baud rate”, “#bits”, “#stop bits”, “parity”*
16. Configure termination: Config Console/DTE/Midi Term = *“character list”*
17. Termination control Config Console/DTE/Midi Term on/off
18. DMX configuration Config DMX Standard/RS485
19. Midi configuration Config Midi Standard/RS232
20. DMX Receiver control Config DMX Receive on/off
21. Null character substitution Config Console/DTE nullsub on/off
22. Read Serial Port Read Console/DTE <options>
23. Peek Serial Port Peek Console/DTE <options>
24. Supress Port Output Silent <Console, DMX, DTE, MIDI, VMSKE/all> on/off
25. Command feedback control Verbose on/off 
26. Install Operating System Install *“file name”* 
27. Status Output Display Status *“output list”*  on/off/all
28. Configuring Triggers Config TRIG*“trigger list”*  = *“de-bounce time”*

## Logging

1. Log Activation Log on/off 
2. Log Update Log update 
3. Log Clearing Log clear 
4. Log Listing Log list 
5. Log Mode Log freq none/hour/day/month(default)
6. Log File Handling Log replace true/false(default)
7. Log Timestamp Log timestamp on/off(default)

## Sequence Programming Direct Commands:

1. Setting Sequence Context Context = *“sequence # (1-16)”* 
2. Local Variable Definition Define *“variable”* 
3. Global Variable Definition Define Global*“variable”* 
4. Local String Definition Define $*“variable”* 
5. Global String Definition Define Global $*“variable”* 
6. Public String Definition Define Public $*“variable”* 
7. Group Enumeration Formgroup *“file name”* as *“group #”* 
8. Starting a Group by File Playgroup *“file name”* 
9. Starting a Group by # Playgroup *“group #”* 
10. Starting a variable Group Playgroup *“variable”* 
11. Loading Sequence files Ldseq *“file name”* on *“sequence # (1-16)”*
12. Starting a Sequence Playseq *“sequence # (1-16)”*
13. Starting a Sequence Playseq *“file name”* {on *“sequence # (1-16)}* 
14. Starting a Sequence Startseq *“sequence # (1-16) list”*
15. Starting a Sequence Startseq *“file name list”*
16. Starting all Sequences Startseq all
17. Starting all Sequences **Go**
18. Stop a Sequence Halt *“sequence # list (1-16)”* 
19. Stop a Sequence Halt *“file name list”* 
20. Stop all Sequences Halt all
21. Stepping a Sequence Step *“# of steps”* 
22. Set breakpoints Break*“line #/label/variable list”* 
23. Clear breakpoints Break*“line #/label/variable list”* off
24. Clear all breakpoints Break all off 
25. Listing a Sequence List *“Starting line #/label” {* to *“end line #/label” /* :*”line count”}* 

## Program Flow:

1. Timer delay Wait *“time (ms)”* 
2. Random Timer delay Wait *“min time (ms)” – “max time (ms)”* 
3. Branch immediate Goto *“line label”* 
4. Branch conditional if *“operation”*   *mandatory header*

.  *additional program lines*

else  *optional*

.  *additional program lines*

endif  *mandatory footer*

1. Branch conditional: select *“variable”*   *mandatory header*

case *n*  *case start*

*.*  *additional program lines*

break   *case end*

*.*  *additional case/break options*

endsel *Select end*

1. While loop: do  *mandatory header*

.  *additional program lines*

while *“operation”*   *mandatory footer*

1. For Loop: for “*variable*” = “*operand1*” to “*operand2*” 

.  *additional program lines*

next  *mandatory footer*

## Event Programming:

1. Start of Event List Eventlist *“Event list name”* 
2. End of Event List Endlist 
3. Event List action Event on/off
4. Event Assignment Event = *“name”* 
5. Event Time Offset Event offset “*timer/constant/variable”* 
6. Timer windows TIMER window “*timer/constant/variable”* 
7. Timer setbacks TIMER setback “*timer/constant/variable”* 
8. Cueing a command Cue *“timer/constant/variable” “command”* 
9. Cueing with groups Cue group *“group number”* *“timer/constant/variable” “command”*
10. Cue group relative Cue group *“group number”* now
11. Cue group with zero offset Cue group *“group number”* never
12. Clearing a Cue Cue clear all
13. Clearing a Cue by group Cue clear group*“group number”* 
14. Print all Cue list Cue print <c><d><e><m><x>
15. Print next pending Cue Cue print <c><d><e><m><x> next
16. Print all Cued by group Cue print <c><d><e><m><x> next

## Timer Synchronization:

1. Sync command Sync on/off/now/block/allow
2. Sync rate Sync rate “*constant/variable”* 
3. Sync master Sync master/slave
4. Syncing audio tracks Lock *“track # list”* to *“offset time (frames)”* 
5. Un-syncing audio tracks Unlock *“track # list”* 

Note: All direct commands from the Command Entry Index are available as a sequence command.

# Reserved words:

These may not be used for jump labels or variable names (Case is sensitive).

;

?CF1

?CF2

?Console

?DMX

?D

?DTE

?ECOUNT

?ENET

?EORIG

?EPKT

?ETYPE

?FILE

?MMOVER

?P

?P*n*

?T

?T*n*

Ain

Amp

Append

Atk

Bind

Boot

Boost

break

Break

case

cd

Chdir

CLOCK

Closefile

Close*n*

Config

Console

Context

Cvol

Cvold

Cue

Datk

Ddek

Define

Dek

del

Dimto

dir

Dmx

DMXin

Dvol

do

DTE

Duck

Dvol

else

endif

EENABLE

EVENT

Eventlist

Endlist

Endsel

ETRIG

excludes

FDELIM

FERR

For

Formgroup

Framerate

Go

Goto

GTEtrig

includes

if

INFO

Inputs

Instring

IPADD

IPPORT

IPMASK

Ivol

Ivold

Lddmx

Ldrecdmx

Ldseq

Ldsnd

Ldsndm

Left

Len

List

Lock

Log

Loop

Loopdmx

Loopm

Makedir

Makeloop

Mid

Mirror

MMCMD

MMDATA

Mount

Mute

next

Note

Off

On

Open

Openfile

Output

Pass

Patch

Pause

Pausedmx

Peek

PFERROR

PFEVENT

PFREAD

PFSPEED

PFSTAT

PFUPDATE

PFWARNINGS

PFRDY

PFRUN

PFY

Pitch

Play

Playdmx

Playgroup

Playm

Playseq

PokeSKE

Pos

Posdmx

Print

Printc

Printd

Printe

Printn

Progress

Rand

RCCMD*n*

RCFCMD

RCFSTAT

RCIDIN

RCIDOUT

RCOM

RCPROF

RCRX

RCSCN

RCSEND

RCSEQ

RCSTAT*n*

RCSTICKY

Read

Readfile

Recdmx

Recv

Reload

Remdir

Remove

Rename

Resume

Right

Seekfile

Select

Send

Sendcmd

Sendcmds

setback

SIGPORT

Silent

Skip

Skip

Skipdmx

Skipdmx

Source

Start

Startseq

STATUS

Status

Step

Stop

Stopdmx

Str

Sync

Halt

Tdest

Test

TIMER

Tlist

Tmr*n*

Treport

Trig*n*

*Tsource*

Tsync

Tvar

Tvol

UDP

Unduck

Unload

Unloaddmx

Unlock

Unloop

Unmount

Unmute

Update

Val

Verbose

Wait

window

While

Writefile

XTRDY

XTRIG

# Table of Operations:

1. Direct *“operand1”* = *“operand2”*
2. Logical, single *“operand1”* = *“single operator” “operand2”*
3. Logical, double *“operand1”* = *“operand2” “logical operator” “operand3”*
4. Arithmetic *“operand1”* = *“operand2” “math operator” “operand3”*
5. If, eval if *“operand1”*
6. If, single op if *“single operator” “operand1”*
7. If, double op if *“operand1” “operator” “operand2”*
8. while, eval while *“operand1”*
9. while, single op while *“single operator” “operand1”*
10. while, double op while *“operand1”* = *“operand2”*
11. for for *“operand1”* = *“operand2”* to *“operand3”*

# Table of Operators:

1. Increment ++
2. Decrement --
3. Logical NOT !
4. Compliment ~
5. Subtraction -
6. Addition +
7. Multiplication \*
8. Division /
9. Modulus %
10. Logical AND &&
11. Binary AND &
12. Logical OR ||
13. Binary OR |
14. Binary XOR ^
15. Equality ==
16. Inequality <>
17. Less than or equal to <=
18. Greater than or equal to >=
19. Shift Left <<
20. Shift Right >>
21. Less than <
22. Greater than >
23. In range includes
24. Out of range excludes

# Table of System Constants and Subroutines:

1. Constants 25, 15, …
2. Local Variables counter, flag, Pass x…
3. Global Variables mode, status, …
4. Serial Input Status ?Console or ?DTE
5. File status ?FILE “filename”, ?FILE Console, and ?FILE DTE
6. System Strings $mystring, Console, DTE, and “” delimited
7. String Functions Val, Str, Len, Instring, Left, Right, and Mid
8. Random number Rand
9. Trigger Input Trig1, Trig2, …
10. Trigger Assertion Close1, Close2, …
11. Trigger Removal Open1, Open2, …
12. Trigger Word Inputs
13. DMX Output DMX1, DMX2, …
14. System Status STATUS
15. Timer Tmode, TIMER…
16. Real-Time Clock CLOCK
17. Audio Track Playing ?P1, ?P2, …
18. Audio Track %complete ?T1, ?T2, …
19. DMX Track Playing ?D1, ?D2
20. DMX framerate Framerate
21. Audio Inputs (rPod only) MICL, MICR, LINL, LINR
22. Multimover (MS2 only) ?MMOVER, MMCMD, MMDATA

# Command Description Index

## File Access Commands

### Change drive:

*Syntax*- cd “*drive name*” 

*Parameter*- “*drive name*” = “a”, ”b”

Both slots are active with slot # 1 as drive a: and slot #2 as drive b: The drives are selected using the “cd” command. To select drive b:, type the following at the prompt a:>.

|  |
| --- |
| cd b  |

Likewise, select drive a: using the command

|  |
| --- |
| cd a  |

The change drive command provides additional functionality on the rPod8.4. Normally, the Console port controls P1 and CF card 1 and the DTE controls P2 and CF card 2. When the directory is changed, control is redirected to the other processor. So, it is possible to have both the Console and DTE control the same processor.

### Directory:

*Syntax-* dir 

To display a directory of files on one of the drives, with the desired drive selected, type

|  |
| --- |
| dir  |

### Change Directory:

*Syntax-* Chdir *“directory name”* 

To open a directory of files on one of the drives, with the desired drive selected, type

|  |
| --- |
| Chdir Mydir\sound files   Play Mydir\sound files\sound1 |

To move to the parent of the currently selected directory, use dot notation in the directory string.

|  |
| --- |
| Chdir .. |

Typing ‘Chdir’ with no “directory name” parameters will print the current directory path.

The current directory path is also contained in the system string variable WORKDIR. It can be assigned to either local of global strings defined by the user.

|  |
| --- |
| $a = WORKDIR |

### Make Directory:

*Syntax-* Makedir *“directory name”* 

To make a new directory of files on one of the drives, with the desired drive selected, type

|  |
| --- |
| Makedir sound files |

### Remove Directory:

*Syntax-* Remdir *“directory name”* 

To remove a directory of files on one of the drives, with the desired drive selected, type

|  |
| --- |
| Remdir sound files  |

### Delete:

*Syntax-* del *“file name”* 

*Parameter* - “*file name*” = A file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional.

To delete a file on one of the drives, with the desired drive selected, type

|  |
| --- |
| del Flim  |

### Rename:

*Syntax-* Rename *“original file name”* as *“new file name”*

*Parameter* - “*file name*” = A file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional.

To rename the file on one of the drives, with the desired drive selected, type

|  |
| --- |
| Rename Flim as Flimbackup  |

### Openfile:

*Syntax-* *“filenumber”* = Openfile *“file name”* for *“filemode”*

*Parameter* - *“filenumber”* = A file reference number that is returned when the “Openfile” command is called. It may be a local or global variable previously defined.

“*file name*” = A file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional.

“*file mode*” = This is a flag defining file mode. The following is a list of all the options:

"r" Open an existing file for reading. The file pointer is positioned at the beginning of the file.

"r+" Open an existing file for reading and writing. The file pointer is positioned at the beginning of the file.

"w" Clears an existing file to zero length or creates a new file for writing. The file pointer is positioned at the beginning of the file.

"w+" Open a file for reading and writing. If the file does not exist, then it is created. Otherwise, it is cleared to zero. The file pointer is positioned at the beginning of the file.

"a" Open for appending (writing to end of file). The file is created if it does not exist. The file pointer is positioned at the end of the file.

"a+" Open a file for reading and appending. The file is created if it does not exist. The file pointer is positioned at the end of the file.

The ‘Openfile’ command is used to open an existing file or create a new file for reading or writing. There must be a companion ‘Closefile’ for every ‘Openfile’ issued. A file number reference obtained, using this command, is can be subsequently used by the ‘Closefile’, ‘Writefile’, ‘Readfile’, and ‘Seekfile’ commands.

To create a new file for writing, type

|  |
| --- |
| Define Fn  Fn = Openfile data.txt for w |

### Closefile:

*Syntax-* Closefile *“filenumber”* 

*Parameter* - *“filenumber”* = A file reference number that is returned when the “Openfile” command is called. It may be a local or global variable previously defined.

The ‘Closefile’ command is used to close a previously opened file. There must be a companion ‘Closefile’ for every ‘Openfile’ issued. A file number reference is obtained using the ‘Openfile’ command and can be subsequently used by the ‘Writefile’, ‘Readfile’, and ‘Seekfile’ commands.

|  |
| --- |
| Closefile Fn |

### File Configuration:

*Syntax-* Config FDELIM = *“character list”*

*“character list”-* A comma delimited list of ASCII codes (decimal/hex), local variables, global variables.

The delimiter string is used to separate data when using the ‘Writefile’ and ‘Readfile’ commands. The default delimiter is a comma (,). To change the File system delimiter string to a tab and space type:

|  |
| --- |
| Config FDELIM = 9, 0x20 |

To set the delimiter back to a comma, use 44 (ascii code) to set:

|  |
| --- |
| Config FDELIM = 44 |

### Write File:

*Syntax-* Writefile *“filenumber”* *“data list”* 

Writefile # *“filenumber”* *“data list”* 

*“filenumber”* *-* This parameter is a file reference number that is returned when the ‘Openfile’ command is called. It may be a local or global variable previously defined.

*“Data list”-* A comma delimited list of numbers (decimal/hex), local variables, global variables, and strings (quote delimited, Console, or DTE). By specifying the context prior to writing, the local variables for a specific sequence can be saved to file. When using the ‘Writefile #’ version, placing a ‘#’ character before the variable(s) causes it to be written as a single byte, ASCII code instead of a four byte, 32 bit word.

*Special system variables-*

*FDELIM-* this is asystem string variable containing the record delimiter. The default delimiter is a comma. It is also used for the ‘Readfile’ and ‘Seekfile’ commands.

*FERR-* this is asystem variable containing the result of the last file operation. FERR is set to zero if the operation was successful. FERR = 2 if the operation failed.

The ‘Writefile’ command is used to write data to a previously opened file. The file must not be opened with the read only flag (‘r’) selected or an error will occur (see FERR).

There are two versions of the ‘Writefile’ command. The first version, ‘Writefile’,uses the system delimiter string to separate variables written to a file. This provides a way for the companion ‘Readfile’ command to extract variable parameters without needing to keep track of parsing.

The second version, ‘Writefile#’, does not use the system delimiter when writing a file. Data list items are directly appended to the previous item. This mode writes only numeric values and not strings. Values can be written as single bytes or 32bit words. To write data as a single byte, place a ‘#’ symbol before the numeric variable or constant value. Parsing is left to the user using the ‘Readfile #’ command. This version provides a method for writing fixed record sizes.

To write four values to the file “data.txt”, enter:

|  |  |
| --- | --- |
| Define Fn  Fn = Openfile data.txt for w  Writefile Fn 1, 2, 3, 4  Closefile Fn |  |

This example shows how to save channel volume settings to a binary file:

|  |
| --- |
| Define volume1 = 70   Define Volume2 = 110   Fn = Openfile config.hex for w  Writefile # Fn #Volume1, #Volume2 |

The # symbol in front of Volume1 and Volume2 indicated that the byte value will be written to the file and not the four byte word.

### Read File:

*Syntax-* Readfile *“filenumber”* *“variable list”* 

Readfile # *“filenumber”* *“variable list”* 

Readfile $ *“filenumber”*  “*string variable”* *<count>*

*“filenumber”* *-* This parameter is a file reference number that is returned when the ‘Openfile’ command is called. It may be a local or global variable previously defined.

*“variable list”-* A list oflocal and global variables, or local and global strings. By specifying the context prior to writing, the local variables for a specific sequence can be read from a file.

To read a file containing raw hex data, place a ‘#’ character before the “filenumber”. Raw data can be read as a single byte or a 32 bit word. One byte can be read into the number variable by placing a ‘#’ symbol before that variable. Four byte integer values will be read into numeric variables without the ‘#’ symbol. String data can be contained within the raw hex data and can be read to a string variable, using the standard null string termination.

Placing a ‘$’ before the file reference causes a read of ‘count’ bytes into the specified string. Only a single string variable may be read in this mode. If the ‘count’ parameter is not specified then null string termination will be used.

*“string variable” -* local and global string variables.

*<count> -* The optional count reference is used to specify how many characters are read into the specified string variable.

*Special system variables-*

*FDELIM-* this is asystem string variable containing the record delimiter. The default delimiter is a comma. It is also used for the ‘Readfile’ and ‘Seekfile’ commands.

*FERR-* this is asystem variable containing the result of the last file operation. FERR is set to zero if the operation was successful. FERR = 1 if the operation was successful and the end of file was detected. FERR = 2 if the operation failed.

The ‘Readfile’ command is used to read data to a previously opened file. The file must not be opened with the write only flag (‘w’ or ‘a’) selected or an error will occur (see FERR).

There are three versions of the ‘Readfile’ command. The first version, ‘Readfile’, uses the system delimiter string to separate variables read from a file.

The second version, ‘Readfile $’, reads raw string data into a single string variable.

The third version, ‘Readfile#’, does not use the system delimiter when reading a file. A ‘#’ symbol before a number variable causes 1 byte to be read. A plain number variable with no ‘#’ symbol indicates that 4 byte will be read and converted to an integer. Strings are read using null termination.

|  |  |
| --- | --- |
| Define Fn  Define n1  Define n2  Define n3  Define n4  Fn = Openfile data.txt for r  Readfile Fn n1, n2, n3, n4  Closefile Fn |  |

|  |
| --- |
| Define volume1  Define Volume2  Fn = Openfile config.hex for r  Readfile # Fn #Volume1, #Volume2  Closefile Fn  Print volume1,” “,volume2  *70 110* |

### Seek File:

*Syntax- “position”* = Seekfile *“filenumber”* *“offset” <R>* *{CURR/BEG/END}*

*“position”* *-* position returns the file character position after the command is executed . It must be a local or global variable previously defined.

*“filenumber”* *-* This parameter is a file reference number that is returned when the “Openfile” command is called. It must be a local or global variable previously defined.

*“offset” -* The relative byte positions offset as defined by the CURR, BEG, or END whence parameter. It must be a local or global variable previously defined or a constant value. The default CURR is used if no parameter is specified.

*<R>* Record flag is optional and can be used to point to a specific record offset. The file system delimiter string defines the separation of records.

*CURR -* The offset is referenced from the current file position.

*BEG -* The offset is referenced from the beginning of the file.

*END -* The offset is referenced, backwards, from the end of the file.

The ‘Seekfile*’* command is used to position the file pointer in a previously opened file. This command is used in conjunction with the ‘Readfile’ command. To obtain the current file position, use an offset of zero and the default “CURR” reference. If the offset is larger than the file size then the end of file position is returned.

If the file “data” contains the following records:

|  |  |
| --- | --- |
| 254,32,this is some text,850,2000 |  |

Then here are the results when the following commands are issued:

|  |  |
| --- | --- |
| Define Fn  Define a  Define p  Fn = Openfile data for r  p = Seekfile Fn 1 R BEG ;position to second record  Print p  *4*  Readfile Fn a ;read second record  Print a  *32*  p = Seekfile Fn 0  ;get current position  Print p  *7*  p = Seekfile Fn 0 R END ;position to last record  Print p  *29*  Readfile Fn a ;read last record  Print a  *2000*  Closefile Fn |  |

## Sound File Playback

### Pre-loading a Sound:

*Syntax-* Ldsnd *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

Ldsndm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

*Parameters* - “*file name*” = Sound file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Global and local string variables and Console or DTE are also valid.

*“ch # list” =* Output sound channel # list. Multiple channels may be specified with a ‘,’ delimiter. Channels may be referenced by local or global variables.

*Options*-T = Flags manual assignment of tracks (optional). The ‘T’ flag must be followed by a track list.

*“track # list” =* This list should have the same number of items as the channel # list. Tracks may be referenced by local or global variables.

X = Flags track cross-fading (optional). The ‘X’ flag must be followed by a cross-fade list.

*“cross-fade list” =* List of the tracks currently playing to be cross-faded with. Each track in this list will be faded out and the newly assigned tracks faded in. Cross-fade tracks may be referenced by local or global variables.

The pre-loading feature can be used to load multiple sound and DMX files from the CF card to the processors sound and control buffers. The sound and control files can then be played using the “Start” command. This method of playing files prevents any time lag in playback due to buffer cuing and forces sample accurate synchronization of multiple sound and control files.

To Pre-Load a sound type:

|  |  |
| --- | --- |
| Define ch5 = 5   Ldsnd Flim on 5   Ldsndm Flim on 5   Ldsnd Flim on ch5  | * Variable Definition*  * Stereo*  * Mono*  * Variable Ref* |

This will pre-load the file “Flim.wav” on channel 5. The track will be automatically selected. The system takes the first free track available. If “Flim.wav” is a stereo file then it will play back on channels 5 and 6. Pre-loaded sounds will begin playing when the next Start command is issued.

To pre-load a sound on multiple channels, type

|  |  |
| --- | --- |
| Define ch5 = 5  Define ch7 = 7  Ldsnd Bell Tree on 1,3  Ldsnd Bell Tree on ch5, ch7   Start  | * Variable Definition*  * Variable Definition*  * Constant Ref*  * Variable Ref*  * Start Playback* |

This will play “Bell Tree.wav” on channels 1, 3, 5, and 7. If the file is stereo, it will play on channels 1, 2, 3, 4, 5, 6, 7, and 8.

Sound pre-loading may be layered by selecting the same channel #. The “.wav” extension may be included. If the channel is not specified then the last channel that was used will be selected.

Optional Track assignment:

The track will automatically be assigned unless the T flag specified. Example:

|  |
| --- |
| Ldsnd Bell Tree on 1,3,5 T 1,2,3  |

Any tracks that are in use are overridden when the Start command is issued.

Optional cross-fade tag:

Use the ‘X’ tag to cause a cross-fade between other playing tracks. The ‘X’ tag stops playback of tracks on any channel. If the track has a non-zero decay setting, it will decay to zero before stopping. The parameter(s) after the ‘X’ specify the tracks to be faded out. Please note that tracks do not have to be on the same channel. Example:

|  |
| --- |
| Atk 1,2,3,4,5,6 = 30   Dek 1,2,3,4,5,6 = 30   Play Bells on 1,3,5 T 1,2,3   ;play sound for a while  Ldsnd Bell Tree on 1,3,5 T 4,5,6 X 1,2,3   Start  |

The above example the sound “Bells.wav” begins playing on tracks 1,2 and 3. After it plays for a while, the sound “Bell Tree.wav” will be pre-loaded on tracks 4,5, and 6. When a ‘Start’ command is issued, tracks 4, 5, and 6 will begin with the specified attack rate (see ‘Atk’ command) and the tracks 1, 2, and 3 will end with a decay rate (see ‘Dek’ command).

### Unloading a Sound Track:

*Syntax-* Unload *“track # list”* 

*Parameters* -  *“track # list” =* This list should have the same number of items as the channel # list. Tracks may be referenced by local or global variable.

The Un-loading feature can be used to release or remove multiple sound tracks that are waiting to be played. Once un-loaded they will not be played when a “Start” command is issued.

To un-Load a sound track type:

|  |  |
| --- | --- |
| Unload 1  | * remove pre-loaded track* |

### Playing Sounds:

*Syntax-* Play *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

Playm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

*Parameters* - “*file name*” = Sound file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Global and local string variables and Console or DTE are also valid.

*“ch # list” =* Output sound channel # list. Multiple channels may be specified with a ‘,’ delimiter. Channels may be specified with local or global variables.

*Options*-T = Flags manual assignment of tracks (optional). The ‘T’ flag must be followed by a track list.

*“track # list” =* This list should have the same number of items as the channel # list. Tracks may be specified with local or global variable.

X = Flags track cross-fading (optional). The ‘X’ flag must be followed by a cross-fade list. Tracks may be specified with local or global variables.

*“cross-fade list” =* List of the tracks currently playing to be cross-faded with. Each track in this list will be faded out and the newly assigned tracks faded in. Cross-fade tracks may be specified with local or global variables.

To play a sound, type:

|  |  |
| --- | --- |
| Define ch5 = 5  Play Flim on 5   Playm Flim on 5   Play Flim on ch5  | * Variable Definition*  * Stereo*  * Mono*  * Variable Ref* |

This will play the file “Flim.wav” on channel 5. The track will be automatically selected. The system takes the first free track available. If “Flim.wav” is a stereo file then it will play back on channels 5 and 6.

To play a sound on multiple channels, type

|  |
| --- |
| Define dog = 1  Define cat = 3  Define pig = 5  Play Bell Tree on dog, cat, pig  |

The example above plays “Bell Tree.wav” on channels 1, 3, and 5. If it is a stereo file then it will play on channels 1, 2, 3, 4, 5, and 6.

Sound play back may be layered by selecting the same channel #. The “.wav” extension may be included. If the channel is not specified then the last channel used is selected.

Optional Track assignment:

The track will automatically be assigned unless the ‘T’ tag is specified. Example:

|  |
| --- |
| Play Bell Tree on 1,3,5 T 1,2,3  |

Any tracks that are in use are overridden.

Optional cross-fade tag: Use the ‘X’ tag to cause a cross-fade between other tracks. The optional tag may be attached to the command. The ‘X’ tag stops playback of tracks on any channel. If the track has a non-zero decay setting, it will decay to zero before stopping. The parameter(s) after the ‘X’ specify the tracks to be faded out. Please note that tracks do not have to be on the same channel. Example:

|  |
| --- |
| Play Bell Tree on 1 X 2,3,4  |

This above command cross fades the file “Bell Tree” with sounds playing on tracks 2, 3, and 4.

### Looping:

*Syntax-* Loop *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

Loopm *“file name”* on *“ch # list”* T *“track # list”* X *“cross-fade list”* 

*Parameters* - “*file name*” = Sound file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Global and local string variables and Console or DTE are also valid.

*“ch # list” =* Output sound channel # list. Multiple channels may be specified with a ‘,’ delimiter. Channels may be specified with local or global variables.

*Options*-T = Flags manual assignment of tracks (optional). The ‘T’ flag must be followed by a track list.

*“track # list” =* This list should have the same number of items as the channel # list. Tracks may be specified with local or global variables.

X = Flags track cross-fading (optional). The ‘X’ flag must be followed by a cross-fade list.

*“cross-fade list” =* List of the tracks currently playing to be cross-faded with. Each track in this list will be faded out and the newly assigned tracks faded in. Cross-fade tracks may be specified with local or global variables.

To loop play a sound, type:

|  |  |
| --- | --- |
| Loop Flim on 1  Loopm Flim on 1  | * Stereo*  * Mono* |

This will Loop back Play the file “Flim.wav”. The track(s) will be automatically selected. The system takes the free track(s) available. If “Flim.wav” is a stereo file then it will play back on channels 1 and 2.

Optional Track assignment:

The track will automatically be assigned unless specified by the ‘T’ flag. Example:

|  |
| --- |
| Loop Flim on 1 T2  |

Any tracks that are in use are overridden.

### Un-Looping:

*Syntax-* Unloop *“track # list”* 

*Parameter* - *“track # list” =* Track(s) to have looping mode canceled. Tracks may be specified with local or global variables.

To convert two looping tracks to tracks that stops when the end of the sound is reached, type:

|  |
| --- |
| Unloop 1,2 |

### Convert to Looping

*Syntax-* Makeloop *“track # list”* 

*Parameter* - *“track # list” =* Track(s) to convert to looping mode. Tracks may be specified with local or global variables.

To convert a track that stops when the end of the sound is reached to a looping track, type:

|  |
| --- |
| Makeloop 3 |

### Appending:

*Syntax-* Append *“file name”* on *“track #”* 

*Parameters* -- “*file name*” = Sound file name on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional. Global and local string variables and Console or DTE are also valid.

*“track #” =* Track number. Tracks may be specified with local or global variables.

To “Append” a new sound to a playing track type:

|  |
| --- |
| Append Bell Tree on 1 |

This will start playback of the file “Bell Tree.wav” on track 1 when the currently active file completes playback. The track must be active or the append command will return an error. If the track is set for looping playback then appended file will also loop playback.

### Stopping Sound Tracks:

*Syntax-* Stop *“track # list”* 

Stop all 

*Parameter* -- *“track # list” =* List of tracks that are actively playing sounds. Tracks may be specified with local or global variables.

To stop a playing sound on a specific track, enter:

|  |
| --- |
| Stop 1,2 |

This will stop tracks 1 and 2. If the track has a non-zero decay setting, it will decay to zero in the time specified before stopping. To stop all playing sound tracks enter:

|  |
| --- |
| Stop all |

### Pausing Sound Tracks:

*Syntax-* Pause *“track # list”* 

Pause all

Resume *“track # list”* 

Resume all

*Parameter* -- *“track # list” =* List of tracks that are actively playing sounds. Track numbers may be specified with global or local variables.

The ‘Pause’ and ‘Resume’ commands are used to freeze audio playback of either specific playing audio tracks or all tracks. The command expects an operational parameter consisting of either a list of tracks in comma-delimited form or the keyword “all”. As an example, if the first four tracks (1-4) are playing, then to pause tracks two and three enter:

|  |
| --- |
| Pause 2,3  |

This command pauses tracks two and three while tracks one and four continue playback.

The command:

|  |
| --- |
| Pause all  |

This will cause all four playing tracks to pause. Playback can be resumed by commanding either specific tracks or all paused tracks using the following commands:

|  |  |
| --- | --- |
| Resume 2,3   Resume all  | Or |

### Mirroring Sound Tracks (rPod only):

*Syntax-* Mirror *“track # list”* on/off 

*Parameter* -- *“track # list” =* List of tracks that are actively playing sounds. Track numbers may be specified with global or local variables.

The ‘Mirror’ command allows a single file and track to be played back on multiple channels that are controlled by the same processor. For processor 1 the channels are 1,2,3,4 and processor 2 controls channels 5,6,7,8. To play back a stereo file on channels 1,2,3,4 enter the following:

|  |
| --- |
| Mirror 1 on   Play tone on 1 T1  |

Mirroring varies depending on whether or not the file is mono or stereo and the type of playback command used, Play or Playm. The table below describes the four possible cases:

|  |  |  |
| --- | --- | --- |
| file  mono  stereo  mono  stereo | Command  Playm  Playm  Play  Play | Channel Mirroring  1-3,2-4,5-7,6-8  1-2-3-4,5-6-7-8  1-2-3-4,5-6-7-8  1-3,2-4,5-7,6-8 |

### Positioning Sound Tracks:

*Syntax-* Pos *“track # list”* to F/S/T *“position reference”* 

Skip *“track # list”* <</>> F/S/T *“position reference”* 

These commands are used to reposition audio playback to a different place within the sound file. The ‘Pos’ command positions playback with respect to the beginning of the File and the ‘Skip’ command positions playback with respect to the current playback position. The track may either be playing or pre-loaded. The commands expect a list of tracks in comma-delimited format and a position reference. The position reference can be expressed in three different ways. An “F” denotes frames reference where each frame is 1/30th second. The “S” identifies the position reference in audio samples where there are 44,100 samples/sec. Finally, the “T” indicates time in units of milliseconds. If the “F”, “S”, or “T” is omitted then time reference “T” is used as the default.

To position Tracks 1 and 3 10secs into the sound enter:

|  |  |
| --- | --- |
| Pos 1,3 to F300   Pos 1,3 to S441000   Pos 1,3 to T10000  |  Example 1   Example 2   Example 3 |

When using the ‘Skip’ command, the position reference is either forward “>>” or backward “<<”. To adjust the current playback position backwards 63 seconds type:

|  |  |
| --- | --- |
| Skip 1 << F1890   Skip 1 << S2778300   Skip 1 << 1:03  |  Example 1   Example 2   Example 3 |

Time and Frames reference may be entered in mixed mode format where:

Time: hh:mm:ss.ms

Frames: hh:mm:ss.frames

Where: hh = hours

mm = minutes

ss=seconds

ms=milliseconds

frames = number of frames

### Frequency Pitching Sound Tracks:

*Syntax-* Pitch *“track # list”* to “percent x 10” <in *“time (ms)”>* 

*Parameter* **--** *“track # list”* ***=*** List of tracks that are actively playing sounds. Track numbers may be specified with global or local variables*.*

*“percent x 10” =* This is the percent of sample rate time ten*.* Range 0 – 2000.

*“time (ms)” =*This optional parameter is expressed in milliseconds. Times may be specified with local or global variables.

This command is used to alter the playback sample rate of the specified track(s). It will only affect the tracks specified and cannot be used when a track is timer locked. The pitch amount is determined by a percentage times 10. Using a percentage of 1000 will cause playback at normal speed. A percentage of 2000 will playback at twice the speed. A percentage of 500 will playback at half speed. If the time parameter is not specified then the change will be made immediately. Otherwise, the pitch will be slewed to the new frequency over the specified time period.

To pitch Tracks 1 and 2 to 110% over a 10 second period enter:

|  |
| --- |
| Pitch 1,2 to 1100 in 10000 |

### Setting the Global Playback Sample Rate:

*Syntax-* Config SAMPLE *“sample rate”* (default is 44100)

*Parameters* -- *“sample rate” =* rate may be specified with a constant, local or global variables.

The command ‘Config SAMPLE’ Sets the audio sample rate for all channels. The default setting is 44.1K samples/sec. This command affects all tracks. Not all sample rates can be achieved. The selected sample rate will be returned by the command. To set the sample rate to 48000 samples/sec type:

|  |
| --- |
| Config SAMPLE 48000 |

## Sound Volume Control

### Channel Volume Control:

Channel Volumes are set during boot up in a specific order using three separate criteria. First, the volumes are set to the default of 64. Next the appropriate “.ini” file will contain saved Cvol settings. These settings are saved each time Cvol is changed. Finally, if a codec configuration file is detected then the Cvol setting established in Sigma Studio will override the “.ini” settings. After booting the unit, any changes made to Cvol through a sequence program or via the command console will be made. It is important to stress that codec configuration file settings will be used just after re-powering the unit.

*Syntax-* Cvol *“ch # list”* = *“volume”* 

*Parameters* --*“ch # list” =* Output sound channel # list. Multiple channels may be specified with a ‘,’ delimiter. Channels may be specified with local or global variables.

*“volume” =* Sound volume setting from 0 – 127 where 0 = OFF and 127 = max volume. Volume values may be specified with local or global variables. Volume values can also be entered in decibels. Range -90 dB – 0 dB.

To set the Volume of any channel:

|  |
| --- |
| Cvol 1,2,3,4 = 64 or  Define vol1 = 64  Cvol 1,2,3,4 = vol1 or  Cvol ch1, ch2, ch3 = vol1  Cvol 6, 7 = -10 dB |

This sets the volume on channels 1, 2, 3, and 4 to a volume level of 64. Channel values can range from 1-12 for the rPod10.2, 1-12 for the rPod8.4 and 1-3 for the MSii. The derived subwoofer channels are the last numerical channels. The table below shows the channel relationships:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **rPod10.2** | | **rPod8.4** | | **Msii** | |
| Primary Channels | Subwoofer Channels | Primary Channels | Subwoofer Channels | Primary Channels | Subwoofer Channel |
| 1,2 | 11 | 1,2 | 9 | 1,2 | 3 |
| 3,4 | 12 | 3,4 | 10 |  |  |
| 5-10 | none | 5,6 | 11 |  |  |
|  |  | 7,8 | 12 |  |  |

Volume settings can range from 0(min volume) to 127(max volume). The default is 127. Channel volume controls are set in hardware using the gain control sections of the codec’s. The default channel volume is 127.

### Track Volume Control:

*Syntax-* Tvol *“track # list”* = *“volume”* 

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

*“volume” =* Sound volume setting from 0 – 127 where 0 = OFF and 127 = max volume. Volume values may be specified with local or global variables. Volume values can also be entered in decibels. Range -90 dB – 0 dB.

To set the volume of any track:

|  |
| --- |
| Tvol 1,5 = 54 |

This sets the volume on tracks 1 and 5 to 54. Track values can range from 1- Max Tracks. Volume settings can range from 0(min volume) to 127(max volume). The default is 127. The volume change duration are set with the ‘Atk’ and ‘Dek’ commands. Track Volume control is scaled in software and is affected by the Channel volume control and any gains or attenuations set within the Sigma DSP.

To set the volume of a track to -3 dB:

|  |
| --- |
| Tvol 1,5 = -3 dB |

### Ducking Volume Setting:

*Syntax-* Dvol *“track # list”* = *“volume”* 

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

*“volume” =* Sound volume setting from 0 – 127 where 0 = OFF and 127 = max volume. Volume values may be specified with local or global variables. Volume values can also be entered in decibels. Range -90 dB – 0 dB.

To Set the ducking level value for a set of tracks:

|  |
| --- |
| Dvol 1,2,3,4 = 32  Dvol 5 = -12 dB |

### Duck Track Control:

*Syntax-* Duck *“track # list”* 

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

To Duck the volume of any track:

|  |
| --- |
| Duck 1,2 |

This sets the volume on tracks 1 and 2 to the ducking volume setting (‘Dvol’). The system overrides the track volume setting until the ‘Unduck’ command is received.

# **Un-Duck Track Control:**

*Syntax-* Unduck *“track # list”* 

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

To un-duck the volume of any track:

|  |
| --- |
| Unduck 1,2 |

This un-ducks tracks 1 and 2 to the track volume setting (Tvol).

### Input Volume Setting:

*Syntax-* Ivol *“ch # list”* = *“volume”* 

*Parameters* --*“ ch # list” =* Multiple input channels may be specified with a ‘,’ delimiter.

rPod10.2, rPod8.4:

MICL = Left Microphone Input, MICR = Right Microphone Input

LINL = Left Line Level Input, LINR = Right Line Level Input

MS-II:

INL = Left Mic./Line Input, INR = Right Mic./Line Input

all = Both left and right input channel

*“volume” =* Sound volume setting from 0 – 127 where 0 = OFF and 127 = max volume. Volume values may be specified with local or global variables.

To set the input volume for all channels on the rPod10.2, rPod8.4:

|  |
| --- |
| Ivol MICL,MICR,LINL,LINR = 80 |

To set the input volume for both left and right input channels on the MS-II:

|  |
| --- |
| Ivol all = 80 |

### Patch Assignment: (rPod10.2, rPod8.4 only)

*Syntax-* Patch A/B= *“input ch list”* 

*Parameters* – Patch assignment = A or B

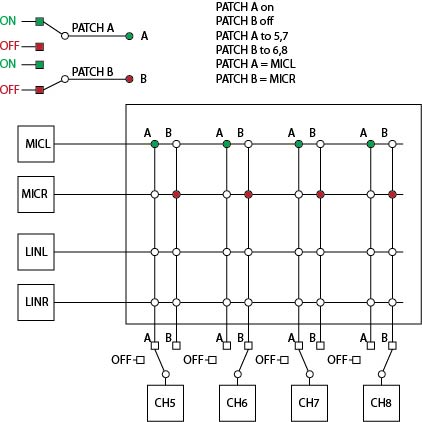
*“input ch list” =* Multiple input channels may be specified with a ‘,’ delimiter.

MICL = Left Microphone Input, MICR = Right Microphone Input

LINL = Left Line Level Input, LINR = Right Line Level Input

This command assigns source input channels to the patch specified. The system supports two patches, A or B. The input channels may be assigned to either or both patches. To assign a microphone input to patch A and the line level inputs to patch B type:

|  |
| --- |
| Patch A = MICL   Patch B = LINL, LINR  |



Input Mapping: (rPod10.2, rPod8.4 only)

*Syntax-* Patch A/Bto *“output ch# list”* 

*Parameters* – Patch assignment = A or B

*“ch # list” =* Output sound channel # list. Multiple channels may be specified with a ‘,’ delimiter. Channels may be specified with local or global variables.

This command assigns the destination output channels to the patch specified. The system supports two patches, A or B. The output channels may be mapped to either or both patches. To assign even output channels to patch A and the odd output channels to patch B type:

|  |
| --- |
| Patch A to 1,3,5   Patch B to 2,4,6  |

### Input Control: (rPod10.2, rPod8.4 only)

*Syntax-* Patch A/Bon/off 

*Parameters* – Patch assignment = A or B

on = Connects the inputs to the outputs specified by Patch Assignment and Input Mapping commands.

off = Disconnects the inputs from the outputs specified by the Patch Assignment and Input Mapping commands.

This command connects or disconnects the patch specified. The system supports two patches, A or B. To activate patch A type:

|  |
| --- |
| Patch A on  |

Input Control:

*Syntax-* Ain = MIC/ LIN/all  (MS-II only)

Ain = ADAT  (rPod8.4 only)

Ain = ANALOG  (rPod8.4 only)

Ain on/off 

*Parameters* – MIC= Selects the mono microphone input channel.

LIN= Selects the stereo line level input channels.

all= Selects the stereo line level input and the microphone channels. (default)

ADAT= Selects the RASR board ADAT optical interface inputs.

ANALOG= Selects the onboard A/D analog audio inputs.

on = Connects line or microphone inputs to audio output channels.

off = Disconnects mic./line inputs from the audio output channels. (default)

The analog inputs are disconnected when the MS-II powers up. To connect the inputs and enable audio from the selected source type:

|  |  |
| --- | --- |
| Ain on  |  connect input |

To disable the selected audio input type:

|  |  |
| --- | --- |
| Ain off  |  disconnect input |

To select just the microphone input type:

|  |
| --- |
| Ain = MIC  |

### Boost Microphone Gain:

*Syntax-* Boost *“input ch”* *“gain”* 

*Parameters* – Input channel = MICL or MICR or all(rPod10.2, rPod8.4 only)

Input channel = INL, INR, or all (MS-II only)

Gain = rPod10.2, rPod8.4:

Four microphone amplifier gain settings are available. These are 0db (no gain), 20db, 40db, and 60db (max gain). For rPOD-8.4 Rev 1.2 and later models, you can set the gain from 0 db to 60 db. The settings range from a 0 db setting, then go from 9 db to 60 db in three db steps. In between settings will be rounded to the closest value.

MS-II:

The MIC input has a fixed 40 db gain stage and is fed to both L and R channels of the ADC. It is summed with the Line L and Line R at the ADC as well.

Five additional gain settings are available. These are 0db (no gain), 3db, 6db, 9db and 12db (max gain). For the MIC these gains add to the 40dB of gain, for the line input, these five gains are absolute gains.

This command sets the microphone input gain on the specified input channel. To set the left microphone input to 40db gain type:

|  |  |
| --- | --- |
| Boost MICL 40db   Boost MICL 0db  |  Example 1: rPod10.2, rPod8.4   Example 2: MS-II |

### Mute:

*Syntax-* Mute 

To Mute all channels type:

|  |
| --- |
| Mute |

### Un-Mute:

*Syntax-* Unmute 

To Un-Mute all channels type:

|  |
| --- |
| Unmute |

### Track Playback and Ducking Attack Time:

*Syntax-* Atk *“track # list”* = *“time”*  *Playback Volume*

Datk *“track # list”* = *“time”*  *Ducking Volume*

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

*“time” =* Time may be specified with local or global variables. Times are in frames, milliseconds, or 1/10 second depending on the TIMEREF setting.

Track values can range from 1- Max Tracks. The default is 0. The attack time is applied for any increase in track volume or un-ducking.

To set the audio attack duration of any Track:

|  |
| --- |
| Atk 1,2 = 3000;*3 seconds in ms mode* |

This sets the attack duration on tracks 1 and 2 to 3 seconds.

By defining variables to represent the channels and ramp value.

|  |
| --- |
| Define ch1 = 1  Define ch2 = 2  Define ramp1 = 30*;1 second in frames mode* |

The equivalent command using variables would be:

|  |
| --- |
| Atk ch1,ch2 = ramp1 |

### Track Playback and Ducking Decay Time:

*Syntax-* Dek *“track # list”* = *“time”*  *Playback Volume*

Ddek *“track # list”* = *“time”* *Ducking Volume*

*Parameters* --*“ track # list” =* Multiple sound tracks may be specified with a ‘,’ delimiter. Track numbers may be specified with local or global variables.

*“time” =* Time may be specified with local or global variables. Times are in frames, milliseconds, or 1/10 second depending on the TIMEREF setting.

Track values can range from 1- Max Tracks. Decay time settings can range from 0(no decay) to 127(12.7 seconds). The default is 0. The decay time is applied for any reduction in track volume or ducking.

To set the decay duration of any Track:

|  |
| --- |
| Dek 5 = 10 |

This sets the audio decay duration on Track five to one second.

### Amplifier Control: (MSII only)

*Syntax-* Amp on/off/stby/mute 

*Parameters* – on = Turn Amplifier on

off = Turn Amplifier off

stby = Place Amplifier in standby mode.

mute = Mute amplifier

To turn the speaker amplifier off type:

|  |
| --- |
| Amp off  |

To turn the speaker amplifier on again type:

|  |
| --- |
| Amp stby   Amp on  |

To avoid a turn on popping, place the Amp in standby before turning it on.

## Clip Based Metaphors

Clip based metaphors simplify managing the parameters when playing an audio file. It can combine many sequence commands in to a single command. The Clip metaphor uses two objects; a sound file and an audio clip that specifies the behavior of the audio playback. There can be more than one clip referencing a single audio file, each having different and unique properties. Clip metaphors allow for the following:

* + “Load”, “Play”, or “Append” a sound without having to manage track assignments, channel assignments or volumes.
  + Allows for more than one clip for each audio file.
  + Track assignments can be made automatically, stored in the clip database and referenced by “clipname.trackname.trackparameter” when needed.
  + Clip entries can also be reference by “clipname.clipparameter”.
  + The Clip database operates on top of the existing command structure and provides a friendlier programming environment.
  + Helps to minimize errors.
  + Clips reside on the flash card along with sequences and audio files.

- Clips simplify playback setup and reduce the complexity of a sequence or event list.

### Load Clip Command:

*Syntax-* Ldclip *“file name”*

*Parameters* - “*file name*” = Clip file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The “.clip” dot extension is also optional. Global and local string variables and Console or DTE are also valid entries for file name.

To load a clip in to memory enter:

|  |  |
| --- | --- |
| Ldclip showsnd  Start C showsnd   Start | ;Loads a clip to memory  ;runs the clip  ;Starts any cued sounds called by the  ;clip |

This command or sequence instruction loads the clip “entrysnd.clip” from the CF card in to memory but does not perform the specified options identified in the clip database. To start the clip, use the “Start C clipname” command (see the Start command). The clip will remain in memory after the clip has finished execution. It can be un-loaded from memory using the “Freeclip clipname” command. Track assignments that are set to zero will not be assigned until the Clip is called using the Start command. After starting the Clip, the track number can be determined using the Tvar function shown below:

|  |  |
| --- | --- |
| Define tr  Define pc  Ldclip showsnd  Start C showsnd  tr = showsnd.t1.track  pc = Tvar tr percent  Stop tr  Stop showsnd.t2.track | ;track variable  ;% complete variable  ;load the Clip  ;start the Clip, track number is set here  ;get the assigned track number  ;get the %complete for track showsnd.t1  ;Stop playback of track showsnd.t1  ;Stop playback of track showsnd.t2 |

### Clip Command:

*Syntax-* Clip *“file name”*

*Parameters* - “*file name*” = Clip file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The “.clip” dot extension is also optional. Global and local string variables and Console or DTE are also valid entries for file name.

To execute a clip enter:

|  |
| --- |
| Clip entrysnd |

This command or sequence instruction loads the clip “entrysnd.clip” and performs the specified options identified in the clip database. Just like the “Ldclip” command, if any of the Clips track assignments are set to zero in the database, then that track will be assigned automatically. The clip will remain in memory after the clip has finished execution. It can be re-started using the “Start C clipname” command or un-loaded from memory using the “Freeclip clipname” command.

### Un-loading a Clip from memory:

*Syntax-* Freeclip *“file name”*

*Parameters* - “*file name*” = Clip file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The “.clip” dot extension is also optional. Global and local string variables and Console or DTE are also valid entries for file name.

To un-load a clip, enter the command:

|  |
| --- |
| Freeclip entrysnd |

The ‘Freeclip’ command or sequence instruction un-loads the clip “entrysnd.clip” from memory. The clip must have been previously loaded using the ‘Ldclip clipname’ or ‘Clip clipname’ commands.

Below is a script example showing use of the ‘Freeclip’ command:

|  |  |
| --- | --- |
| Clip Intro  Ldclip scene1  Ldclip scene2  Ldclip scene3  L1  if Close1 == 1  Start C scene1  endif  if Close2 == 1  Start C scene2  endif  if Close3 == 1  Start C scene3  endif  if Close4 == 1  Goto exitnow  endif  Goto L1  exitnow  Freeclip Intro  Freeclip scene1  Freeclip scene2  Freeclip scene3 | ;Play the Intro Clip  ;Pre-load three show clips  ;Start the 1st pre-loaded clip  ;Start the 2nd pre-loaded clip  ;Start the 3rd pre-loaded clip  ;Go clean up and exit  ;Un-load the Introduction show  ;Un-load the three show clips |

### Clip Metaphor Data Base:

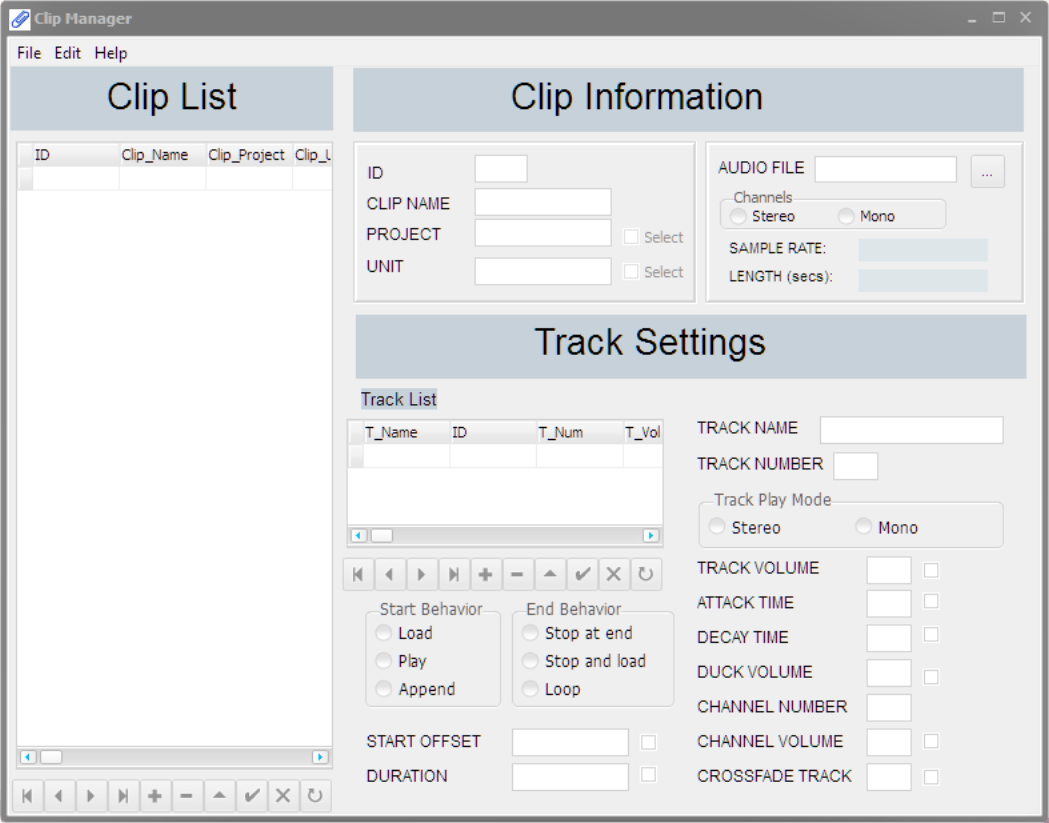
The Clip database uses the standard **©**MicrosoftAccess database (.accdb) format. It can be maintained and managed using Access or the client Clip Manager application (Clipmet). The database consists of two separate relational record sets, the parent Clip List and the child Track List. The two record sets are connected through the “ID” parameter. The Clip Manger client program maintains this relationship. The user must manage this relationship if **©**MicrosoftAccess is used to edit the database.

A blank database is provided with the Clip Manager program. Care should be taken not to modify the database field names or delete any fields. Failure to do so could compromise Clip operations. The database may be duplicated or renamed. The Clip Manager provides the mechanism to manage multiple projects within the same database.

### Clip Manager Program:

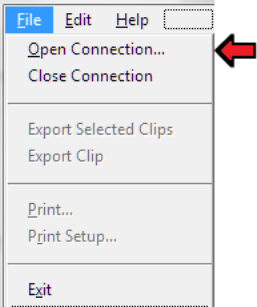
The Clip Manager application provides a user interface for managing a Clip Database. The program does not provide a means for creating new databases but provides the tools to add, edit, and delete records in an existing database. The Clip Manager also provides the important function of exporting the Clips for use by the Min-Sam or rPod units. The exported “.clip” file contains a single Clip List record and its associated track record(s). A Mini-Sam can have two track records and the rPod 8.4 can have up to eight track records. The default name assigned to the “.clip” file is derived from the CLIP NAME field in the Clip List with an appended “.clip”.

The image shown below displays the program after it is opened. At this point, there is no connected database. The next step is to open a connection with an existing database.

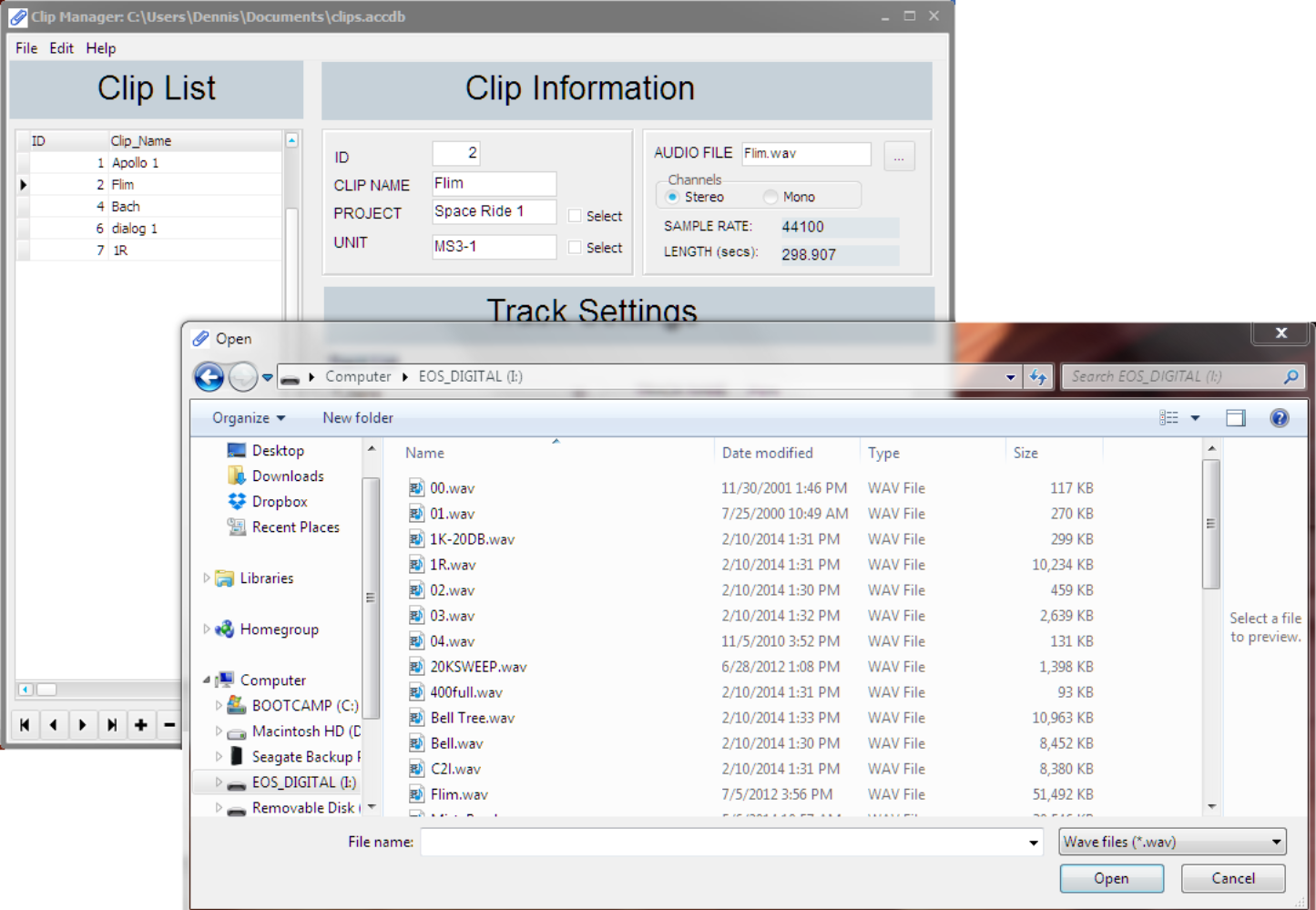


### Connecting to a Clip Database:

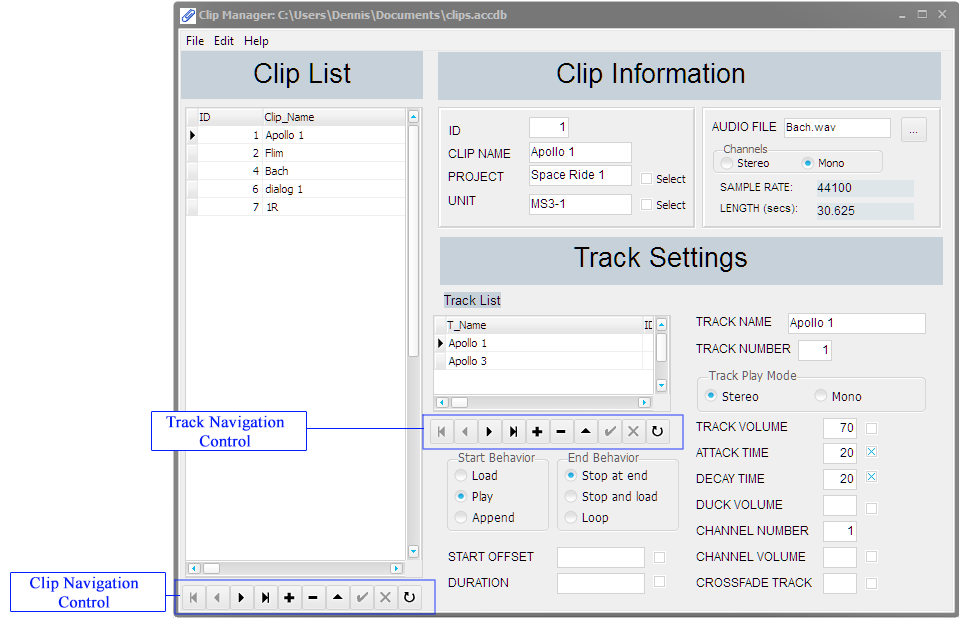
To connect to a clip database, select Open Connection… from the File menu.



From the “Open” dialog, navigate to a “.accdb” database and select “Open”.

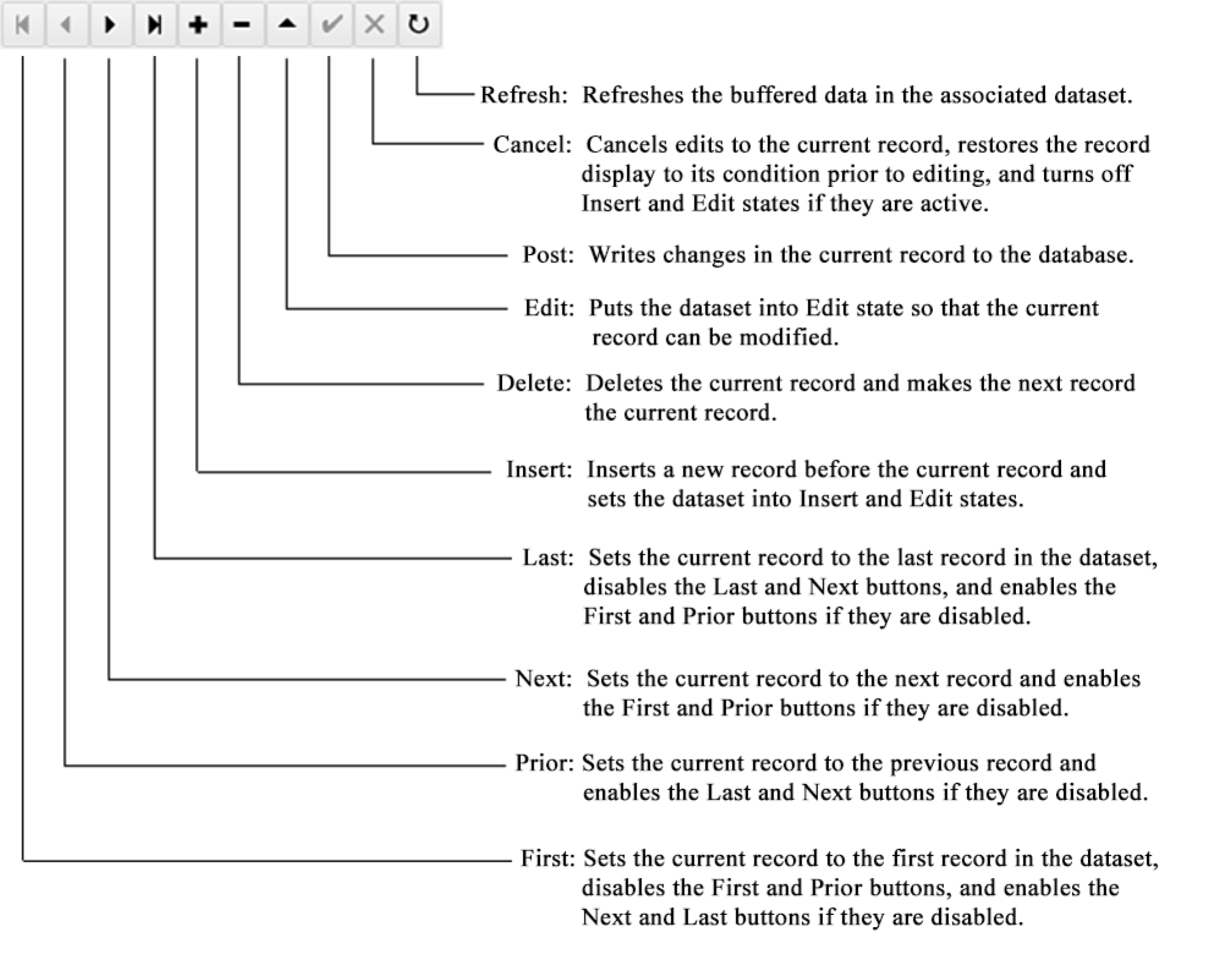


The image below shows an open and connected database. This is indicated by the list of clip records displayed in the far left list box and the database path and name in the title bar.



### Navigating the Clip Database

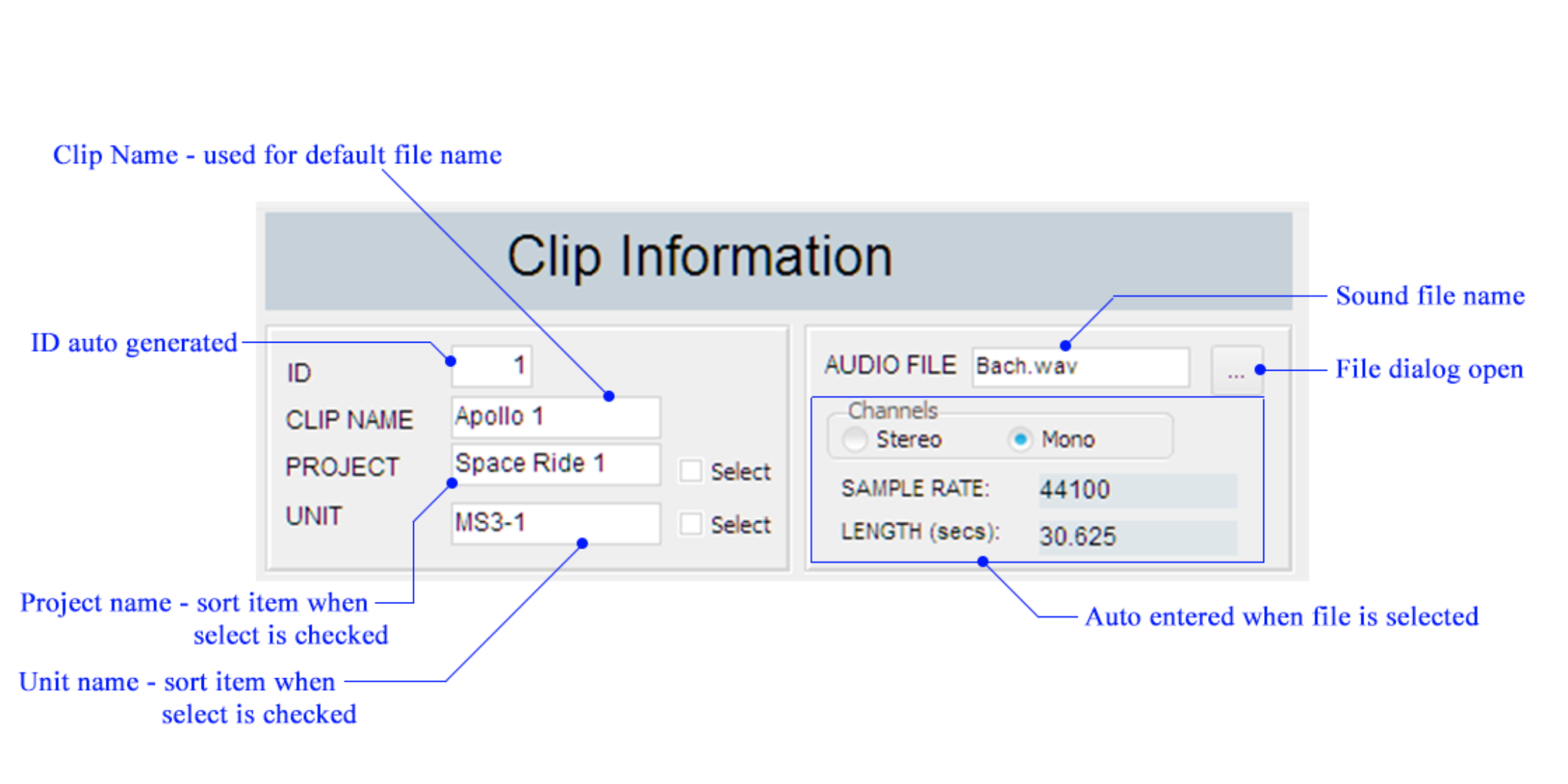
As shown above, there are two navigation controls, one for the Clip list and the other for the Track list. They both perform the same functions on there respective databases.



### Clip Database Fields

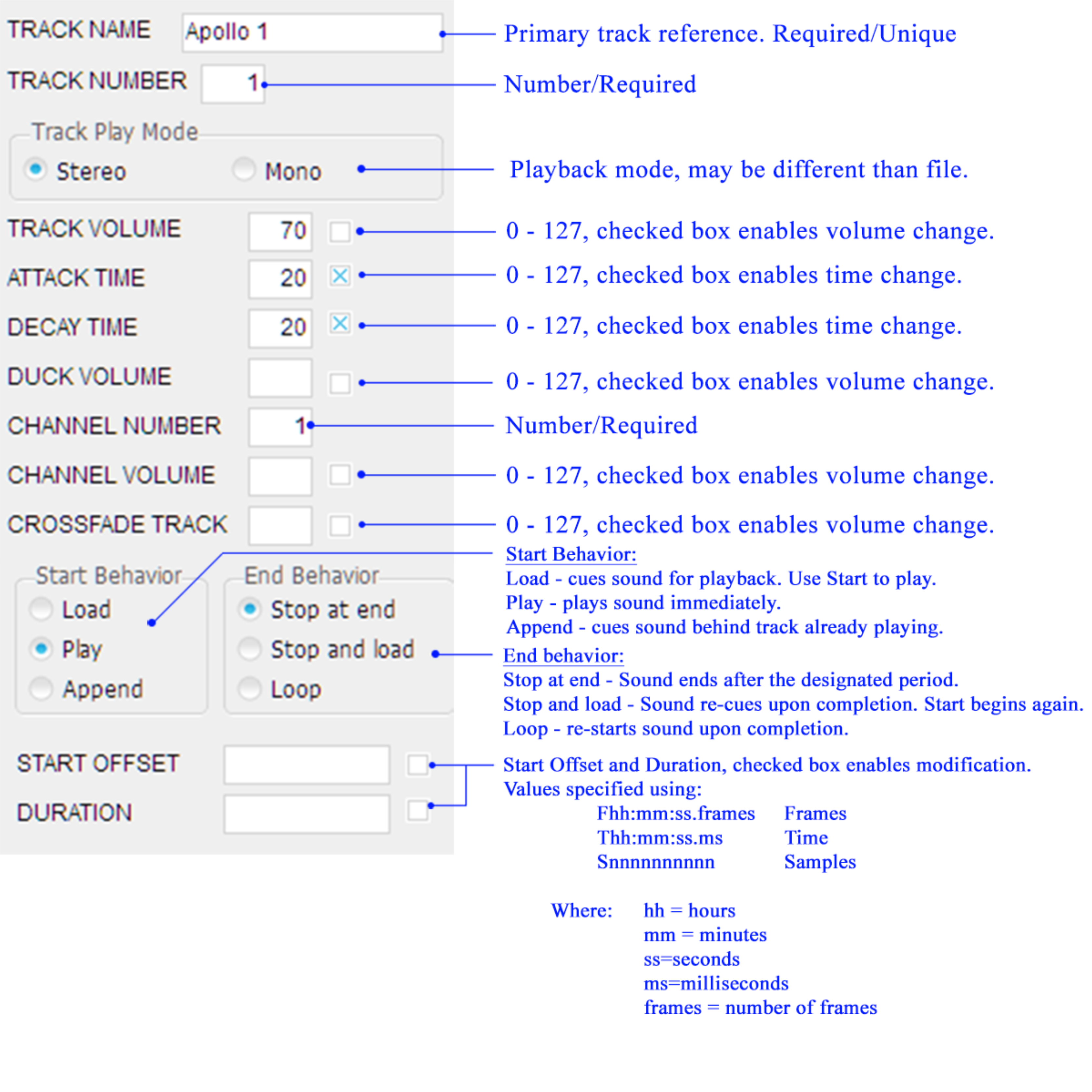
The diagram below shows the fields associated with the Clip database records. The auto-generated ID is assigned by the database and provides a link to the associated Track database records.

The Project and Unit entries have associated Select check boxes. When one or both of the check boxes is selected, the entries shown will be filtered to show only records matching the query in those fields.

When the file dialog open button is pressed, an open dialog is presented to allow selection of the audio file to be played. The Channels, Sample Rate, and Length fields will be automatically entered upon reading the selected file.

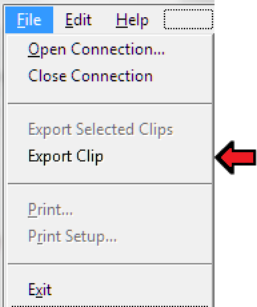
### Track Database Fields

The diagram below shows the fields associated with the Track database records. Many of the fields have a check box option. If a check box is selected, then the field value is applied when the clip is executed except when append is the start behavior. If append is used then the change will occur when the currently playing file completes and the specified file starts. Therefore, if a check box is not selected, then the field value is ignored when the clip is executed.



### Exporting Clips

A clip must be exported to use it in an rPod or Mini-Sam. Select the clip from the clip list and then select “Export Clip” from the “File” menu. A save dialog will appear and a default name will be presented using the “Clip Name” with an appended “.clip”. Select a destination path and Hit Enter to select the default or type in a new name as desired. Once the file has been saved, it can be transferred to the CF media for use in the system.



## DMX/RS485 and Output Control Commands

These commands provide bit, byte, and Dimmer level controls

Channels are numbered 1- maxdmx.

Bits are numbered 0-7.

Control outputs are numbered 1-4096

DMX output values range from 0-255

Time has units of milliseconds or frames depending on the TIMREF settting.

### Bit Level Control:

*Syntax-* Output *“control # list”* on/off/toggle 

*Parameters* --*“ control # list” =* Output control number list. The range is 1 to 4096. Control output numbers may be specified with local or global variables. The Mini-Sam and rPod have the first 16 outputs mapped to the output connectors and correspond to DMX channels 1 and 2. The remaining outputs are mapped to DMX channels 3 – 512.

The following command:

|  |
| --- |
| Output 1,3 on |

turns on control outputs 1 and 3 whilst the command

|  |
| --- |
| Output 4,8,9 off |

turns off control outputs 4, 9 and 9. If the toggle parameter is used then the ouputs specified are set to the alternate state. Off is set to on and on is set to off.

*Syntax-* Dmx *“ch #”*- *“bit #”* on/off 

*Parameters* --*“ ch#” =* DMX output channel # in the range of 1 to maxdmx. Channels may be specified with local or global variables.

*“bit#” =* DMX output bit # in the range of 1 – 8 where 1 = LSb and 8 = MSb. Bit numbers may be specified with local or global variables.

The following command:

|  |
| --- |
| Dmx 5-4 on |

turns on DMX channel 5, bit 4 (4th physical bit) and

|  |
| --- |
| Dmx 1-8 off |

turns off DMX channel 1, 8 (last physical bit)

### Byte Level Control:

*Syntax-* Dmx *“ch #”* = *“output value”* 

*Parameters* --*“ ch#” =* DMX output channel # in the range of 1 to maxdmx. Channels may be specified with local or global variables.

*“ output value” =* DMX output byte value in the range of 0 to 255 where 0 = all bits off and 255 = all bits on. Output bytes may be specified with local or global variables.

The following command:

|  |
| --- |
| Dmx 22 = 255  |

turns on all bits of DMX channel 22 and

|  |
| --- |
| Dmx 14 = 3  |

turns only the first two bits of DMX channel 14 on.

### Dimming Control:

*Syntax-* Dimto *“output value”* in *“time (ms)”* on *“ch # list”* 

*Parameters* –“*output value” =* DMX output byte value in the range of 0 to 255 where 0 = all bits off and 255 = all bits on. Output values may be specified with local or global variables.

*“time” =* Time duration to perform the dimming. Time may be specified with local or global variables. Times are in frames, milliseconds, or 1/10 second depending on the TIMEREF setting.

*“ch# list” =* DMX output channel # list in the range of 1 to maxdmx. Channels may be specified with local or global variables.

The following command:

|  |
| --- |
| Dimto 120 in 1000 on 1,2 |

This programs channels 1 and 2 to dim to a value of 120 in 1 second.

The 120 is the output value (0-255).

1000 is the dimming duration in milliseconds.

The 1,2 references DMX channels 1 and 2.

### DMX/RS485 Output Driver Control:

*Syntax-* Dmx DRIVE

Dmx RELEASE

Enables and disables the DMX/RS485 driver. These commands are used for half duplex RS485 operation or non-standard bi-directional Dmx control. ‘RELEASE’ puts the differential driver output in a high impedance state.

|  |
| --- |
| Dmx DRIVE;enable driver  Dmx RELEASE;disable driver |

## DMX File Playback

The files used for playback are generated using the “Programmer” or “DMXedit” software applications or recorded using the ‘Recdmx’ command. Files can contain one or more channels. The files include the channel assignments to be used.

There are two versions of some of the commands. One version maintains a temporary copy of the file in memory and the other allows the file to remain in memory after playback has completed. The benefit of leaving tracks mounted is that it reduces the valuable CF memory card data access bandwidth necessary for streaming audio playback.

Before DMX playback begins, the system always checks to see if the file specified is already mounted on an existing track. If it is, then the loading stage is skipped and the file is played. Otherwise the file is mounted and then played. When playback is completed the mounted file is either removed or left mounted depending on how playback was specified. Files are left mounted if either the track is or if it was previously mounted to a track.

### DMX file Pre-loading:

*Syntax-* Lddmx *“file name”* on *“track #”* 

*Parameters-* “*file name*” = DMX file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

*Options*- *“ track #” =* DMX track number. Tracks may be specified with local or global variables.

This command pre-loads a file to the specified DMX output(s).

The pre-loading feature can be used to load multiple sound and DMX files from the CF card to the processors sound and control buffers. The sound and control files can then be played using the “Start” command. This method of playing files prevents any time lag in playback due to buffer cuing and forces sample accurate synchronization of multiple sound and control files.

The command ‘Lddmx’ is the similar to the ‘Playdmx’ command in both syntax and operation except that playback does not occur until the “Start” command is received. This can be used in conjunction with the audio ‘Ldsnd’ command. The following series of commands illustrate its use:

|  |
| --- |
| Lddmx light1DMX on 1   Lddmx light2DMX on 3   Lddmx light3DMX  .  .  Start  |

This series of commands pre-loads “light1DMX.wav” on track 1, “light2DMX” on track 3, and “light3DMX”, temporarily, on track 2. Finally, when the “Start” command is reached, the three files begin synchronized playback. When playback is complete, tracks 1 and 3 are left mounted and track 2 is un-mounted.

### Unloading DMX track:

*Syntax-* Unloaddmx *“track # list”* 

Unloaddmx all 

*Parameters-* *“track # list” =* Tracks may be referenced by local or global variable.

The Un-loading feature can be used to release or remove multiple DMX tracks that are waiting to be played. Once un-loaded, they will not be played when a “Start” command is issued.

To un-Load a sound track type:

|  |  |
| --- | --- |
| Unloaddmx 1  | * remove pre-loaded track* |

### Play DMX File:

*Syntax-* Playdmx *“file name”* on *“track #”* 

*Parameters-* “*file name*” = DMX file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

*Options*- *“track #” =* DMX track number. Tracks may be specified with local or global variables.

This command plays a file to the specified DMX output(s).

|  |
| --- |
| Playdmx light1DMX on 1  |

This command plays “light1DMX.wav” on track 1. The track will remain mounted after playback is complete and all subsequent requests to play “light1DMX” will not require access to the CF card. The file remains mounted and can only be un-mounted by:

* ‘Unmount’ command
* Redefinition by another command
* Re-booting command
* Hardware reset

It can be restarted with the following command:

|  |
| --- |
| Playdmx light1DMX |

This is an example for playback of a temporary DMX file:

|  |
| --- |
| Playdmx light7DMX |

Assuming the file has not already been mounted, this command will mount the file named “light7DMX.wav into the first available track When the track has completed, the file is removed from memory or un-mounted.

### DMX Looping:

*Syntax-* Loopdmx *“file name”* on *“track #”* 

*Parameters-* “*file name*” = DMX file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

*Options: “ track #” =* DMX track number. Tracks may be specified with local or global variables.

This command loops a file to the specified DMX output(s).

The command ‘Loopdmx’ is the similar to the ‘Playdmx’ command in both syntax and operation except that when the end of file is reached, playback continues from the beginning of the file. A looping file may be stopped using the ‘Stopdmx’ command. Ex.

|  |
| --- |
| Loopdmx light1DMX on 1 |

### DMX File Mounting:

*Syntax-* Mount *“file name”* on *“track #”* 

*Parameters-* “*file name*” = DMX file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

*“ track #” =* DMX track #. Tracks may be specified with local or global variables. Tracks may be specified with local or global variables.

This command mounts a file to the specified DMX track

Before a file is mounted, the system always checks to see it is already mounted on an existing track. Once the file is mounted, it remains in memory until either an ‘Unmount’ command is received or the track is redefined. Note the following command does not cause playback of the specified file.

|  |
| --- |
| Mount light1DMX on 5  |

This command mounts the file “light1DMX.wave” on track 5. Then, to playback the mounted file type:

|  |
| --- |
| Playdmx light1DMX on 2  |

This will play the file “light1DMX.wav” already mounted on track 5 to DMX channel 2. Note that this second command requires no access to the CF memory card because the file was already mounted. When playback is completed, the file will remain mounted on track 5.

### Un-mounting DMX Files:

*Syntax-* Unmount *“track # list”* 

*Parameters-*  *“ track #list” =* A list ofDMX track numbers. Tracks may be specified with local or global variables.

Un-mounts a file(s) from the specified DMX track(s)

Note the following command is not accepted when the mounted track(s) are busy.

|  |
| --- |
| Unmount 1,2,3,4  |

To un-mount all DMX tracks enter:

|  |
| --- |
| Unmount all |

### Stopping DMX Tracks:

*Syntax-* Stopdmx *“track # list”* 

*Parameter-*  *“track # list” =* List of tracks that are actively playing DMX data. Tracks may be specified with local or global variables.

To stop playing DMX tracks enter:

|  |
| --- |
| Stopdmx 1,2 |

This will stop tracks 1 and 2.

### Stopping all DMX Tracks:

*Syntax-* Stopdmx all 

To stop playing all DMX and Sound tracks enter:

|  |
| --- |
| Stop all |

### Positioning DMX Playback:

*Syntax-* Posdmx *“track# list”* to F/T *“position reference”* 

Skipdmx *“track# list”* to F/T *“position reference”* 

*Parameters-* *track # list” =* Output DMX track. Tracks may be specified with local or global variables.

These commands are used to reposition DMX playback to a different place within the control file. The ‘Posdmx’ command positions playback with respect to the beginning of the File and the ‘Skipdmx’ command positions playback with respect to the current playback position. The file may either be playing or pre-loaded. The commands expect a list of tracks in comma-delimited format and a position reference. The position reference can be expressed in three different ways. An “F” denotes frames reference where each frame is 1/30th second. The “T” indicates time in units of milliseconds. When using the skip command, the position reference may be either positive or negative.

To position Tracks 1 and 3 10secs into the sound enter:

|  |  |
| --- | --- |
| Posdmx 1,3 to F300   Posdmx 1,3 to T10  |  *Example 1*   *Example 2* |

To adjust the current playback position back 63 seconds type:

|  |  |
| --- | --- |
| Skipdmx 1 F-1890   Skipdmc 1 T-1:03  |  *Example 1*   *Example 2* |

## DMX File Capture

### Record DMX:

*Syntax-* Recdmx *“file name”* from *“ch # (start)”* to *“ch # (end)”* 

*Parameter* – “*file name*” = DMX file name to be save to the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

“*ch #” =* Input DMX channel #.

This command records the specified DMX channels from the DMX input.

To record DMX input channels 1 –10 type:

|  |
| --- |
| Recdmx light10DMX from 1 to 10  |

### Pre-load Record DMX:

*Syntax-* Ldrecdmx *“file name”* from *“ch # (start)”* to *“ch # (end)”* T *“linked sound track #”* 

*Parameter* – “*file name*” = DMX file name to be save to the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The dot extension is also optional. Console or DTE is also valid.

“*ch #” =* Input DMX channel #.

TFlags linking recording to a specific sound file. The linked sound track number must follow this flag.

“*linked sound track #” =* Sound track to link DMX recording to. When the sound file ends, recording is terminated.

The ‘Ldrecdmx’ command prepares the system for synchronous recording with playback of other audio or DMX tracks. Recording will begin when the “Start” command is received. The following shows an example:

|  |
| --- |
| Lddmx light1DMX on 11   Ldsnd Flim on 1 T1  Ldrecdmx light10DMX from 1 to 10 T1   .  .  Start  |

The first three commands prepare the DMX file “light1DMX” for playback on DMX channel 11, sound file “Flim” for playback on sound channel 1, and the DMX file “light10DMX” for recording of DMX inputs 1-10. Also, DMX recording is linked to soundtrack 1 (“T1” part of the ‘Ldrecdmx’ command) and will stop DMX recording when “Flim” completes playback.

### Starting Pre-loading Record DMX action:

*Syntax-* Start 

The ‘Start’ command will begin synchronous playback and recording.

### Stop Recording DMX:

*Syntax-* Stop all 

Stopdmx all

To stop recording type one of the following:

|  |
| --- |
| Stop all  or  Stopdmx all  |

## Ethernet

### Configuring Ethernet:

*Syntax-* Config ENET *<“ip address”>* : <*“port”* >/<*”subnet address”>*

*Parameter* – “*ip address*” = IP addresses are a standard 32 bits long. The format uses dot notation such as “192.168.1.10”. This parameter is optional

“*port” =* The port designation is used to set the UDP port destination filter. This parameter is optional.

“*subnet address” =* Subnet addresses are a standard 32 bits long. The format uses dot notation such as “255.255.255.0”. This parameter is optional

The ip address and port assignments are stored on the resident Compact Flash Card in the “.ini” file. This file may be edited using an editor or by issuing the “Config” command. To change the ip address enter:

Config ENET 192.168.1.5

The destination port assignment can be set independently by typing:

Config ENET :1000

Both the ip address and port can be changed together using:

Config ENET 192.168.1.5:1000/255.255.0.0

### Ethernet Command Status Reporting:

*Syntax-* Verbose ENETon/off

This command enables or disables command feedback to the requester (see ‘Sendcmd’). The default is off = no command status, on = command status transmitted to sender. To enable feedback enter:

Verbose on

To disable feedback enter:

Verbose off

### Configuring FTP Password:

*Syntax-* Config PASSWORD *“string”* 

*“string” =* string constant.

The command ‘Config PASSWORD’ changes the file transfer protocol password. The default password is “SKE”.

### Configuring UDP:

*Syntax-* Config UDP Manual/Auto(default)

The command ‘Config UDP Auto’ causes all raw UDP data to be sent to the Console. ‘Config UDP Manual’ should be set when using the ‘Read UDP’ or ‘Peek UDP’ commands.

### IP Referencing:

*Syntax-* Bind *“ip address”* to *“bind number”* 

*Parameter* – “*ip address*” = The IP address of the network device to bind to. The format uses dot notation such as “192.168.1.10”. This parameter is optional

“*bind number” =* The bind number is an arbitrary numerical assignment from 1 to 255.

This function assigns the IP address of an external network device to a short hand reference number. All devices on the network must have a unique IP address. The reference number is used by the commands ‘Send’ and ‘Sendcmd’ to identify the destination device. The source device is determined when a UDP message is received using the ‘Recv’ command and is contained in the system variable ‘?EORIG’. The ‘?EORIG’ variable represents the bind address of the last received message. Before sending commands or data from unit to unit, a bind assignment must be established. To bind an IP address to 1, type:

Bind 169.254.144.60 to 1

A network broadcast bind can be established by setting the last octet to 255. For example:

Bind 169.254.144.255 to 255

In this case, 255 is the network broadcast bind number. Each bind number can only reference one IP Address at a time but are locally defined and not visible to external devices.

### UDP Control:

*Syntax-* Sendcmd *“bind number”*: *“constant string”*

Sendcmds *“bind number”*: *“variable string”*

*Parameter* – “*bind number” =* Message destination reference. The bind number points to a valid IP address on the network. Range = 1 to 255. (See Bind command)

“*constant string” =* This is a constant string representation of a command to be executed by the receiving device.

“*variable string” =* This is a local, global, or public string variable containing a valid control command to be executed by the receiving device.

The ‘Sendcmd’ and ‘Sendcmds’ commands can be used to control other Mini-Sam or rPod units on a network through the Console, DTE, or DMX port or through a sequence program. The bind number must have been previously established using the Bind command. Any command that can be issued via the Console can be issued using this command. This includes playing or stopping audio or DMX files, launching sequences, setting or clearing outputs, changing global or local variables, etc. This version allows for passing the command in as string allowing for the variable construction of commands.

Note that, when using the ‘Sendcmds’ version with a rPod8.4 on the P1 side, you must use a Public string variable. The P2 side can use either a public, global, or local string variable.

The following example sets the state, loads an audio file and DMX file, and then plays the two files on the referenced remote device:

|  |
| --- |
| Sendcmd 2:state=2  Sendcmd 2:Ldsnd flim on 1  Sendcmd 2:Lddmx lighting on 1  Sendcmd 2:Start |

The ‘Sendcmd’ command can be simulated using a UDP terminal like Hercules from HW-group.com by omitting the ‘Sendcmd’ and bind number. So, to send a command from a device other than a MiniSam or rPod use the following:

|  |
| --- |
| :state=2  :Ldsnd flim on 1  :Lddmx lighting on 1  :Start |

The following example shows the setup for a Sendcmds:

|  |
| --- |
| Define $a ;string to carry command  Bind 192.168.1.255 ;create a bind to broadcast  $a = “Play flim on 1 T1” ;construct a valid command  Sendcmds 1:$a ;Send the command over ethernet |

### UDP Sending Data:

*Syntax-* Send *“ bind number”.<”packet number”>-<“ message type* “>|*“string or variable list”*

Printe *“ bind number”|”* *string or variable list”*

*Parameter* – “*bind number” =* Message destination reference. The bind number points to a valid IP address on the network. Range = 1 to 255. (See Bind command)

*“packet number” =* optional packet number (32 bits) assigned and managed by the user. (0=default)

*“message type” =* optional and arbitrary message type (32 bits) establishing the type and format of the data contained in the list. (0 = default)

**| =** thedelimiter separating the packet description from the packet variable list.

“*string or variable list” =* This is and arbitrary comma delimited list of variable, string, or constant data.

The ‘Send’ command can be used to exchange common data between UDP compatible devices. The bind number sets the destination for the message must have been previously established using the ‘Bind’ command. The pre-arranged message content is specified by the message type field. The ‘Send’ command is used in conjunction with the ‘Recv’ command. For example:

|  |
| --- |
| Send 25.103-2| x, $a, "This is a String", 0xaf4b |

“25” = is the Destination bind number, “.103” = packet number and “-2” = data type

The ‘Printe’ command is like the ‘Send’ command but does not contain any packet number or message type data in the transmitted UDP packet. It is typically used to send data to devices other than the Mini-Sam or rPod.

|  |
| --- |
| Printe 25| x, $a, "This is a String", 0xaf4b |

### Receiving Data using UDP Protocol:

*Syntax-* Recv *“string or variable list”*

Read UDP

Peek UDP

Config ENET Term *“Term method”* 

*Parameter* – “*string or variable list” =* This is and arbitrary comma delimited list of variable, string, or constant data.

The ‘Recv’ command assigns data received to resident string or numeric global or local variables. When a UDP packet is received, the packet is scanned for a ‘|’ character, and if found, indicates that it is a data exchange request. The ?ETYPE, ?EORIG, ?ECOUNT, ?EPKT and ?ENET system variables are then updated to reflect the packet characteristics. These are defined as follows:

?ETYPE = User defined configuration of variable list. 0 - 65535

?EORIG = Source bind number from 1 to 255.

?ECOUNT = Number of string or number variables received

?EPKT = Packet number from sender . 0 to 65535

?ENET = UDP buffer ready flag 0=no data, 1=Receive variables, 2=raw UDP data

The ‘Read UDP’ command affects the ?ENET flag and will indicate that raw UDP data has been received. It will also identify the source or sender using the ?EORIG variable. A Read is completed when ?ENET = 2. Any data in the “UDP” system variable will be replaced by the new packet data and then that packet is discarded.

The ‘Peek UDP’ command is the same as the Read UDP command except the packet is not discarded and remains the current active packet buffer. Any subsequent Read UDP or Peek UDP command will return the same Packet information as before.

When receiving data the string termination character can be changed from the default NULL character to the following:

Config ENET Term NULL ;default

Config ENET Term CR ;carriage return

Config ENET Term CRLF ;carriage return + line feed

Config ENET Term TAB ;Tab character

Config ENET Term NONE ;no termination

The following is an example of the Recv command:

|  |
| --- |
| ; message type 1 = ack(1)/nack(0). Message type 2 = variable packet  if ?ENET == 1 ;is there a UDP data packet  if ?ETYPE == 2 ;yes, is the message type a variable packet?  Recv x,$a,$b,z ;yes, grab the data  y = ?EORIG ;y = who sent the data  Send y|1 ;send back an ack(1)  endif  endif |

## System Commands

*Note: There are also the system constants ?P and ?T that are used in sequences to return the playback status. The commands and constants are two separate items.*

### Re-Boot:

*Syntax-* Boot 

Boot causes the system to re-initialize. It is similar to hitting reset and can be used to restore system operation in the event of a problem.

### Setting the Time Base:

*Syntax-* Config TIMEREF ms/frames/mixed

*Options*- ms = This sets the basic units of time measurement to milliseconds.

frames = This sets the unit of measurement to frames or 1/30 of a second.

mixed = Legacy mode using a mixture of time bases. (see table below)

When using this setting, it is best to set it through the Console rather than in the startup sequence. A CF card should be installed in the unit when the command is issued because the unit will attempt to save the selection to the \*.ini file. This setting stays with the CF card and anytime the unit is powered up, the setting will be restored to the unit.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Mode**  **ms** | **Mode**  **frames** | **Mode**  **mixed** |
| tmr | ms | frames | frames |
| TIMER | ms | frames | frames |
| Wait | ms | frames | ms |
| Event Lists | ms | frames | frames |
| Embedded Timer Points | ms | frames | frames |
| Cues | ms | frames | frames |
| Track Attack | ms | frames | 1/10 second |
| Track Decay | ms | frames | 1/10 second |
| Duck Attack | ms | frames | 1/10 second |
| Duck Decay | ms | frames | 1/10 second |
| Sound Positioning | Mixed | Mixed | Mixed |
| DMX Dimming | ms | frames | ms |

### Cosmic variable commands:

*Syntax-* Update

Reload

Remove *“variable list”/*all

*Options*- *“variable list”* = List of variable to convert to global.

Cosmic numeric and string variables are created using the ‘Define Cosmic <variable>’ command. They will not automatically be saved to the compact flash memory card after this creation. This only happens when an ‘Update’ command is issued. The command causes the current value of all cosmic variables to be saved to the boot file (boot.ini) located on the compact flash card.

When the Mini Sam or rPod powers up or boots, it refreshes the variable list from a boot file. It does this by calling the ‘Reload’ command. Strings and variables listed in this file will be automatically defined and values assigned from the file. Variables listed in the boot.ini file will be restored if the ‘Reload’ command is used from the command line or a sequence program. This can cause values to change from their current value

The Remove command can be used changes a boot variable to a regular global variable. Again, an ‘Update’ command must be issued to reflect the change to the boot.ini file. The ‘Remove’ command can use a list of variable or the “all” modifier can be used to remove all Cosmic variables.

### Start Playback:

*Syntax-* Start <T *“track # list”* D *“channel # list”* C “clip name list”>

*Options*- T = Flags the selective starting of pre-loaded sound tracks (optional). The ‘T’ flag must be followed by a track list. If the ‘T’ option is not specified then all pending or pre-loaded sound tracks are started.

*“track # list” =* This is a list of sound track to be started.

D = Flags the selective starting of pre-loaded DMX channels (optional). The ‘D’ flag must be followed by a list of DMX channels. If the ‘D’ option is not specified then all pending or pre-loaded DMX files are started.

*“channel # list” =* This an optional list DMX channels to be started .

C = Flags the selective starting of pre-loaded Clips. Unlike the ‘T’ and ‘D’ flags, it must be present to start a clip. Also, it must be followed by a list of resident Clips. Clips that are loaded in memory but not listed in the command will not be started.

*“clip name list” =* This not optional and is case sensitive. This uses clip names and not clip file names.

The pre-loading feature is used to load multiple sound, DMX and Clip files from the CF card to the processors sound, control, and clip buffers (see ‘Ldsnd’, ‘Lddmx’ , and ‘Ldclip’ commands). The sound, control, and clip data can then be played using this command. For sound and DMX files, this method of playing files prevents any time lag in playback due to buffer cuing and forces frame accurate synchronization of multiple sound and control files.

To start all pre-loaded sound tracks and DMX channels enter:

|  |
| --- |
| Start |

To start specific pre-loaded sounds, enter:

|  |
| --- |
| Start T 1,3,5 |

Make sure to include the sound tracks that will fade out when starting pre-loaded sounds that will crossfade with other tracks. Here is an example:

|  |
| --- |
| Ldsnd sound2 on 1 T2 X1  Start T 1,2, |

To start specific pre-loaded sounds and DMX channels, enter:

|  |
| --- |
| Start T 1,3 D 1,2 |

### Query Playing Tracks:

*Syntax-* ?P

The ‘?P’ command prints the playback status of the system. If no tracks are playing then the following is returned:

|  |
| --- |
| 0 tracks playing: |

Otherwise, the status would return the number of total tracks playing followed by the itemized number. Ex/

|  |
| --- |
| 2 tracks playing:1,2 |

### Query Track Status:

*Syntax-* ?T *“track # list”* 

The following command:

|  |
| --- |
| ?T 1,2 |

prints the track status of all specified tracks. If no tracks are playing then the following is returned:

|  |
| --- |
| T1:UNUSED  T2:UNUSED |

Otherwise,

|  |
| --- |
| *T1: Playing Stereo on channels 1 and 2*  *Gain 127, Nom 127, Duck 64, Atk 4.4s, Dek 4.4s*  *File:a:\Apollo 13.wav, 9.45 percent complete*    *T2: Playing Stereo on channels 3 and 4*  *Gain 127, Nom 127, Duck 64, Atk 0.0s, Dek 0.0s*  *File:a:\Flim.wav, 3.36 percent complete* |

### Printing:

The ‘Print’ command can be used to display user or system status to any serial communication port. It is a handy tool for debugging sequence programs. It can be used to determine the state of input triggers or DMX outputs or input text strings from the Console and DTE ports. The Print command will insert a CR/LF character before the variables are printed and it succeed them with a CR/LF and drive prompt (a:>). The lower level printing command ‘Print #’ inhibits the addition of these characters and only outputs the variables specified.

*Syntax-* Print*(port)* *“variable list”* 

Print*(port)* # *“variable list”* 

*variables-* numbers (decimal/hex), local variables, global variables, and strings (quote delimited, Console, or DTE). By specifying the context prior to printing, the local variables for a specific sequence can be printed. See the ‘Context’ command. Placing a ‘#’ character before the variable causes it to be printed as an ASCII code instead of a numeric ASCII characters.

(port) = optional port definition. If n is not specified then the current port is used.

c = Console, d = DTE, e = Ethernet, x=DMX port (serial mode), m=MIDI port (serial mode), n = Network (Programmer 2 command window).

To print a local variable enter:

|  |  |
| --- | --- |
| Context = 1   Define x   x = 5   x = x + 3   Print “x = “, x   *x = 8* |  *Sets the context to sequence 1*   *Creates a variable x*   *Assigns the constant 5 to x*   *Add 3 to x*   *Print the value of x*   *Displayed value from the unit* |

Example showing hexadecimal use:

|  |  |
| --- | --- |
| x = 0x14   Print “0x14 is “, x   *0x14 is 20* |  *Assigns the constant 0x14 to x*   *Print the value of x*   *Displayed value from the unit* |

This example shows printing using the ‘Print #’ version of the ‘Print “Hello”’ command. The line feed (10) and carriage return characters (13) are used here:

|  |
| --- |
| Define LF = 10   Define CR = 13   Print # #CR, #LF, “Hello”, 10, 13, “a:>”  |

When using the ‘Print #’ version of this command, all **numeric constants will be printed as ASCII control characters**. To print a specific value, assign the constant to a variable and Print # the variable.

Adding an optional port definition will redirect the printing to a specific port.

|  |  |
| --- | --- |
| Printd “Hello”   Printc “Hello”  |  *Prints to the DTE port*   *Prints to the Console port* |

### Monitor Serial Port:

*Syntax-* Monitor Console/DMX/DTE/Midi on/off  *(rPod8.4 rev1.2 only)*

Monitor Console/DMX/DTE on/off  *(rPod8.4 rev1.0a only)*

Monitor Console/DMX/ on/off  *(MS2e)*

Monitor all off 

The ‘Monitor’ command enables or disables monitoring of the specified serial ports transmit and receive signals.

The MS2e units put Tx on Status 1 and Rx on the Status 2 indicator. The default at startup is to monitor the Console.

The rPod units put Tx for the Console/DMX on Status 1, Rx for the Console/DMX on Status 2, Tx for the DTE/Midi on Status 3, and Rx for the DTE/Midi on Status 4. It is possible to monitor the Console and DTE ports simultaneously.

To monitor and use the DMX port in RS485 mode, enter the following:

|  |  |
| --- | --- |
| Config DMX RS485   Monitor DMX on  |  *Set the DMX port to 485 mode*   *Monitor the DMX transfers* |

To restore monitoring of the Console port, enter the following:

|  |  |
| --- | --- |
| Monitor Console on  |  *Set the DMX port to Console mode* |

### Configure Serial Port:

*Syntax-* Config Console/DTE/DMX = *“baud rate”, “#bits”, “#stop bits”, “parity”* 

Config Midi = *“baud rate”, “#bits”, “#stop bits”, “parity”*  *(rPod8.4 rev1.2 only)*

Config Console/DTE Term = *“character list”*

Config Midi Term = *“character list”* *(rPod8.4 rev1.2 only)*

Config Console/DTE Term on/off

Config Midi Term on/off *(rPod8.4 rev1.2 only)*

Config DMX Standard/RS485 *(MS2e and rPod8.4 rev1.2 only)*

Config Midi Standard/RS232 *(rPod8.4 rev1.2 only)*

Config DMX Receive on/off

This command is used to modify operating characteristics of the RS-232 ports(s). The console, DTE port defaults to 115,200 baud, 8 bits, 1 stop, and no parity.

The DMX port on the MS2e and rPod8.4 rev1.2 can be configured as a RS485 port providing an alternate command interface. The Midi port on the rPod8.4 rev1.2 can also be configured as a standard RS-232 port.

When the DMX is set to RS485 mode the DMX receiver should be turned off. The DMX input receiver is turned off by default. To enable the DMX receiver use:

|  |
| --- |
| Config DMX Receive on |

And to disable the receiver use:

|  |
| --- |
| Config DMX Receive off |

To modify the Console port for 9600 baud type:

|  |
| --- |
| Config Console = 9600 |

To change the port configuration to 7 bits and 2 stop bits type:

|  |
| --- |
| Config DTE = 115200, 7, 2, 0  |

To change the Console Termination string to a linefeed and carriage return type:

|  |
| --- |
| Config Console Term = 10, 13  |

To disable the port input string termination type:

|  |
| --- |
| Config Console Term off  |

To restore port input string termination to the default type:

|  |
| --- |
| Config Console Term = 13   Config Console Term on  |

### Null Character Substitution:

*Syntax-* Config Console/DTE nullsub on/off 

This command is used to modify operating characteristics of the RS-232 ports(s). The console, DTE port port can receive any data but system strings interpret null characters as the string termination. This can truncate data prematurely. To solve this issue, null character substitution will convert any incoming null values (0x00) to spaces (0x20). To enable substitution on the DTE port type:

|  |
| --- |
| Config Console nullsub on  |

### Read Serial Port:

*Syntax-* Read Console/DTE #/&“*count”*>

Read Console/DTE Lock

Read Console/DTE Release

*Parameter* – “*#*” = Optional number conversion flag. If this flag is included, a string representing the decimal numerical value is returned. When using this parameter the maximum count is 4. This parameter is for the term off mode only.

*“&” =* Optional hexadecimal number conversion flag. If this flag is included, a string representing the hexadecimal value is returned. This parameter is for the term off mode only.

“*count=* Number of characters to read from the buffer. This parameter is for the term off mode only.

Lock This flag causes the Console to be locked in the Read mode and all command line processing is disabled.

Release This flag cause the Console to be released from the lock mode after a Read and command line processing will be subsequently enabled.

This command is used to capture an input string from the specified serial port. The captured string is held in a special string variable of the same name. For the Console port the string is “Console” and for the DTE port it is “DTE”.

This example illustrates use of the command:

|  |  |
| --- | --- |
| Read Console   Hello  Print Console   *Hello* |  *Displayed value from the unit* |

If string termination is “on” then the ‘Read’ command ignores the “count” parameter. The system reports the receive status through the ‘?Console’ or ‘?DTE’ variable where 0=termination not detected and 1=termination found. Data is transferred from the input buffer to the system variable only after the termination is detected.

If string termination is “off” then the ‘Read’ command will immediately read “count” characters from the input buffer to the “?Console” or “?DTE” system variable. These variables should be checked to verify the count before issuing the Read command. Otherwise, the returned string may be shorter than expected. If the buffer contains more than “count” characters then only “count” character are transferred from the buffer. The following shows an example of this feature:

|  |  |
| --- | --- |
| Config Console Term off  J1  x = ?Console   if x >= 5  Read Console 5   Print Console   endif  Goto J1 |  |

### Peek Serial Port:

*Syntax-* Peek Console/DTE #/& “*count”*

*Parameter* – “*#*” = Optional number conversion flag. If this flag is included, a string representing the decimal numerical value is returned.

*“&” =* Optional hexadecimal number conversion flag. If this flag is included, a string representing the hexadecimal value is returned.

“*count=* Number of characters to read from the buffer.

This command is the same as ‘Read Serial port’ except the contents of the input buffer are not removed. Only subsequent ‘Read Serial port’ access to the buffer will remove its contents.

### Silent Mode:

*Syntax-* Silent <Console, DMX, DTE, MIDI, VMSKE/all> on/off

on = suppress all status to the specified ports.

off = enable all status to the specified ports.

This command is used to enable and disable all status data printed to the specified port. Ports can be individually specified or as a group. This affects all prompts, error messages, splash screen info but not information specifically directed to them. All ‘Print’ commands will operate as normal. Use this command to prevent the transmission of spurious data to the port that could cause faulty operation of a connected device. The silent mode is saved to the “\*.ini” file and will be restored at power up.

To place the DTE port in the silent mode type:

|  |
| --- |
| Silent DTE on |

### Command feedback Enhancement:

*Syntax-* Verbose on/off 

on = Enables command feedback from the active port.

off = Disables command feedback on the active port.

This command is used to enable and disable command feedback from the controller to the terminal. This is useful when implementing custom serial control or adapting to an existing serial protocol. The default is off at power up.

### Install Operating System:

*Syntax-* Install *“file name”* 

*Parameters-* “*file name*” = Operating system file name on the specified CF card. The prefix drive name, “a:\” or “b:\”, is optional. The “.ldr” dot extension is also optional.

This command is used to install a new operating system. Installation of the boot jumper or setting the boot switch is not required. Once the command starts, removal of power to the system will result in the improper installation of the software and the “BootLoader” method of installation must be used to restore proper operation. Consult the Operating manual for a complete description of the procedure.

### Controlling Status LEDs:

*Syntax-* Status *“output list”* on/off 

*Parameters-* “*output list*” = A list of outputs to be turned on or off. This includes {all, 1, 2, 3, 4, 5, P, A}. Outputs 1-4 correspond to Status LEDs 1-4. Output P is the Processor OK display and output A is the Audio OK display. Specifying “all” turns on or off all displays.

### Configuring Triggers:

*Syntax-* Config TRIG *“trigger list”* = *“de-bounce time”* 

*Parameters-* *“trigger list” =* This is a list of triggers.

*“de-bounce time” =* time in miliseconds.

This command is used to set the de-bounce time for triggers specified in the trigger list. The system starts up with a default de-bounce time of 20 milliseconds. Reducing the de-bounce time will improve system performance increase trigger sensitivity. Increasing the de-bounce time decrease trigger sensitivity and degrade system performance.

To decrease the de-bounce time to 2 milliseconds for triggers one and two, type:

|  |
| --- |
| Config TRIG 1,2 = 2  |

## Logging

Logging commands are provided to simplify recording of device activity to the CF card. Previously, logging was accomplished using standard file commands through sequence programming. The logging feature will automatically log all commands, responses, and I/O to file. It can create a file with the system date (month, day, hour) as part of the name. The format of the file name depends on the mode.

LOGxxxxxx.txt

If the “replace” option is selected then the previous file will replaced by a new one when the roll over replace period occurs. The replace rate can be set with the mode modifier and allows for hour, day, or months’ worth of logging before the file is erased and a new log file is over written. A timestamp can be prefixed to the log information using the timestamp modifier setting. Log data can be printed to the console using the ‘List’ command. Because file commands are used to implement logging, an “update” command is provided to force the flushing of the log write buffer. This command should be periodically issued when using the logging feature.

### Log Activation:

*Syntax-* Log on 

This command turns on system logging. Logging does not automatically turn on verbose operation from the command input ports and must be enabled/disabled using a separate command. The ‘Log replace’ and ‘Log mode’ commands must be set before the ‘Log on’ command is issued. The ‘Log timestamp’ command can be changed at any time.

### Log Deactivation:

*Syntax-* Log off 

This command turns off system logging. A ‘Log update’ command does not need to be sent prior to the “Log off” command as it will be performed automatically as part the command.

### Log Update:

*Syntax-* Log update 

This command forces the flushing of the log write buffer so that in the event of power loss, the file will be up to date. It should be sent periodically in a sequence program. The ‘Log off’ command will automatically send an ‘Log update’ command.

### Log File Clearing:

*Syntax-* Log clear 

This command clears the current log file. Logging can be enabled or disabled. However, the log mode must be the same as when the original log file was created.

### Log Listing:

*Syntax-* Log list 

This command lists the current log file to the command input device requesting the printout. Logging can be enabled or disabled. Logging is temporarily disabled when the listing is printed.

### Log Mode:

*Syntax-* Log freq none/hour/day/month(default)

This command sets the file creation rate. If none is selected then additional files will not be created beyond the initial file. The hour/day/month options will create a new file upon the interval transition that is selected. If the replace mode is set then the previous file will be erased, included the recorded data, before the new file is created. The file name will reflect the mode setting.

To set the Log mode to hour, enter the following:

|  |
| --- |
| Log freq hour |

### Log file handling:

*Syntax-* Log replace true/false(default)

This command defines how log files are handled by the system. If true is selected then the file is erased before a new file is created. If false is selected then the file is left on the CF card and a new file is added upon the interval transition that is selected.

### Log timestamp:

*Syntax-* Log timestamp on/off(default)

This command determines if a timestamp is placed before each log entry. This command can be sent at any time.

## Sequence Programming Commands:

A powerful capability of the system is programmable logic control. This provides for full automation and advanced control of the unit. It can be used to sense trigger inputs or read the serial ports and then respond by playing audio or DMX control programs in a timed manner. Programs, called sequences, can be created in a standard text editor and then loaded into the compact flash card. Any file name may be used for a sequence program but it must have the dot extension “.seq” appended to the name.

The programs can then be automatically executed when the unit starts up or manually started through direct commands entered from the Console or DTE serial ports. A special file name “startup.seq” is reserved for the auto-executed sequence. The unit will look for this file at startup and then, if found, loads, compiles, and executes it. One sequence can also be used to load and start another sequence by including the ‘Ldseq’ and ‘Playseq’ commands.

Up to 16 sequences of 2048 lines each can be played simultaneously. Unless otherwise stated, all commands listed in this document can be incorporated in a sequence program. Both local and global variables can be created using ‘Define’ command. Each sequence maintains its own set of local variables and has access to a set of global variables.

A group file is a subset of the sequence file. Like sequence files, they are text files that have the “.seq” extension. The line limit for a group file is limited only by the CF card size. One advantageous feature of group files is that lines in the file are not counted as additional lines in the calling sequence program and do not contribute to the 2048 line limit.

The group files may be played directly by filename. However, the primary use for them is to provide a means for playing a common set of sound, control, and lighting files using a single numeric reference. This facilitates random and sequential show playback. The reference number is assigned using the enumeration command ‘Formgroup’. The group file can then be played using the ‘Playgroup’ command.

Comments can be used to make a program easier to read and understand. They are used for documentation and perform no programmatic action. The “;” character is used to define a comment. It may be placed on its own line or at the end of a sequence command. Once a “;” character is detected, additional characters on that line are ignored. For example:

|  |
| --- |
| ;This is the program body  main do ;Start of main loop |

Jump labels mark program positions and provide destinations for the ‘Goto’ command instruction and identify breakpoint position for program debugging. They may precede other commands or can be on their own line. They may not be names that are in the reserved words list. Program line jump labels can be used in sequences to facilitate program flow and also provide a reference for the debugging commands ‘Setbreak’, ‘Step’, and ‘List’ described in this section. Please refer to the reserved words list for names that may not be used for jump labels or local or global variables.

The list that follows describes commands that are used to load, start, stop, and monitor the execution of sequences from the Console or DTE serial port.

### Setting Sequence Context:

*Syntax-* Context *“sequence # (1-16)”* 

The ‘Context’ command defines the active sequence. It gives visibility to all local variables created by the referenced sequence so that they may be accessed from the Console or DTE serial interface ports.

The following series illustrates use of the command by establishing two separate instances of variable “x”. One is in sequence 1 and the other in sequence 2. The “Context” command is used to switch back and forth between the two:

|  |  |
| --- | --- |
| Context 1   *SEQ 1 Loaded*  Define x   x = 5   Context 2   *SEQ 2 Empty*  Define x   x = 10   Print x   *x=10*  Context 1   *SEQ 1 Loaded*  Print x   *x=5* |  *Sets the context to sequence 1*   *Displayed status from the unit*  *local variable created for seq. 1*   *Sets the context to sequence 2*   *Displayed status from the unit*   *local variable created for seq. 2*   *Displayed value from the unit*   *Sets the context to sequence 1*   *Displayed status from the unit*   *Displayed value from the unit* |

### Local Integer and String Variable Definition:

*Syntax-* Define <$>*“variable”* 

This command creates a local variable in the active sequence. The active sequence is can be defined through direct commands using the ‘Context’ command. Local variables are only available to the sequence they were defined in and therefore, can be repeated in any other sequence. The term “variable” can include any text characters not included in the reserved words list. Ex/

|  |  |
| --- | --- |
| Define pause32   Define $mystring  |  *Creates a integer variable*   *Creates a string variable* |

### Global Integer and String Variable Definition:

*Syntax-* Define Global <$>*“variable”* 

Define Public <$>*“variable”*  (rPod 8.4 only)

This command creates a global variable. Global variables are available to all sequence programs. The term “variable” can include any text characters not included in the reserved words list. The Public variation applies to the rPod 8.4 only. Public variables are also shared by all sequences but also between the two separate processors on the rPod. Ex/

|  |  |
| --- | --- |
| Define Global main\_count   Define Global $astring   main\_count = 50   $astring = “Hello”   Print main\_count   *(G)main\_count=50*  Print $astring   *Hello* |  *Creates a global integer variable*   *Creates a global string variable*   *Displayed value from the unit*   *Displayed value from the unit* |

### Group Enumeration:

*Syntax-* Formgroup *“file name”* as *“group #”* 

*Parameters-* *“group #” =* group # in the range of 1 to 500. The group number may be specified with local or global variables.

“*file name*” = Sequence file name on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional..

The ‘Formgroup’ command is used to create a numeric reference to a group file. The group file is text based source code residing on a CF card. Once the group # is assigned to a file then the file can be executed by reference with either a constant value or a variable value. Group number/file assignments may be changed at any time as required by the program. See the ‘Playgroup’ command. Ex/

If the file “show1.seq” contains the following:

|  |
| --- |
| Play Flim on 1 ;Play the sound file for show1  Playdmx lights on 3 ;Play the companion DMX lighting file  Dmx 1-1 on ;Turn on channel W1 control bit |

Then the file can be enumerated as a group by entering:

|  |
| --- |
| Formgroup show1 as 1 |

Then the three commands in the file show1.seq can be executed with the commands:

|  |  |
| --- | --- |
| Playgroup 1   Playgroup x  | or  where x = 1 |

### Group Playback:

*Syntax-* Playgroup *“file name”* 

Playgroup *“group #”* 

Playgroup *“variable”* 

*Parameters-* “*file name*” = Sequence file name on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional.

*“group #” =* group # in the range of 1 to 500.

The ‘Playgroup’ command is used to playback a “group” file. Group files are text based sequence files, residing on a CF card, that contain direct executable code. Any command that can be entered through the Console may be included in a “group” file. All branching, looping, timer or delay commands are invalid. “Group” files may be called by referencing the file name, or by a constant or variable number. To play a group by number it must have been previously enumerated using the ‘Formgroup’ command. The following is an example of the three ways to play a group:

|  |  |
| --- | --- |
| Playgroup show1.seq   Playgroup 1   Playgroup x  | or  or  where x is a previously defined global or local variable |

### Loading a Sequence File:

*Syntax-* Ldseq *“file name”* {on *“sequence # (1-16)}* 

*Parameters-* “*file name*” = Sequence file name on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional. Console or DTE is also valid.

*“sequence #” = Optional* sequence program # in the range of 1 to 16. If the sequence # is not specified then the first free sequence is used. Sequence numbers may be specified with local or global variables.

The ‘Ldseq’ command is used to load text based source code from the CF card and then compile and locate the executable program into memory. Up to 16 programs may be resident and running at the same time. Each program may contain up to 1024 lines. The ‘Ldseq” command must be called before running it with the ‘Playseq’ command. It only needs to be called one time unless the unit power is cycled or another sequence subsequently replaced it. To load a sequence type:

|  |  |
| --- | --- |
| Ldseq Show1.seq  Ldseq Show1 on 1  Ldseq Show1 on x | or  or  where x = 1 |

The unit can be setup to automatically run a sequence file at startup. During power up, the system looks for a script called “startup.seq”. If it is found then it is loaded, compiled, and executed. Please note that sequence #1 is used for this auto-executed program and loading over this program will stop its execution.

### Play Sequence Program:

*Syntax-* Playseq *“sequence # (1-16)”* 

Playseq *“file name”* {on *“sequence # (1-16)}*  

Startseq *“sequence # list (1-16)”* 

Startseq *“file name list”* 

Startseq all 

Go 

*Parameters-* *“sequence #” =* Sequence program # in the range of 1 to 16. Sequence numbers may be specified with local or global variables.

“*file name*” = Sequence file name on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional. Console or DTE is also valid.

The ‘Playseq’, ‘Startseq’, ‘Startseq all’, and ‘Go’ commands start execution of sequence programs. The sequence script may have been previously loaded and compiled by the unit and associated to the sequence # using the ‘Ldseq’ command. An example showing how to start the sequence #2 program is shown below:

|  |  |
| --- | --- |
| Playseq 2  Playseq x | or  where x = 2 |

To load and start a sequence that has not been previously loaded type:

|  |
| --- |
| Playseq show1 on 2  |

Sequences can be designed to either self-terminate or loop forever. In either case, calling the “’Playseq’ command again will always cause the running sequence to start over.

The ‘Startseq’, ‘Startseq all’, ‘Go’ commands are used to resume playback of paused programs that have been stopped with the ‘Halt’ command. The ‘Go’ command will only start the sequence referenced by the current context. Programs will continue execution from the instruction line that was last executed.

### Stopping Running Sequences:

*Syntax-* Halt *“sequence # list (1-16)”* 

Halt *“file name list”* 

Halt all 

*Parameters-* *“sequence # list” =* A list of sequence program #s in the range of 1 to 16. Sequence numbers may be specified with local or global variables.

“*file name list*” = A list of Sequence file names on the specified CF card. The prefix drive name, “a:\” or “b:\” is optional. The dot extension is also optional. Console or DTE is also valid.

The ‘Halt’ command stops the execution of one or more sequence programs. When the sequence stops, the sequence# and next instruction to be executed is listed to the Console or DTE port when the verbose mode is enabled (default). To resume execution of the sequence(s), issue either the ‘Go’ or ‘Startseq’ command. To restart the sequence from the beginning, use the ‘Playseq’ command. Please note that these commands will not stop any audio or control files playing.

To stop the sequence, enter:

|  |  |
| --- | --- |
| Halt 1, 2   Halt all  | or |

### Stepping through a sequence:

*Syntax-* Step *“# of steps”* 

*Parameters-* *“# of steps” = optional* number of program lines to advance including blank or comment lines.

The ‘Step’ command is a useful tool for debugging sequence programs. It allows the user to trace the execution a sequence a specified number of program steps. This command can only be used if the sequence is active and in a stopped condition. See the ‘Halt’ and ‘Break’ commands. The sequence to be stepped is set by the ‘Context’ command. Blank or commented lines are considered valid program steps and are included. If the “# of steps” is not specified then one step is used. When the steps are completed, the sequence stops and the sequence# and next instruction to be executed is listed to the Console or DTE port when the verbose mode is enabled (default). If a breakpoint is encountered, the program will stop before the specified number of steps is executed. To Step a sequence 1000 steps type:

|  |
| --- |
| Step 1000  |

### Setting program breakpoints:

*Syntax-* Break *“line #/label/variable list”* 

Break *“line #/label/variable list”* off

Break all off 

*Parameters-*  *“line#/label/variable” =* program line number, jump label, or contents of a variable

The ‘Break’ command is another useful tool for debugging sequence programs. It allows the user to specify a stop position in a sequence program. When a sequence is started using the either the ‘Playseq’, ‘Startseq’, or ‘Go’ command, execution will proceed until a breakpoint is encountered. After the program stops at the breakpoint, the line will be displayed on the Console or DTE. The sequence is defined by the ‘Context’ command. Blank or commented lines are considered valid program steps and may be selected for breakpoints. The breakpoint position can be specified by either the line number, a jump label, or by the contents of an existing global or local variable. Ex:

|  |  |
| --- | --- |
| Break Loopstart, programend   Break 15, 20, 37  | *or* |

To remove an existing breakpoint, add the “off” modifier to the command. To remove all breakpoints, issue the ‘Break all off’ command. Ex/

|  |  |
| --- | --- |
| Break Loopstart off   Break all off  | *or* |

### Listing a Sequence:

*Syntax-* List *“Starting line #/label” {* to *“end line #/label” /* :*”line count”}* 

*Parameters-* “*line #/label” = Optional* listing start and end print range. This can be specified as a line number or a jump label.

:“*line count” =* specifies the number of lines to print. This optional parameter overrides the end line#/label when specified.

The ‘List’ command is used to display a sequence to the Console or DTE ports. The sequence is defined by the ‘Context’ command. A “>” character marks the current program position only when the sequence is stopped. In addition, “\*” characters mark any lines that have breakpoints assigned to them. The entire sequence can be listed by entering:

|  |
| --- |
| List  |

To list a range of program steps from the jump label “Mainloop” to line 27 type:

|  |
| --- |
| List Mainloop to 27  |

To list 10 lines from a specific jump label “Mainloop” type:

|  |
| --- |
| List Mainloop :10  |

To list a specific range of the sequence using only jump labels, type:

|  |
| --- |
| List branch1 to branch5  |

## Program Flow

The commands in this section are only available for use in sequence programs and are not intended for direct entry on the Console or DTE ports. The main purpose for the following commands is to facilitate sequence program flow control. They consist of branching, looping, and conditional decision making constructs.

### Timer Delay:

*Syntax-* Wait *“time”* 

*Parameter* – “*time*” = All time parameters are expressed in milliseconds. Times may be specified with local or global variables. Times are in frames or milliseconds depending on the TIMEREF setting.

This sequence program command is used to delay program execution of the running sequence. Two versions of the instruction are available. The first one uses a specific time value. Ex.

|  |  |
| --- | --- |
| Wait 1000   Wait t1  | or  Where t1 = 1000 |

### Branch immediate:

*Syntax-* Goto *“line label”* 

*Parameter* – “*line label*” = Jump label .

The ‘Goto’ instruction is used to re-direct program flow. The destination of the jump is always a line jump label. Programs may not use any of the reserved words for line labels as they may be interpreted as valid commands. Branch destinations can be forward or backward. The following shows an example of how to use the ‘Goto’ instruction.

|  |
| --- |
| Define status   Main Play Flim on 1 T1   do   status = ?P1   while status <> 0   Goto Main  |

The program above shows an alternate method for looping a sound file. The more efficient ‘Loopsnd Flim on 1 T1’ will perform the same function.

### Branch conditional construction:

*Syntax-* if *“operation”* 

else 

endif 

*Parameter* – “*operation*” = see the operation forms section for a list of valid operations.

This is the method for making conditional decisions within a sequence program. Every branch conditional construct begins with the ‘if’ statement following by one or more program instructions. Instructions after the ‘if’ statement are only executed if the “operation” is true. The construct is completed with the “endif” statement. An optional ‘else’ statement may be included to provide false case processing. Any number of instructions may follow the ‘else’ statement. Conditional branch constructs may be nested multiple levels. The following shows an example of this command construct.

|  |
| --- |
| Define Count   Count = 0   B1 if Close1 == 1   Play Flim on 1 T1   Count = 0   else   Count = Count + 1   endif   Goto B1  |

### Branch on Selection construction:

*Syntax-* select *“variable”* 

case n 

operations

break

case m 

operations

break

.

.

.

endsel 

*Parameter* – “*variable*” = see the variables section.

This construction always begins with a “select” and ends with an “endsel”. Any number of “case/break” combinations can be implemented. In each case, a constant number (n and m) is compared to the select variable. If the comparison is true then all operations after the case and up to the first break encountered are executed. Fall through is permitted so that operations can be shared by multiple cases.

The selection construction can be nested and there is no practical limit to the nesting. The following example demonstrates the use of Branch on selection. The sequence looks for a closure on trigger input 1 and then on each trigger, steps through the playback of three separate messages.

|  |
| --- |
| Define a   a = 0   B1  if Close1 == 1   Goto B2  endif  Goto B1   B2  select a  case 0  Play Flim on 1 T1   break  case 1  Play Howl on 1 T1   break  case 2  Play Horn on 1 T1   break  endsel   a = a + 1  if a > 2  a = 0  endif  Goto B1 |

### do loop construction:

*Syntax-* do 

while *“operation”* 

*Parameter* – “*operation*” = see the operation forms section for a list of valid operations.

Every ‘do’ construct begins with the ‘do’ statement and is followed by one or more program instructions. Instructions after the ‘do’ statement are always executed. The construct is completed with the ‘while’ statement. The operator is evaluated when the program counter reaches the ‘while’ statement. If it is true then a branch to the line after the originating ‘do’ statement is made. The program will continue at the line following the ‘while’ statement if it is evaluated as false. ‘While loop’ constructs may be nested multiple levels. The following shows an example of this command construct.

|  |
| --- |
| Define Count   Count = 0   Play Flim on 1 T1   do   Count = Count + 1   while ?P1 <> 0  |

### For loop construction:

*Syntax-* for *“variable”* = *“operand1”* to *“operand2”* 

next 

*Parameter* – “*variable*” = see the Define command section for description of variables.

“*operand*” = see the operand section for a list of valid operands.

The ‘for’ construct begins with the ‘for’ statement and is followed by one or more program instructions. The “variable” parameter is the loop “incrementer” and must be a global or local variable. Each time the program reaches the ‘for’ statement, it will be incremented and compared to the parameter “operand2”. The “operand1” parameter defines the initial state of the parameter “variable” and may be a variable or constant value. When “variable” is equal to”operand2” then the loop will be terminated and program execution will continue at the line after the “next” statement. ‘For loop’ constructs may be nested multiple levels. The following shows an example.

|  |
| --- |
| Define Count   for Count = 1 to 100   Play Flim on 1 T1   do   while ?P1 <> 0   next  |

## Event Programming:

This section provides the set of procedures to control the timer in the MS2 and rPod models. See the Timer Based Actions section provided earlier in this document.

### Event List Header and Footer:

*Syntax-* Eventlist *“name”* 

Endlist 

*“name”* *-* This parameter is a constant string indicating the event list name. There can be 64 uniquely named Events per system.

Each ‘Eventlist’ command must have a header at the beginning and footer at the end. Timer cues are placed between the header and footer

### Event Command:

*Syntax-* EVENT on 

EVENT off 

EVENT = *“name”* 

EVENT offset “*timer/constant/variable”* 

*“name”* *-* This parameter is the Eventlist name.

“*timer/constant/variable” -* This is a constant or numeric variable representing frames. A time-formatted constant is also valid.

The “’EVENT’ command must be used to specify the active event list. This feature allows for show variations based on external events. The command must be in the body of the main sequence program. See below:

|  |
| --- |
| EVENT = mainshow *;Select the “mainshow” event list*  Tsync = LOCAL *;Set the timer source to onboard*  TIMER = CLOCK *;Set timer to real time clock*  TIMER on 15:00:00 *;Turn on the timer with Event offset of 3 PM*  EVENT on *; activate the event list*  J1 if Close1 == 1 *;If trigger input 1 has closed then*  EVENT off *;Stop the event list .*  endif  if Open1 == 1 *;If trigger input 2 has opened then*  EVENT on *;activate the event list.*  endif  if Close2 == 1 *;If trigger input 2 has closed then*  EVENT = sideshow *;Change the show to “sideshow”*  endif  if Open2 == 1 *;If trigger input 2 has opened then*  EVENT = mainshow *;Change the show back to “mainshow”*  Endif  Goto J1 |

To disable all event lists, use ‘EVENT off’ (default).

The Event offset command allows for a positive or negative offset from the system timer. This value is the number of frames to offset the event list playback from.

### Event Sensitivity:

Event sensitivity can be controlled using the “window” and “setback” options for the TIMER command.

*Syntax-* TIMER window “*timer/constant/variable”* 

TIMER setback “*timer/constant/variable”* 

“*timer/constant/variable” -* This is a constant or numeric variable representing frames. A time-formatted constant is also valid.

‘TIMER window n’: Sets the timer window for recalculating and event list trigger list. Any timer where the change is greater than the window setting will cause an event list recalculation. Any change less will cause execution of all events between a given time and the newly requested time position.

‘TIMER setback n’: Sets the event time allowable jitter. Setback values less than the expected jitter may result in multiple triggering of an event or events.

Two system readable variables, window and setback are available for reading.

*Syntax- “operand” =* window

*“operand” =* setback 

*Parameters* -- *“operand” =* a destination integer. Any defined local or global variable.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x = window   |  |  |  | | --- | --- | --- | | x = setback |  |  | | x will contain the current window setting  x will contain the current setback value |

### Cue List Commands:

Cue:

*Syntax-* Cue *“timer/constant/variable” “command”* 

Cue group *“group number”* *“timer/constant/variable” “command”*

*“command”* *-* This is any valid sequence instruction or console directive defined in the command list index.

*“group number”* *-*This is a group number index. (1-64)

“*timer/constant/variable” -* This is a constant or numeric variable representing time in frames. A time-formatted constant like [00:14:23.25] is typically used.

This instruction loads a timed command into the cue list. If it is associated with a group then an offset timer reference may be used after the ‘Cue group now’ command is issued. When a group is not specified, the timer value is used as defined.

### Cue relative:

*Syntax-*

Cue group *“group number”* now

Cue group *“group number”* never

*“group number”* *-*This is a group number index. (1-64)

The ‘Cue group “group number” now’ command sets the group offset time to the current timer value. All subsequent ‘Cue group’ commands will add this offset to the specified time. This relative time is then used to trigger the command. ‘Cue group “group number”’ never clears the timer offset to zero. In this case, ‘Cue group’ commands will operate the same as the standard ‘Cue’ command. The ‘Cue group “group number” now’ command must be issued prior to Cue group commands are issued.

### Cue Clear:

*Syntax-*

Cueclear all

Cueclear group *“group number”* 

*“group number”* *-*This is a group number index. (1-64)

The ‘Cue clear’ command will either purge all pending cues in the cue buffer or just the cues with the specified group number.

### Cue print:

*Syntax-*

Cueprint<c><d><e><m><x>

Cueprint <c><d><e><m><x> next

Cueprint<c><d><e><m><x> group *“group number”* 

*“print”* *-* print : print to the originating device.

printc : print to the console port.

printd : print to the DTE port.

printe : print to the Ethernet port.

printm : print to the MIDI port in serial mode.

printx : print to the DMX port in RS-485 mode.

*“group number”* *-*This is a group number index. (1-64)

This command prints the current cue buffer list to the specified device. If “next” is specified then only the next command in the cue buffer is printed. Likewise, if a “group” is specified then only cues matching that group are printed.

## Timer Synchronization:

A method for synchronizing multiple units is provided using UDP Ethernet protocol. This protocol requires a unique socket using port 11001. It uses a single master and multiple slaves. The protocol accounts for both time offset and propagation delay that may occur from Wifi or radio network transmission latencies. Time correction varies depending on the timer mode. Smooth correction tracking will occur when the timer is on or in the hold mode. Immediate correction occurs when timer is in the off mode or if the timer change is outside the window parameter.

Timer accuracy varies depending on network configuration. It is guaranteed to be accurate to a frame (1/30 sec.). However, a typical wired network shows synchronization to +/-1ms.

### Timer Sync Commands:

*Syntax-* Sync on 

Sync off  (Default)

Sync rate 10000 (Default is 10 seconds)

Sync now 

Sync block 

Sync allow 

TIMER master

TIMER slave (Default)

“*constant/variable” -* This is constant or numeric variable representing milliseconds. A value of 10,000 represents 10 seconds.

At power-up, the unit defaults to the slave synch mode. However, if timer sync is used, there must be a single system master. The master can be established through a sequence program or by an external command. The ‘TIMER master’ command is used to set this mode. Once a master is defined, all other units on the network will respond to any Sync related command. For example, if the master issues a ‘TIMER stop’ command, all other units on the network will have their timer stopped.

The timer synchronization system can operate with all units set to slave mode. A slave can issue a ‘Sync now’ command causing all other units in the network to be synchronized to its timer. However, no other timer related commands would be forwarded to the receiving unit.

When a TIMER master has been defined, synchronization can be set to happen on an interval defined by the ‘Sync rate n’ command. A master can use the ‘Sync on’ command to enable periodic synchronization. Periodic synchronization can be disabled using ‘Sync off’ command.

By default, timer synchronization is enabled. To disable timer synchronization use the ‘Sync block’ command. To re-enable synchronization use the ‘Sync allow’ command.

## Synchronizing Audio to the Master Timer

Audio tracks can be synchronized to the main timer using the ‘Lock’ command. A specified offset from the master clock can be optionally provided in the command. This offset is correlated to the sound file start position. Playback is throttled using a pitch algorithm to maintain synchronization. Because of buffering, the tracking accuracy limited to approximately +/- 3 milliseconds. If the time is more than +/-1 second off then the track is consider severely out of synchronization and will abruptly skip to the correct time. Stopping the master timer will also stop audio playback of locked tracks. When the timer is resumed then the audio tracks will resume, remaining in synchronization. When a sound file is assigned to a locked track and the time + offset has not been reached yet, the file will be pre-loaded instead of playing immediately. For this case, playback will not begin until time + offset is reached.

### Locking Sound Tracks:

*Syntax-* Lock *“track # list”* to *“offset time (frames)”* 

Unlock*“track # list”* 

*Parameter* **--** *“track # list”* ***=*** List of tracks that are actively playing sounds. Track numbers may be specified with global or local variables*.*

*“offset time (ms)” =*This optional parameter is expressed in frames. Times may be specified with local or global variables.

These commands are used to lock or unlock the playback of the specified track(s) to the main timer + offset time. If a sound is started prior to the timer + offset then it will be put in the pre-loaded state. Then, when the proper start time is reached the sound will begin playing. If the timer is stopped, the sound will stop playing. Resuming the timer will cause the sound to resume playing where it left off. Changing the timer to within the sound play window will cause the sound to be re-positioned as well.

To lock Tracks 1 to timer + 10 seconds enter:

|  |
| --- |
| Define toffset  toffset = TIMER  toffset = Toffset + 300  Lock 1 to toffset   Play sound 1 on 1 T1  |

## Custom Audio Processing

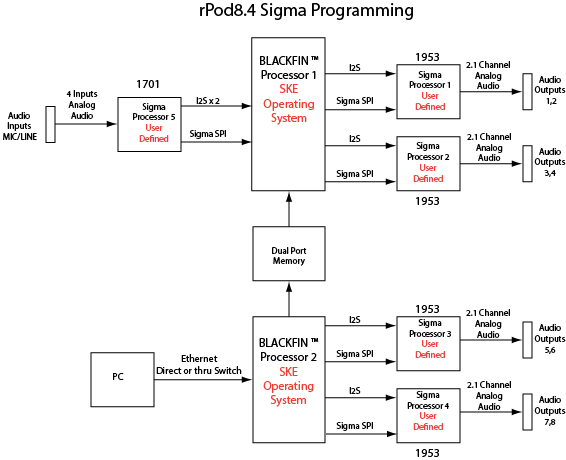
This section provides the set of procedures to configure and customize the family of Sigma Processor based systems such as the MSII and rPod models. The following assumes that a copy of Sigma Studio has been obtained from Analog Devices. This software is provided for free by Analog Devices as of the writing of this document. The user should install the software following the instructions provided by the manufacture.

### System Description:

The MSII, rPod10.2 and rPod8.4 uses multiple DSP based processors to generate audio sourced from either the analog inputs or from a compact flash card. The Blackfin main processor provides the overall system management and coordinates the transfer and conversion of raw audio data to track and channel stereo audio data. It manages track audio volume control and polyphony processing. Each Sigma processor receives two channel audio data and then performs additional signal processing algorithms on the data such as:

1. Channel volume control
2. Multi-band equalization
3. Subwoofer cross-over filtering
4. Dynamics processing (compression, expansion, midnight control, etc.)
5. Spatial processing (Fat stereo, Dynamic Bass, etc.)
6. Line delay for speaker position and phasing.

Finally, the Sigma processor converts the 2.1 audio data to analog signals using the onboard triple digital to analog converters.

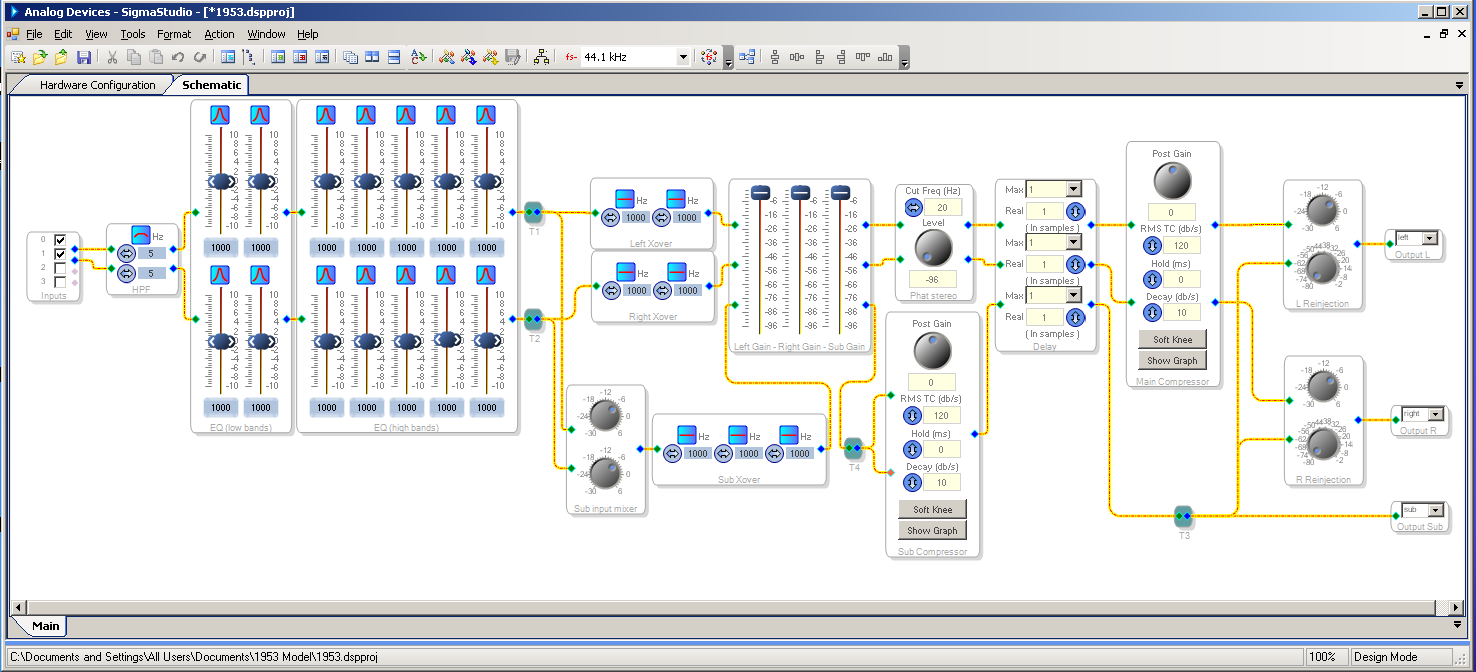


### Using the Default Model:

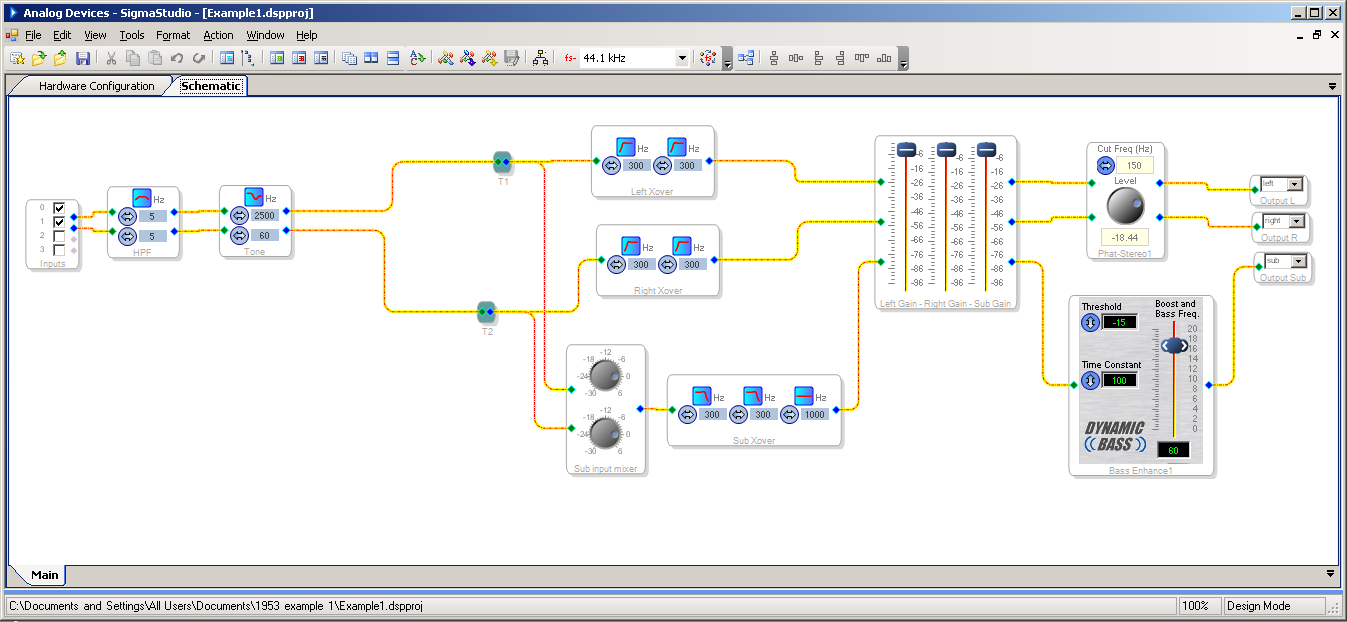
Each Sigma processor can be setup using a default program model or by a user defined model. The design models reside on each CF card and downloaded each time the unit is powered up. For the 1953 audio codecs, a default model and setup parameters are contained in the on-board ROM (read only memory) and are transferred to program memory when the chip is powered up. The user may choose to use the default model in which case no custom program or setup files need to be provided.

### Creating a Custom Design:

If a user defined program and parameter setup is to be used then they are loaded on one of the CF cards and automatically transferred by the SKE operating system as part of the booting process. An equivalent Sigma Studio model of the default model is provided by SKE and can be customized by the user to provide the desired system performance. The model is contained in the folder 1953 Model. It consists of a seven band EQ stage, cross-over filtering, master volume control, spatial processing, dynamics processing and sub-channel re-injection stage as shown below. The user can, using the Sigma Studio application, customize the characteristics of each element and then compile and generate the custom program and setup files.



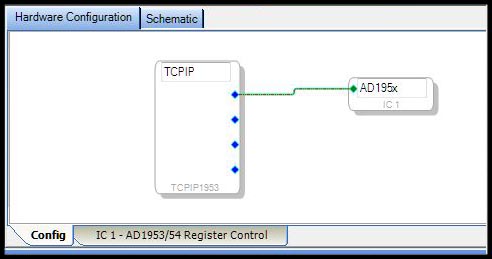
An alternate model is also provided showing the versatility of this feature. This model is in the 1953 example 1 folder. The user is free to develop their own schematic using the Sigma Studio tools. It should be noted that the design elements must be chosen consistent with the AD1953 processor. The schematic is shown below.

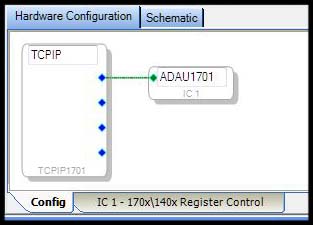
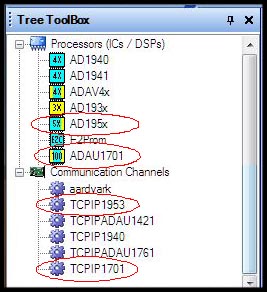


### Input DSP Processing

The rPod-8.4 Rev 1.2 and later versions feature an input DSP that allows you to use Sigma Studio to process the signals from the MICL, MICR, and LINEL, and LINER. The process is the same as for the output CODECs shown above, with the exception that DSP Processor Selection should be the ADAU1701 instead of the AD1953 in the setup, and the files that are generated must be named “ProgD1.hex” and “SetupD1.hex” and placed on CF Card 1.

### Sigma Studio Setup:

The following hardware configuration tools are available in Sigma Studio. In order to connect a Mini-Sam or rPod to the Sigma Studio software, a communications channel must be established. The two channels available are the TCPIP1953 and the TCPIP1701. The TCPIP1953 is used to target an output audio codec processor and the TCPIP1701 input audio processor. Each of these channels supports four nodes (hardware chips). Only one of the output processor nodes is used on the Mini-Sam while the rPod uses four nodes. The TCPIP1701 input processor is used only by the rPod. It requires only one node. Each communications channel must be connected to a processor IC. There are two processor ICs used, the AD195x and the ADAU1701. Below shows how to connect these ICs:



### Ethernet Control of DSP Processing

The rPod and Mini-Sam uses a TCP/IP interface to setup and test the DSP design in real time. Once the design has been tested and verified, the resident program and filter data can be extracted from the specified processor to create the desired configuration files. The files are automatically created and stored on the resident compact flash card(s). To simplify development on the rPod-8.4, a single two channel design can be re-directed to any of the four codec sections using the mapping feature.

**Connecting to a unit**

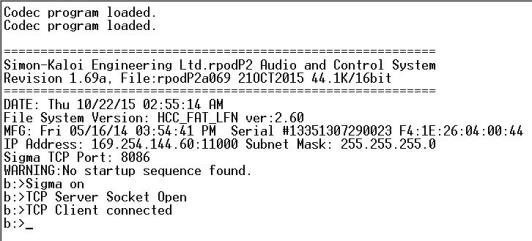
The DSP design service uses a default Ethernet port of 8086. This must match the port defined in Sigma Studio. The TCP/IP connection is enabled and disabled using these commands:

*Syntax-*

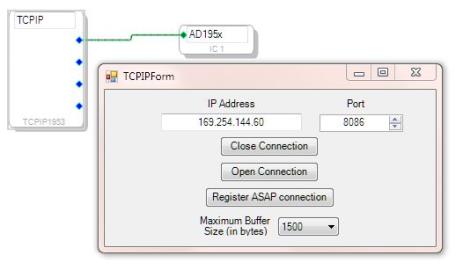
Sigma on ;Enable connection to Sigma Studio

Sigma off ;Disable connection to Sigma Studio

Sigma port *<port #>*  ;Set the TCP server port number.(default is 8086)



After the TCP Server Socket is opened and upon opening a Sigma Studio project file, you should get a “TCP Client connected”. If this does not happen then in Sigma Studio on the hardware configuration tab, right click the communications channel and next to the “TCPIP” label and it will open the dialog below. Type in the unit IP address and verify the Port number. A connection can be opened by selecting the “Open Connection” button.



**Mapping Processors to a Sigma Studio Design**

This command applies only to the AD1953 output codec/processors for the rPod. It maps the Ethernet Streaming Connection nodes to rPod DSP SPI connections. All four processors must have an assignment. Assignments can be made to non-existing Sigma Studio IC’s. 1701 connections are differentiated by protocol.

*Syntax-*

Sigma map *<processor map list>*

*“processor map list”* *-* This is a list of the 4 rPod DSP processors

The command below maps each processor to its own Sigma design IC.

Sigma map 1,2,3,4* ;Default*

For a single IC design the following maps IC1 in Sigma Studio to all 4 codecs (audio output channels 1-8):

Sigma map 1,1,1,1**

It’s possible to provide two Sigma IC designs and map any one of the processors to an IC. The following command maps IC2 to audio output channels 1-4 and IC1 to channels 5-8.

Sigma map 2,2,1,1**

**Saving the Design**

Extracts the resident program and filter data from the specified processor and creates the new configuration files. The designs must be loaded through Sigma Studio before executing these commands.

*Syntax-*

Sigma save *<processor map list> ;rPod version*

Sigma saveall * ;rPod version*

Sigma save* ;Mini-Sam version*

*“processor map list”* *-* This is a list of any one or more of the 5 rPod DSP processors

Here are some command examples:

Sigma save all/1,2,3,4,5

Sigma save 1 *;Saves codec 1 codec configuration files (audio output channels 1 and 2)*

Sigma save 2 ;*Saves codec 2 codec configuration files (audio output channels 3 and 4)*

Sigma save 3 ;*Saves codec 3 codec configuration files (audio output channels 5 and 6)*

Sigma save 4 *;Saves codec 4 codec configuration files (audio output channels7 and 8)*

Sigma save 5 ;*Saves the input processor configuration file.*

## ASCII Table

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex |
| (nul) | 0 | 0x00 | (sp) | 32 | 0x20 | @ | 64 | 0x40 | ` | 96 | 0x60 |
| (soh) | 1 | 0x01 | ! | 33 | 0x21 | A | 65 | 0x41 | a | 97 | 0x61 |
| (stx) | 2 | 0x02 | " | 34 | 0x22 | B | 66 | 0x42 | b | 98 | 0x62 |
| (etx) | 3 | 0x03 | # | 35 | 0x23 | C | 67 | 0x43 | c | 99 | 0x63 |
| (eot) | 4 | 0x04 | $ | 36 | 0x24 | D | 68 | 0x44 | d | 100 | 0x64 |
| (enq) | 5 | 0x05 | % | 37 | 0x25 | E | 69 | 0x45 | e | 101 | 0x65 |
| (ack) | 6 | 0x06 | & | 38 | 0x26 | F | 70 | 0x46 | f | 102 | 0x66 |
| (bel) | 7 | 0x07 | ' | 39 | 0x27 | G | 71 | 0x47 | g | 103 | 0x67 |
| (bs) | 8 | 0x08 | ( | 40 | 0x28 | H | 72 | 0x48 | h | 104 | 0x68 |
| (ht) | 9 | 0x09 | ) | 41 | 0x29 | I | 73 | 0x49 | i | 105 | 0x69 |
| (nl) | 10 | 0x0A | \* | 42 | 0x2A | J | 74 | 0x4A | j | 106 | 0x6A |
| (vt) | 11 | 0x0B | + | 43 | 0x2B | K | 75 | 0x4B | k | 107 | 0x6B |
| (np) | 12 | 0x0C | , | 44 | 0x2C | L | 76 | 0x4C | l | 108 | 0x6C |
| (cr) | 13 | 0x0D | - | 45 | 0x2D | M | 77 | 0x4D | m | 109 | 0x6D |
| (so) | 14 | 0x0E | . | 46 | 0x2E | N | 78 | 0x4E | n | 110 | 0x6E |
| (si) | 15 | 0x0F | / | 47 | 0x2F | O | 79 | 0x4F | o | 111 | 0x6F |
| (dle) | 16 | 0x10 | 0 | 48 | 0x30 | P | 80 | 0x50 | p | 112 | 0x70 |
| (dc1) | 17 | 0x11 | 1 | 49 | 0x31 | Q | 81 | 0x51 | q | 113 | 0x71 |
| (dc2) | 18 | 0x12 | 2 | 50 | 0x32 | R | 82 | 0x52 | r | 114 | 0x72 |
| (dc3) | 19 | 0x13 | 3 | 51 | 0x33 | S | 83 | 0x53 | s | 115 | 0x73 |
| (dc4) | 20 | 0x14 | 4 | 52 | 0x34 | T | 84 | 0x54 | t | 116 | 0x74 |
| (nak) | 21 | 0x15 | 5 | 53 | 0x35 | U | 85 | 0x55 | u | 117 | 0x75 |
| (syn) | 22 | 0x16 | 6 | 54 | 0x36 | V | 86 | 0x56 | v | 118 | 0x76 |
| (etb) | 23 | 0x17 | 7 | 55 | 0x37 | W | 87 | 0x57 | w | 119 | 0x77 |
| (can) | 24 | 0x18 | 8 | 56 | 0x38 | X | 88 | 0x58 | x | 120 | 0x78 |
| (em) | 25 | 0x19 | 9 | 57 | 0x39 | Y | 89 | 0x59 | y | 121 | 0x79 |
| (sub) | 25 | 0x1A | : | 58 | 0x3A | Z | 90 | 0x5A | z | 122 | 0x7A |
| (esc) | 27 | 0x1B | ; | 59 | 0x3B | [ | 91 | 0x5B | { | 123 | 0x7B |
| (fs) | 28 | 0x1C | < | 60 | 0x3C | \ | 92 | 0x5C | | | 124 | 0x7C |
| (gs) | 29 | 0x1D | = | 61 | 0x3D | ] | 93 | 0x5D | } | 125 | 0x7D |
| (rs) | 30 | 0x1E | > | 62 | 0x3E | ^ | 94 | 0x5E | ~ | 126 | 0x7E |
| (us) | 31 | 0x1F | ? | 63 | 0x3F | \_ | 95 | 0x5F | (del) | 127 | 0x7F |

|  |  |  |
| --- | --- | --- |
| ASCII Name | Description | C Escape Sequence |
| nul | null byte | \0 |
| bel | bell character | \a |
| bs | backspace | \b |
| ht | horizontal tab | \t |
| np | formfeed | \f |
| nl | newline | \n |
| cr | carriage return | \r |
| vt | vertical tab |  |
| esc | escape |  |
| sp | space |  |

## FAQ

### How do I read an input trigger and play a sound when the trigger occurs?

*Answer: The following script continuously reads trigger 1 and plays the sound “Flim” on channel 1.*

|  |
| --- |
| J1 if Close1 == 1   Play Flim on 1 T1   endif   Goto J1  |

### How do I cross fade two sound files?

*Answer: The command entry that follows will cross fade the playing sound “Flim” on channel 1, track 1 with “Music 1” on channel 1, track 2.*

*First, set the attack and decay times for the two tracks:*

|  |
| --- |
| Atk 1,2 = 30   Dek 1,2 = 30  |

*Next, Start playback of “Flim”enter:*

|  |
| --- |
| Play Flim on 1 T1  |

*Now, execute a cross fade type:*

|  |
| --- |
| Play Music 1 on 1 T2 X1  |

*Finally, to cross fade back to “Flim” type:*

|  |
| --- |
| Play Flim on 1 T1 X2  |

### How do I start a sound file 10 seconds into the file?

*Answer: First, pre-load the file to the desired track. Next, issue the “Pos” or “Skip” command and position the playback pointer at 10 seconds. Finally, issue the Start command to begin Playback:*

|  |
| --- |
| Ldsnd Flim on 1 T1   Pos 1 to T10   Start  |

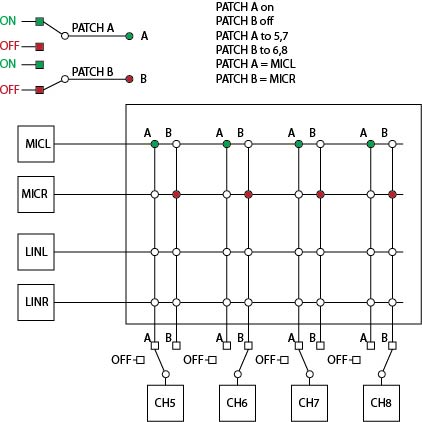
### How do I patch microphone inputs to analog outputs on the rPod10.2 and rPod8.4 .

**Also, I would like the left and right channels to be consistent from input to output?**

*Answer: The three variants of the Patch command are used to define the mapping and connect the patch:*

|  |
| --- |
| Ivol MICL = 100 ;Set the Microphone input level  Patch A = MICL ;Assign left microphone to patch A  Patch B = MICR ;Assign Right microphone to patch B  Patch A to 1,3 ;Connect Left mic. to ch 1 and 3  Patch B to 2,4 ;Connect Right mic. To ch 2 and 4  Patch A on ;”Turn on” the A connection  Patch B on ;”Turn on” the B connection  .  .  Patch A off ;”Turn off” the A connection  Patch B off ;”Turn off” the B connection |

*Below is an example of the software block diagram of a Patching of MICL to Channels 5,7 and MICR to Channels 6,8 with the associated commands.*

How do I play shows sequentially using a single trigger?

*Answer: the playback of sequential shows is accomplished in the program Main.seq using a counter defined as “showplayed”. The following example has five shows with a keep-alive feature that provides looped ambient audio in “Alive.wav” and DMX control from “Alivew.wav . The show is started by closing trigger 1. Additional triggers will be prevented until the show is complete. Upon completion, the looping keep-alive show will be restarted and the trigger enabled.*

*The Main program file startup.seq consists of:*

|  |
| --- |
| Define showplayed  showplayed = 0  Mount Alivew ;preload the DMX file for keep alive  Mount Babyw ;preload the DMX file for show1  Mount Junglew ;preload the DMX file for show2  Mount holidayw ;preload the DMX file for show3  Mount shoutw ;preload the DMX file for show4  Mount Wooliew ;preload the DMX file for show5  J1  if ?P1 == 0 ;play keep-alive show?  Playm Alive on 1 T1  Playdmx Alivew on 1  endif  if Close1 == 1  if showplayed == 0 ;play show1?  Playm CongoHeyBaby on 1 T1  Playdmx Babyw on 1  endif  if showplayed == 1 ;play show2?  Playm CongoJungleLove on 1 T1  Playdmx Junglew on 1  endif  if showplayed == 2 ;play show3?  Playm Holiday on 1 T1  Playdmx holidayw on 1  endif  if showplayed == 3 ;play show4?  Playm Shout on 1 T1  Playdmx Shoutw on 1  endif  if showplayed == 4 ;play show5?  Playm Woolie on 1 T1  Playdmx Wooliew on 1  endif  showplayed = showplayed + 1 ;point to next show  if showplayed > 4 ;go back to start?  showplayed = 0 ;yes  endif  do ;wait until show is complete  while ?P1 <> 0  endif  Goto J1 |

### How do I play random shows using a single trigger?

*Answer: the playback of random shows is accomplished in the program startup.seq using the random operation. The program shown below is streamlined through the use of four additional group files. The following example implements a three show system using only nine compiled sequence lines in main.seq. All other initialization and show commands reside on the CF Card and are retrieved as needed by the main sequence program.*

*The Main program file startup.seq consists of:*

|  |
| --- |
| Playgroup init ;initialize from init.seq file  J1  if Close1 == 1 ;Is trigger 1 closed?  x = Rand 1 3 ;get a random number from 1 to 3  Playgroup x ;Play random show  do ;Wait for track 1 to complete  while ?P1 <> 0  endif  Goto J1 |

*Group file init.seq contains initialization code for the main sequence:*

|  |
| --- |
| Define x ;random value index  Mount show1dmx ;preload the DMX file for show1  Mount show2dmx ;preload the DMX file for show2  Mount show3dmx ;preload the DMX file for show3  Formgroup show1 as 1 ;enumerate Show1.seq as 1  Formgroup show2 as 2 ;enumerate Show2.seq as 2  Formgroup show3 as 3 ;enumerate Show3.seq as 3 |

*Group file show1.seq consists of:*

|  |
| --- |
| Ldsnd show1a on 1 T1 ;Load sound a for show 1  Ldsnd show1b on 3 T2 ;Load sound b for show 1  Lddmx show1dmx on 1 ;Load DMX file for show 1  Start ;Start show 1 |

*Group file show2.seq consists of:*

|  |
| --- |
| Ldsnd show2a on 1 T1 ;Load sound a for show 2  Ldsnd show2b on 3 T2 ;Load sound b for show 2  Lddmx show2dmx on 1 ;Load DMX file for show 2  Start ;Start show 2 |

*Group file show3.seq consists of:*

|  |
| --- |
| Ldsnd show3a on 1 T1 ;Load sound a for show 3  Ldsnd show3b on 3 T2 ;Load sound a for show 3  Lddmx show3dmx on 1 ;Load DMX file for show 3  Start ;Start show 3 |

*Using this method, each show can be separately tested from the Console by typing:*

|  |
| --- |
| Playgroup show1  *or*  Playgroup show2  *or*  Playgroup show3  |

### Strings:  Can any 8bit value be stored in a string variable?

(i.e., ascii control characters such as STX(x02) or CR(x0d)

- *Storing of embedded control characters is supported by direct assignment. The following example creates a single string $a. A carriage return and line feed character is inserted between the strings “Hello” and “There”. The example assumes that CR, LF, $a, and $b have been defined.*

|  |
| --- |
| CR = 13   LF = 10   $a = “Hello” |
| $b = “There”  $a = $a + CR  $a = $a + LF  $a = $a + $b |
| Print $a  |
| Hello  There |

- *Embedded control charaters can also be printed to the port. Ex/*

|  |
| --- |
| Printd # 0x02, “1104”,0x0d |

### Comments?  What are the comment delimiters?

- The comment delimiter is the semicolon ‘;’. It may be placed on its own line or at the end of a command or operation.

### Does VAL convert the string “HB” into  0x8A? (0x48 + 0x42)

*- This conversion can be made as follows:*

|  |
| --- |
| $a = “HB”  |
| x = $a  |
| Print x  |
| 138 |

*The value 138 = 0x8A = (0x48 + 0x42)*

Will VAL convert the string “SC11104” into 0x18D? (All the acsii values added up)

Also, how can I limit it to 1 CHAR (ie  0x8D).

*- This calculation can be made as follows:*

|  |
| --- |
| $a = “SC11104”  |
| x = $a   Print x  397  x = x & 0xff |
| Print x  |
| 141 |

*The value 397 = 0x18D and 141 = 0x8D.*

### Is $a = $a + $b legal?

*- Yes this is Legal.*