rPod10.2, rPod8.4, MS-2, and MS-2E

Software Description Manual

Simon-Kaloi Engineering, Ltd.

Revision r43

Table of Contents

Introduction 9
Command Entry 9
Sequence Program Development 10
  Execution Modes 10
  Sequence Structure 10
    Programs 10
    Group Files 10
    Event Lists 11
    Cue Lists 11
    Program Layout 11
    Key Structural Guidelines 11
  Math and Logical Operations 12
  Decision Operations 13
  Looping Operations 14
  Constants 14
  Common Variables 14
  Trigger Variables 15
  Passing Processor Data (rPod8.4 Only) 15
String Operations 16
    String Concatenation 16
    String Comparison 16
    String to Number 17
    Number to String 17
    String Length 18
    Search String 18
    Left String 18
    Right String 19
    Mid String 19
Sound Channel Information 20
  Sound Channel Output Information 20
  Sound Channel Input Gain Information 20
Sound Track Operations 21
  Single Track Information 21
  Single Track Progress 23
  Track Lists 24

System Variables 25
Audio and Control Status 25
Accessing Clip Parameters 26
System Information 28
Ethernet 28
Serial Communications 29
Triggers 29
Expansion Triggers 31
DMX Input 33
Compact Flash and File Status 34
Random Numbers 34
Real Time Clock 35
System Timer 36
General Purpose Timers 37
Timer-Based Actions 38
Event Lists 38
Event List Behavior 39
Embedded Timer Points 39
Cue Lists 41
rPod8.4 Considerations 42
Table of Operations 43
Table of Operators 43
Table of System Constants and Subroutines 44
Command Description Index 46
File Access Commands 46
Change Drive 46
Display Directory 46
Change Directory 46
Make Directory 47
Remove Directory 47
Delete File 47
Rename File 47
Open File 47
Close File 48
File Configuration (Delimiter) 48
Write File 49
Read File 49
Seek File 50

Sound File Playback 51
Pre-load a Sound 51
Unload a Sound Track 52
Play Sound 52
Loop Sound 53
Un-loop Track 53
Convert to Looping 53
Append Sound 54
Stop Sound Tracks 54
Pause / Resume Sound Tracks 54
Mirror Sound Tracks (rPod Only) 55
Position Sound Track 55
Pitch Sound Track 55
Set Global Playback Sample Rate 56
Sound Volume Control 56
Channel Volume Control 56
Track Volume Control 57
Ducking Volume Setting 57
Duck Track Control 57
Un-Duck Track Control 57
Input Volume Setting 58
Patch Assignment (rPod10.2 and rPod8.4 Only) 58
Input Mapping (rPod10.2 and rPod8.4 Only) 58
Input Control (rPod10.2 and rPod8.4 Only) 59
Input Control (MS-II and rPod8.4) 59
Boost Microphone Gain 60
Mute / Un-Mute 60
Track Attack Time 60
Track Decay Time 61
Amplifier Control (MS-II Only) 61
Clip-Based Metaphors 61
Load Clip 61
Execute Clip 62
Unload Clip 62
Clip Manager Program 63
Connecting to a Clip Database 63
Navigating the Clip Database 65
Clip Database Fields 66

Track Database Fields 66
Exporting Clips 67
DMX/RS485 and Output Control Commands 68
Bit Level Control — Output 68
Bit Level Control — DMX Bit 68
Byte Level Control 69
Dimming Control 69
DMX/RS485 Output Driver Control 69
DMX File Playback 69
Pre-load DMX File 69
Unload DMX Track 70
Play DMX File 70
Loop DMX File 70
Mount DMX File 71
Un-mount DMX Files 71
Stop DMX Tracks 71
Position DMX Playback 72
DMX File Capture 72
Record DMX 72
Pre-load Record DMX 72
Start Pre-loaded Recording 73
Stop Recording DMX 73
Ethernet 73
Configure Ethernet 73
Configuring the Gateway 74
Ethernet Command Status Reporting 74
Configure FTP Password 74
Configure UDP 74
IP Referencing (Bind) 74
UDP Control (Send Command) 75
UDP Sending Data 75
Receive UDP Data 76
System Commands 76
Re-Boot 76
Set Time Base 77
Cosmic Variable Commands 77
Start Playback 78
Query Playing Tracks 78

Query Track Status 78
Print 79
Monitor Serial Port 80
Configure Serial Port 80
Null Character Substitution 81
Read Serial Port 81
Peek Serial Port 82
Silent Mode 82
Command Feedback (Verbose) 82
Install Operating System 82
Controlling Status LEDs 83
Configure Triggers 83
Logging 83
Log Activation / Deactivation 83
Log Update 84
Log File Clearing 84
Log Listing 84
Log Mode 84
Log File Handling 84
Log Timestamp 85
Sequence Programming Commands 85
Setting Sequence Context 85
Local Variable Definition 86
Global / Public Variable Definition 86
Group Enumeration 86
Group Playback 87
Load Sequence File 87
Play Sequence Program 87
Stop Running Sequences 88
Step Through a Sequence 88
Set Program Breakpoints 89
List a Sequence 89
Program Flow 89
Timer Delay 89
Branch Immediate (Goto) 90
Branch Conditional (if/else/endif) 90
Branch on Selection (select/case/endsel) 90
Do Loop (do/while) 91

For Loop (for/next) 92
Event Programming 92
Event List Header and Footer 92
Event Command 93
Event Sensitivity 93
Cue List Commands 94
Cue Relative 94
Cue Clear 94
Cue Print 95
Timer Synchronization 95
Timer Sync Commands 95
Synchronizing Audio to the Master Timer 96
Lock / Unlock Sound Tracks 96
Custom Audio Processing 96
System Description 96
Using the Default Model 97
Creating a Custom Design 97
Input DSP Processing (rPod8.4 r1.2 and Later) 98
Sigma Studio Setup and Ethernet Connection 99
Mapping Processors to a Sigma Studio Design 100
Establishing the Sigma Studio Connection 100
Saving the Design 101
ASCII Table 101
Quick Reference Guide 103
File Access 103
Sound File Playback 103
Sound Volume Control 104
Clips 105
DMX/RS485 and Output Control 105
DMX File Playback 105
DMX File Capture 106
Ethernet 106
System Commands 106
Logging 107
Sequence Programming Direct Commands 108
Program Flow 109
Event Programming 109
Timer Synchronization 109

Frequently Asked Questions (FAQ) 111
How do I read an input trigger and play a sound when the trigger occurs? 111
How do I cross-fade two sound files? 111
How do I start a sound file 10 seconds into the file? 111
How do I patch microphone inputs to analog outputs on the rPod10.2 and rPod8.4? 111
How do I play shows sequentially using a single trigger? 112
How do I play random shows using a single trigger? 113
Can any 8-bit value be stored in a string variable? 114
What are the comment delimiters? 114
Does Val convert the string "HB" into 0x8A (0x48 + 0x42)? 115
Will Val convert the string "SC11104" into 0x18D? 115
Is $a = $a + $b legal? 115

Introduction

This manual provides a software reference for the rPod10.2, rPod8.4, and MS-2 family (including the MS-2E) sound and motion control units. It covers both direct and automated control of the units using the Sequence Programming Language, and is organized into four sections:

- Operation and Use
- Quick Reference Guide
- Command Description Index
- Frequently Asked Questions (FAQ)

> **NOTE:** *This document will be updated as new features are added and software bugs are fixed. Check the SKE website or contact SKE directly for the latest revision. For hardware setup and operation, refer to the rPod10.2, rPod8.4, MS-2, and MS-2E Hardware Manuals.*

Throughout this document, references to the MS-2 also apply to the MS-2E (which is the MS-2 with Ethernet capability). Command entries are completed by pressing the Enter key, generating a carriage return (ASCII 13 / 0x0D) and line feed (ASCII 10 / 0x0A). All commands are case-sensitive and must be followed by a space character.

Command Entry

Commands can be entered through either the Console or DTE serial ports using a standard terminal program. The default port settings are:

- Baud rate: 115,200
- Data bits: 8
- Stop bits: 1
- Parity: None

The on-board console port connector is a DCE-type 9-pin female configuration. All commands are case-sensitive and must be followed by a space character.

Sequence Program Development

Sequences are small programs written in the SKE proprietary programming language. They are invoked by the unit operating system to execute specific tasks. Key characteristics include:

- Concurrency: Up to 16 sequences can execute simultaneously.
- Independence: Each sequence operates independently; execution of one does not impact others.
- Shared Information: Sequences can exchange data through a predefined set of global variables.
- Compilation: The built-in compiler accepts source text files created in any standard text editor on a PC.
- Installation: Compiled executable files are transferred to a Compact Flash (CF) card and loaded into the unit.

Execution Modes

The system operates in three distinct execution modes:

Manual Mode: Initiated through commands entered from a control port (Console or DTE). Typically used during initial development or when the unit is controlled by an external device such as a PLC or computer. The Ldseq and Playseq commands are used to load and execute programs.

Boot Mode: Automatically executed upon power-up. Used when the unit functions as a standalone controller. The system recognizes startup.seq as the designated startup program. During development and testing, remove the startup file to prevent unexpected operation.

Program Mode: Enables loading and playing of sequences from within another executing sequence using the Ldseq and Playseq commands.

Sequence Structure

Programs

Up to 16 sequence programs can operate concurrently, stored as text files on the CF card. Once loaded, programs can be executed, stopped, or restarted without reloading or recompiling. The main sequence is typically a continuously looping program using for, do, or Goto constructs.

Group Files

Group sequences initiate, execute, and terminate without interruption. They are loaded, compiled, and executed simultaneously when called. Debug features are not available for group sequences.

> **NOTE:** *The additional file accesses required for group sequences may impact timing.*

Event Lists

The system supports embedded timer event lists for fixed show control. Only one event list can be active at any given time. The rPod10.2 and rPod8.4 also offer MIDI triggering; the rPod8.4 supports external SMPTE clock triggering with the add-on board.

Cue Lists

Cue lists provide an alternative method for creating timer events, primarily used for dynamic show control. They can be generated spontaneously or embedded within sequence files.

Program Layout

The recommended structure for a typical main program:

Section	Description / Example
Variable definitions	`Define x`
Variable initialization	`x = 0`
Port initialization	`Config Console = 9600`
Control file mounting	`Mount light1 on 1`
Subprogram loading	`Ldseq Show1 on 2`
Main loop start	`main:  Playseq 1`
Program body	`x = x + 1`
Main loop end	`Goto main`
Event lists	`Eventlist  EventList1`

Key Structural Guidelines

- Port initialization and subprogram loading should not be called repetitively within the main loop.
- The main loop begins with a jump label. This label serves as the entry point and can be followed by any valid command.
- The main loop concludes with a jump back to the start label. Line numbers are not required for Goto instructions.
- The do...while loop construct provides a structured alternative to explicit jump labels.

Math and Logical Operations

Math and logical operations are the primary method for processing, calculating, and evaluating system and program resources. Statements can include a wide variety of operands and operators.

> **NOTE:** *The system restricts operations to one operator per line. For example, x = x + y * z is INVALID.*

The four major operation types are Assignment, If, While, and For. The assignment group includes direct, logical, and arithmetic forms. The table below shows examples of each:

Type	Example	Description
Direct	x = 25	Store 25 into x
Direct	y = z	Store the contents of z into y
Direct	x = Trig1	Store the state of external trigger 1 into x
Direct	$a = x	Store the character value of x in $a
Direct	x = $a	Store the ASCII value or sum-check of $a into x
Logical	y = ! x	Store the logical NOT of x into y
Logical	x = x & 5	Store the binary AND of x and 5 into x
Increment	x ++	Increment x (x = x + 1)
Increment	y = x --	Set y = x, then decrement x
Increment	y = ++ x	Increment x, then set y = x
Arithmetic	x = y - 5	Store the result of y minus 5 into x

Decision Operations

The if statement is the primary method for conditional decisions in sequence programs. It evaluates to true (non-zero) or false (zero). Two range-checking operators are also available: includes and excludes, each taking three operands (low range, high range, check value).

If statement example 1 — trigger-based conditional:

```
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 statement example 2 — variable comparison with else clause:

```
if x == 5               ; Is 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
```

Range checking example using includes and excludes operators:

```
Define xmin = 3         ; Low range value
Define xmax = 10        ; High range value
Define x                ; Check value
if xmin xmax includes x ; Is x between xmin and xmax?
    Print "x>=3 and x<=10"
else
    if 5 8 excludes x   ; Is x > 8 or x < 5?
        Print "x is out of range"
    endif
endif
```

Looping Operations

do...while: The block executes before the logical condition is evaluated. If true, the block repeats; if false, execution continues after the while statement.

```
do                              ; Start of loop
    count = count + 1           ; Increment a counter
while Trig1 <> 0                ; Continue looping while trigger 1 is asserted
```

for...next: The condition is evaluated before the block executes. The incrementing variable is compared to operand2 on each pass. When equal, the loop terminates.

```
for I = 1 to 10     ; Counter I starts at 1, compared to 10 each pass
    x = x << 1      ; Shift x left by 1
next                ; Increment I before next evaluation
```

Constants

Constants are signed 32-bit integers, strings enclosed in double quotes, or 32-bit floating-point values. Examples: integer 42, string "Hello", float 3.14.

Common Variables

Sequence programs support nine variable types: Cosmic Integers, Public Integers, Global Integers, Local Integers, Floating-Point Numbers, Cosmic Strings, Public Strings, Global Strings, and Local Strings.

Variable definition examples:

Statement	Description
Define x	Create a local variable x
Define Global y	Create a global variable y
Define $a	Create a local string $a
Define Global $b	Create a global string $b
Define Cosmic s1	Create a cosmic variable s1
Define Cosmic $c	Create a cosmic string $c
Define Public z	Create a Public variable z (rPod8.4 only)
Define Public $s	Create a Public string $s (rPod8.4 only)
Define Float d	Create a global floating-point variable d

Public Variables: Exclusive to rPods. Defined on both processors and visible to all sequences.

Cosmic Variables: Saved to the CF card. Values are automatically restored on power-up from boot.ini.

Floating-Point Variables: Global to all sequences.

Variable naming rules: any combination of numbers and letters; no spaces or ASCII control characters; underscores permitted. Do not use reserved words. Avoid giving variables of different scope the same name to prevent shadowing.

Variables can be viewed using Print and modified through direct assignment. The Update, Define, and Context commands provide additional information on cosmic variables.

Trigger Variables

Trigger variables function similarly to Public variables but use a numerical reference. Up to 256 trigger variables can be defined, numbered ETRIG1 through ETRIG65535. All trigger values are signed 32-bit numbers. Like Public variables, triggers are common to both processors and all sequences, and must be defined in only one sequence on one processor. All triggers must be allocated with a Define statement.

Trigger variable examples:

Statement	Description
`Define ETRIG 2`	`Creates trigger 2 without initialization`
`Define ETRIG 300 = 20`	`Creates trigger 300 and initializes it to 20`
`ETRIG 50 = 5`	`Sets trigger 50 to 5`
`If ETRIG 2 == 1`	`Conditional: is trigger 2 equal to 1?`
`ETRIG 10 = 50`	`Assigns the value 50 to trigger 10`

Passing Processor Data (rPod8.4 Only)

The rPod8.4 has dual processors with separate scripting engines. Global and local variables are separate between processors, but Public variables are shared automatically. The pass command assigns variables in the other processor; the pass operator retrieves variables from the other processor.

Statement	Description
`Pass x to y`	`Store variable x on Ps to variable y on Pd`
`Pass 25 to y`	`Store constant 25 to y on Pd`
`Pass $b to $a`	`Store string $b on Ps to string $a on Pd`
`Pass "Hello" to $a`	`Store constant "Hello" on Ps to string $a on Pd`
`x = Pass y`	`Store variable y on Pd to variable x on Ps`
`$a = Pass $b`	`Store string $b on Pd to string $a on Ps`

> **NOTE:** *Ps = Source Processor, Pd = Destination Processor.*

String Operations

String functions manipulate local and global strings read from serial ports or generated within a sequence program. Functions can be called from a serial port, sequence program, or event list.

> **NOTE:** *Operations are limited to one operator per line. For example, x = Left a 3 + Right a 3 is INVALID.*

String Concatenation

Syntax: `"dest_string" = "string1" + "string2"`

Parameters:

`"dest_string"` — A defined local or global string variable.

`"string1", "string2"` — Local or global string variables, string constants, or system string variables.

Concatenates two ASCII strings and stores the result in the destination string.

Example	Description
`$a = $b + $c`	Concatenate variable strings
`$a = "String = " + $b`	Concatenate constant and variable string

String Comparison

Syntax: `"string1" == "string2"`

: `"string1" <> "string2"`

Parameters:

`"string1", "string2"` — Defined local or global string variables, string constants, or system string variables.

Returns 1 if the comparison is true; 0 if false.

Example	Description
`if $b == $c`	True if $b equals $c
`x = $a <> "Foobar"`	x = 1 if $a does not equal "Foobar"

String to Number

Syntax: `"operand" = Val "string"`

Parameters:

`"operand"` — Local or global integer variable. Receives the converted numeric result.

`"string"` — A defined local or global string variable, string constant, or system string variable (e.g., Console, DTE).

Converts an ASCII string to its equivalent integer value.

Example	Description
`x = Val $a`	`Variable string version`
`x = Val "32"`	`Constant string version`
`x = Val "0x10"`	`Hexadecimal string version`
`x = Val Console`	`String received on the Console port`

Number to String

Syntax: `"string" = Str "operand"`

Parameters:

`"string"` — A defined local or global string variable. Receives the converted ASCII string.

`"operand"` — Local or global integer variable or constant to convert.

Converts an integer variable or constant to an ASCII string.

Example	Description
`$a = Str 10`	`Constant value version`
`$a = Str count`	`Variable value version`

String Length

Syntax: `"operand" = Len "string"`

Parameters:

`"operand"` — Local or global integer variable. Receives the character count.

`"string"` — A valid local or global string variable, string constant, or system string variable.

Returns 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. Receives the 1-based position of the first match, or 0 if not found.

`"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. Returns the position (1-based), or 0 if not found.

```
$a = "Say Hello"
x = Instring $a "Hello"
Print x
5               ; "Hello" found at position 5
```

Left String

Syntax: `"string1" = Left "string2" "count"`

`: "operand" = Left "string2" "count"`

Parameters:

`"string1"` — Destination string — any defined local or global string variable.

`"operand"` — Destination integer — any defined local or global variable. When used, sums the ASCII values of the extracted characters instead of returning a string.

`"string2"` — Source string — any defined local or global string variable.

`"count"` — Number of characters to extract from the left of string2.

Extracts the left count characters from string2. When the destination is an integer, sums the ASCII values of those characters.

String destination example — get the left 4 characters of $a into $b:

```
$a = "Fire Truck"
$b = Left $a 4
Print $b
Fire
```

Integer destination example — get the ASCII value of the first character of $a:

```
$a = "ab"
x = Left $a 1
Print x
97              ; ASCII value of 'a'
```

Right String

Syntax: `"string1" = Right "string2" "count"`

`: "operand" = Right "string2" "count"`

Parameters:

`"string1"` — Destination string — any defined local or global string variable.

`"operand"` — Destination integer — any defined local or global variable. When used, sums the ASCII values of the extracted characters instead of returning a string.

`"string2"` — Source string — any defined local or global string variable.

`"count"` — Number of characters to extract from the right of string2.

Extracts the right count characters from string2. When the destination is an integer, sums the ASCII values of those characters.

String destination example — get the right 8 characters of $a into $b:

```
$b = Right $a 8
```

Integer destination example — get the ASCII value of the last character of $a:

```
$a = "ab"
x = Right $a 1
Print x
98              ; ASCII value of 'b'
```

Mid String

Syntax: `"string1" = Mid "string2" "start" "count"`

`: "operand" = Mid "string2" "start" "count"`

Parameters:

`"string1"` — Destination string — any defined local or global string variable.

`"operand"` — Destination integer — any defined local or global variable. When used, sums the ASCII values of the extracted characters instead of returning a string.

`"string2"` — Source string — any defined local or global string variable.

`"start"` — Character position at which to begin extraction (1-based).

`"count"` — Number of characters to extract.

Extracts count characters from string2 starting at position start. When the destination is an integer, sums the ASCII values of those characters.

String destination example — get 10 characters of $a starting at position 5:

```
$b = Mid $a 5 10
```

Integer destination example — sum the ASCII values of 2 characters starting at position 2 in "1ab2":

```
$a = "1ab2"
x = Mid $a 2 2
Print x
195             ; ASCII('a') + ASCII('b') = 97 + 98 = 195
```

Sound Channel Information

Sound Channel Output Information

Syntax: `"operand" = Cvol "Channel #"`

: `"operand2" = Cvold "Channel #"`

Parameters:

`"operand"` — Destination integer variable (local or global). Receives volume as 0–127, where 0 = off and 127 = maximum.

`"operand2"` — Destination floating-point or integer variable (local or global). Receives volume in decibels (–90 dB to 0 dB).

`"Channel #"` — Channel number, specified by a constant or a local or global variable.

Cvol returns the channel volume as a value from 0 to 127 (0 = off, 127 = max). Cvold returns the volume in decibels (–90 dB to 0 dB).

Example	Description
`x = Cvol 1`	`x will contain the volume of channel 1 (0-127)`
`f = Cvold 3`	`f will contain the volume of channel 3 in dB (-90 to 0)`

Sound Channel Input Gain Information

Syntax: `"operand" = Ivol "Channel"`

: `"operand2" = Ivold "Channel"`

Parameters:

`"operand"` — Destination integer variable (local or global). Receives gain as 0–127, where 0 = minimum gain and 127 = maximum gain.

`"operand2"` — Destination floating-point or integer variable (local or global). Receives gain in decibels (–90 dB to 0 dB).

`"Channel"` — Input channel identifier: MICL (left microphone), MICR (right microphone), LINL (left line input), or LINR (right line input).

Channel is one of: MICL, MICR, LINL, LINR. Ivol returns gain as 0–127 (0 = min, 127 = max). Ivold returns gain in dB (–90 dB to 0 dB).

Example	Description
`x = Ivol LINL`	`x will contain the left line input gain (0-127)`
`f = Ivold MICR`	`f will contain the right mic input gain in dB`

Sound Track Operations

This section provides functions to access track information for status and reporting purposes.

Single Track Information

Syntax: `"operand" = Tvar "Track #" "Fmt index/descriptor"`

`: "string" = Tvar "Track #" "Fmt index/descriptor"`

Parameters:

`"operand"` — Destination integer variable (local or global). Used when the requested parameter returns a numeric value.

`"string"` — Destination string variable (local or global). Used when the requested parameter returns a string value.

`"Track #"` — Track number — specified by a constant or a local or global variable.

`"Fmt index/descriptor"` — Sound track parameter index, specified by either the numeric index or the descriptor label from the table below. May be a constant, local variable, or global variable.

Obtains track information specified by the format index number and stores the result in the destination variable or string. The following table lists all available format index values:

Index	Descriptor	Numeric	String	Description
0	label			Format text label
1	pitch	✓		Track number
2	name		✓	Track name
3	channel	✓		Channel number
4	status	✓		Playback status
5	percent	✓		Percent complete
6	length	✓	✓	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

Example	Description
x = Tvar 2 13	x will contain the sample rate of track 2
$a = Tvar 3 2	$a will contain the sound file name assigned to track 3

Single Track Progress

Syntax: `"operand" = Progress "Track #" "mode"`

Parameters:

`"operand"` — Destination integer variable (local or global). Receives the time value in the current TIMEREF units.

`"Track #"` — Track number — specified by a constant or a local or global variable.

`"mode"` — Selects which time value to return (see table below). May be a constant, local variable, or global variable.

Mode	Description
0	Total time length of the sound
1	Time played so far
2	Time remaining to play

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

Track Lists

Syntax: `"string" = Tlist "Format string" "Track list string"`

Parameters:

`"string"` — Destination string variable (local or global). Receives the formatted report output.

`"format string"` — A local or global string variable or constant defining the report format. A comma-delimited list of track parameter descriptors and optional single-quoted label text.

`"track list string"` — A local or global string variable or constant specifying which tracks to include. Must be a string — inline constants such as 1,3 are not valid.

Creates a formatted report for all tracks specified in the track list string. The format string is a comma-delimited list of sound track parameters and optional single-quoted labels.

Example — the following format string uses two labels and two report variables:

```
$b = "'Track #',track,' is playing on ch# ',channel"
```

Then the function call — tracks 1 and 3:

```
$a = Tlist $b "1,3"     ; or
$a = Tlist $b $c        ; where $c = "1,3"
```

Printing $a produces:

```
Track #1 is playing on ch# 1
Track #3 is playing on ch# 3
```

> **NOTE:** *Track list must be contained in a string variable. $a = Tlist $b 1,3 is INVALID. Use %% to produce a literal % character.*

System Variables

System variables are pre-defined by the system and may be referenced in sequences. These variable 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:

Variable	Value	Description
?Pn	0	Audio track n is not playing
?Pn	1	Audio track n is playing
?Pn	2	Audio track n is looping
?Pn	3	Audio track n is pre-loaded in buffer
?Pn	4	Audio track n is waiting for buffer closure
?Pn	5	Audio track n is paused
?Tn	0–100	Audio track n completion (0 = start, 100 = end)
?Dn	0	DMX track n is not playing
?Dn	1	DMX track n is playing
?Dn	2	DMX track n is looping
?Dn	3	DMX track n is pre-loaded in buffer
?Dn	4	DMX track n is waiting for buffer closure

The variable Source (rPod8.4 only) determines the codec source:

Source Value	Description
0 = Split	Ch 1-4, 9 → P1; Ch 5-8, 10 → 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 access and/or modify information for Clips already loaded into memory. Changes are temporary and are not written to the CF card. Syntax: ClipName.ClipParameter or ClipName.TrackName.TrackParameter.

Example — retrieving the audio file name from a loaded clip:

```
Define $n
Ldclip cApollo                  ; Load the clip cApollo.clip
$n = Apollo.audiofilename       ; Retrieve the wave file name
```

Available Clip Parameters:

Clip Parameter	Numeric	String	Description
.name		✓	Clip name (main reference for clip)
.id	✓		Clip identification number
.audiofilename		✓	Name of the audio file
.channels	✓		Number of channels in the audio file
.trackcount	✓		Number of associated tracks

Example — temporarily changing a pre-determined track volume before starting a clip:

```
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:** *The track change will not take effect until the next Start C cApollo command is issued. Since Start does not re-load the clip, it will not change the clip file. If the clip is re-loaded, the previous track volume will be restored.*

To change the volume of an actively playing clip track:

```
Tvol Apollo.apollo1.track = 80  ; Change an active clip's track volume
```

Available Track Parameters:

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

System Information

Access system version, manufacturing date, and MAC address through the INFO system variable. Example output of Print INFO:

```
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 counts dual-port memory data transfer misses; it resets to zero after each read.

Ethernet

Variable	Description
IPADD	IP address of the unit
IPPORT	IP port
IPMASK	IP subnet mask
GATEWAY	Default gateway IP address of the unit
UDP	Contents of the active UDP packet
INFO	Version, manufacturing date, MAC address
SIGPORT	Port for Sigma DSP Ethernet programming

UDP Variable	Description
?ETYPE	User-defined variable list configuration (0-65535)
?EORIG	Source bind number (1-255)
?ECOUNT	Number of variables received
?EPKT	Packet number from sender (0-65535)
?ENET	UDP buffer flag: 0=no data, 1=variables, 2=raw UDP

The EENABLE variable enables or disables UDP data reception. Example:

```
if EENABLE == 0      ; Is UDP reception disabled?
    EENABLE = 1      ; Yes, enable UDP reception
else
    EENABLE = 0      ; No, disable UDP reception
endif
```

Serial Communications

Variables ?Console, ?DTE, and ?DMX flag read completion of the respective serial ports. When termination is on, they return 1 (complete) or 0 (not complete). When termination is off, they return the number of characters in the receive buffer.

Example — reading the DTE port and responding to a command:

```
M0  Read DTE              ; Command DTE port to look for a string
    if ?DTE == 1          ; Has a string been received?
        if DTE == "Play"  ; Compare received string to "Play"
            Play Flim on 1 ; If match, play the sound file "Flim"
        endif
    endif
    Goto M0
```

Triggers

Four system variables determine the status of the Input Trigger Port:

Variable	Description
Inputs	Trigger word — current state of all inputs as a binary-weighted byte
Trig1…Trig8	Trigger input — current state of a single input (1 = asserted, 0 = not)
Close1…Close8	Trigger assertion — sticky edge-detect (1 = edge detected since last read)
Open1…Open8	Trigger removal — sticky edge-detect (1 = removal detected since last read)

The Inputs variable returns all trigger states as a binary-weighted byte. Bit 0 = Input 1, Bit 7 = Input 8:

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

Reading Inputs example (triggers 1, 2, and 4 asserted):

```
x = Inputs
Print x
11              ; Bits 0+1+3 = 1+2+8 = 11
```

Reading a single trigger:

```
x = Trig5
Print x
1               ; Trigger 5 is asserted
```

Edge detection using Close and Open — each read resets detection until the next event:

```
x = Close2      ; Read trigger 2 close edge
Print x
1               ; Trigger 2 edge was detected
x = Close2      ; Read again
Print x
0               ; Trigger 2 edge was not detected (reset by first read)
```

Expansion Triggers

External trigger expansion uses one input as a serial connection, supporting up to 1024 unique triggers from a 5–10 bit serial word.

Command	Description
Config TEXP on/off	Enable or disable serial trigger expansion mode
Config TEXP input1 n	First trigger input (1-8 MS3; 1-12 rPod)
Config TEXP input2 n	Second (mirror) trigger input
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 level polarity
Config TEXP stops n	Number of stop bits (0-2)
Config TEXP order lsb/msb	Transmit order

XTRDY: Trigger received flag (0 = not ready, 1 = ready). Clear to 0 after detecting readiness. XTRIG: Trigger number received; valid after XTRDY = 1.

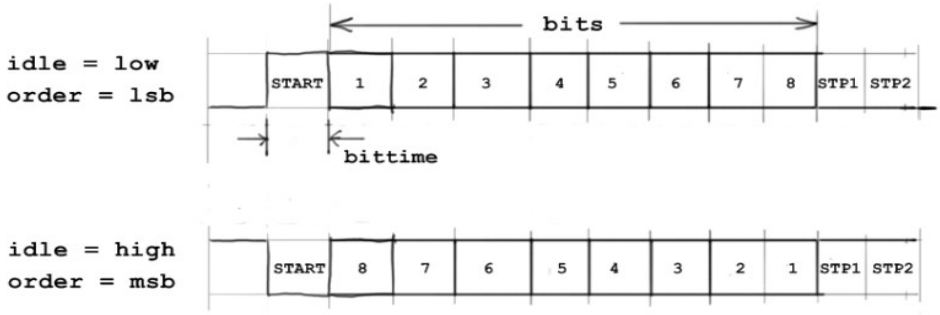

Figure 1 — Serial trigger word format showing bit ordering for LSB-first (idle low) and MSB-first (idle high) configurations. The bit period is set by Config TEXP bittime. Stop bits STP1 and STP2 follow the data word.

Trigger expansion example:

```
; Trigger expansion example
Define trig
Config TEXP 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 ready flag
        trig = XTRIG        ; Assign trigger to variable
        if trig == 1        ; Was the trigger a 1?
            Play sound1 on 1 T1
        endif
        if trig == 2        ; Was the trigger a 2?
            Play sound2 on 1 T1
        endif
    endif
    Goto Main
```

DMX Input

Dmxin accesses any DMX channel from 1 to 512. The nonzero, zero, and change options compare the previous and current channel values; a baseline read is required.

Option	Description
value	Reads the instantaneous value
nonzero	Detects an edge change from zero to nonzero
zero	Detects an edge change from nonzero to zero
change	Detects any change in value

DMX input example — establish baselines, then monitor for changes:

```
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 edge change from zero to nonzero
if x <> 0              ; x returns 0 for no change, 1 for a change
    Play flim on 1 T1
endif
x = Dmxin 2 zero       ; Look for edge change from nonzero to zero
if x <> 0
    Play Apollo 13 on 1 T2
endif
x = Dmxin 3 change     ; Look for any change on DMX channel 3
if x <> 0
    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 ?FILE function checks for a file on the CF card. A quote-delimited string, Console, or DTE may be used as the file reference. Example:

```
J1  Read Console              ; Put Console in read mode
    if ?Console == 1          ; Has a string been received?
        if ?FILE Console == 1 ; Is the console input a valid sound file?
            Play Console on 1 ; Yes, play it
        else
            if ?FILE "Flim" == 1 ; Does Flim exist?
                Play Flim on 1   ; Yes, play it
            endif
        endif
    endif
    Goto J1
```

?CF1 and ?CF2 return 0 (card missing) or 1 (card installed). ?CF2 applies to the rPod only.

Random Numbers

Syntax: `"value" = Rand "start" "end"`

Parameters:

`"value"` — Destination integer variable (local or global). Receives the pseudo-random result.

`"start"` — Lower bound of the random number range (inclusive).

`"end"` — Upper bound of the random number range (inclusive).

Returns a pseudo-random integer seeded from the real-time clock at power-up. Example — return a random number between 1 and 10:

```
index = Rand 1 10
```

Real Time Clock

Set and read using CLOCK. Entry format: MMM DD, YYYY hh:mm:ss.

```
CLOCK = Jun 4, 2008 11:24:00
```

To print the current time:

```
Print CLOCK
06/04/08 11:26:01 AM
```

Sub-field access examples — Console entry:

```
A:> Define x
A:> x = CLOCK.DAY
A:> Print x
4               ; Displayed on a Wednesday
```

Variable	Description
CLOCK.MONTH	Month number (1=January to 12=December)
CLOCK.DATE	Day of the month (1-31)
CLOCK.YEAR	Year (1900-3000)
CLOCK.HOUR	Hour (00-23, military format)
CLOCK.MINUTE	Minute (00-59)
CLOCK.SECONDS	Seconds (00-59)
CLOCK.DAY	Day of the week (1=Sunday to 7=Saturday)
CLOCK.NUMBER	Numerical representation of CLOCK

Clock example — sequence program that chimes 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

An incrementing time-of-day counter based on the system clock, MIDI, or SMPTE. Disabled at startup, common to all sequences. The constant format for the timer is hh:mm:ss.ff (frames) or hh:mm:ss.mmm (ms/mixed).

To set the timer to 5 minutes 30 seconds and 20 frames:

```
TIMER = 5:30.20     ; or
TIMER = 5:30.660    ; equivalent in ms mode
```

Timer source/destination ports:

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

To synchronize the timer to SMPTE:

```
Tsync = SMPTE
```

To set the timer destination to MIDI time code:

```
Tdest = MIDI1
```

Mode	Description
on	Starts incrementing; optional timer offset may be specified
off	Stops timer and event cueing
hold	Stops timer but permits manual event cueing
stop	Stops timer, pauses all sound and DMX playback (General Stop)
resume	Same as on, plus resumes all paused sound and DMX playback

Timer sub-element access — when the timer source is LOCAL, individual fields can be read and written:

Field	Read Example	Write Example
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 independent timers, Tmr1–Tmr8, operate identically to the system timer but are independent of it and each other. Each has on and off modes.

To set timer 5 to 5 minutes 30 seconds and 20 frames:

```
Tmr5 = 5:30.20
```

Valid timer commands:

```
Tmr1 = 0            ; Reset timer 1 to zero
Tmr1 on             ; Start timer 1
Tmr1 off            ; Stop timer 1
Print Tmr1
00:00:00:05.22      ; Current value
Tmr2 = Tmr1         ; Copy Tmr1 to Tmr2
Print Tmr2
00:00:00:05.22
x = Tmr1            ; Assign Tmr1 to numeric variable
Print x
172                 ; Value in frames
Tmr2 = x + 5        ; Assign computed value to Tmr2
Print Tmr2
00:00:00:05.27
```

A sequence using timer 3 as a 10-second interval timer:

```
; 10-second timer example
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
    endif
    Goto J1
```

Timer-Based Actions

Timer events specify a precise time to execute commands, operations, or functions. A cue point uses the format [hh:mm:ss.ff] (frames) or [hh:mm:ss.mmm] (ms/mixed). Timer events can appear in the sequence body or in event lists.

Event Lists

A linear show-control tool. Up to 64 event lists can be distributed among the 16 sequence programs, but only one may be active at any given time. The general format is:

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

Example event list:

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

> **NOTE:** *All conditional, looping, or branching instructions are ignored in event lists. Only single-line executable instructions may be included. Timer events must be in chronological order.*
>
> **NOTE:** *When the TIMER rolls over from 23:59:59.29 to 00:00:00.00, all events are re-initialized. TIMER.DAYS is ignored in all event list timing calculations.*

Event List Behavior

The active event list maintains a pointer that increments as timer events are processed. The following table describes event list and timer action behavior based on timer mode changes:

From	To	Action
TIMER off	TIMER on	Position list pointer, enable timer, activate event list
TIMER off	TIMER hold	Position list pointer, disable timer, activate event list
TIMER on	TIMER off	Disable timer and event list
TIMER on	TIMER hold	Disable timer; event list remains active
TIMER hold	TIMER on	Activate timer; event list active
TIMER hold	TIMER off	Disable timer and event list

- The list pointer repositions whenever the timer is manually adjusted or the event source changes.
- Setting the timer to an earlier time performs no events.
- Setting the timer to a later time executes all events between the old and new pointer positions if the timer mode is on or hold. In off mode, no events are performed.

Embedded Timer Points

Timer points embedded in the sequence program body provide pause points for a show. If the timer has already passed the event when the line executes, the instruction runs immediately.

Example 1 — sequence using embedded timer points:

```
TIMER = 0              ; Initialize the timer
TIMER on               ; Enable the timer
Main
    [1:32.25]  Play Flim on 1 T1     ; Wait until 1:32:25, then play
    [5:51.00]  Play Apollo 13 on 1 T1 ; Wait until 5:51:00, then play
    do                 ; Wait for Apollo 13 to finish
    while ?P1 <> 0
    TIMER = 0          ; Reset the timer
    Goto Main          ; Loop to beginning
```

Example 2 — using the real-time clock as a timer reference to play a show at noon every day:

```
TIMER = CLOCK          ; Initialize timer to real-time clock
TIMER on 11:00:00      ; Enable timer with 11 AM offset
Main
    [1:00:00]  Play Flim on 1 T1     ; Play at 12:00:00 (11:00 + 1:00)
    [1:10:00]  Play Apollo 13 on 1 T1 ; Play at 12:10:00
    Goto Main
```

Example 3 — event list with trigger-based pause and resume, cycling DMX output bits:

```
EVENT = mainshow       ; Select the event list
Tsync = LOCAL          ; Set 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     ; Trigger 1 closes: pause the show
        TIMER off
        Print # "Stop show", 13, 10
    endif
    if Close2 == 1     ; Trigger 2 closes: resume the show
        TIMER on
        Print # "Start show", 13, 10
    endif
    Goto J1

Eventlist   mainshow
[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
```

Cue Lists

Cue lists provide dynamic show control independent of the event list system. Maximum 2048 concurrent cues. Each cue can belong to a group with a relative timer offset.

Example — loading cues on the fly when triggers are detected:

```
Define t1 = 300        ; 10 seconds (in frames)
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
        TIMER on
    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
        TIMER on
    endif
    if Close3 == 1
        TIMER off
        TIMER = 0
    endif
    Goto J1
```

rPod8.4 Considerations

The rPod8.4 has dual processors with separate scripting engines and CF card slots. A transparent inter-processor communications interface is provided.

Physical resource assignments:

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	

Audio channel mapping (CMAP) mode assignments:

CMAP Mode	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

- In SPLIT mode, CF1 files play on channels 1–4 and subs 9–10; CF2 files play on channels 5–8 and subs 11–12.
- Control files should reside on CF card slot 1 (accessible from both processors).
- Global and local variables are processor-specific. Use the Pass operator to exchange data between processors.
- Operating system loader files (.ldr) must reside on the respective CF card (CF1 for P1, CF2 for P2).
- Each processor has a separate system timer derived from the same master clock. P1 timer: Console port and P1 sequences only. P2 timer: DTE port and P2 sequences only.

Shared Resource	Notes
Audio and DMX tracks	Common set for both processors
Triggers and output controls	Either processor may access
Console and DTE ports	Either processor may access
Analog inputs	Mapped to both processors
System clock	Either processor may access

Shared Resource	Notes
Ethernet data	Shared by both processors

Table of Operations

Operation Type	Syntax
Direct	"operand1" = "operand2"
Logical, single	"operand1" = "single operator" "operand2"
Logical, double	"operand1" = "operand2" "logical operator" "operand3"
Arithmetic	"operand1" = "operand2" "math operator" "operand3"
If, evaluate	if "operand1"
If, single op	if "single operator" "operand1"
If, double op	if "operand1" "operator" "operand2"
While, evaluate	while "operand1"
While, single op	while "single operator" "operand1"
While, double op	while "operand1" = "operand2"
For loop	for "operand1" = "operand2" to "operand3"

Table of Operators

Operator	Symbol
Increment	++
Decrement	--
Logical NOT	!
Bitwise Complement	~
Subtraction	-
Addition	+
Multiplication	*
Division	/
Modulus	%
Logical AND	&&
Bitwise AND	&
Logical OR	||

Operator	Symbol
Bitwise OR	|
Bitwise XOR	^
Equality	==
Inequality	<>
Less than or equal to	<=
Greater than or equal to	>=
Shift Left	<<
Shift Right	>>
Less than	<
Greater than	>
In range	includes
Out of range	excludes

Table of System Constants and Subroutines

Category	Examples / Identifiers
Constants	25, 15, ...
Local variables	counter, flag, ...
Global variables	mode, status, ...
Serial input status	?Console, ?DTE
File status	?FILE "filename", ?FILE Console, ?FILE DTE
System strings	$mystring, Console, DTE, "" delimited
String functions	Val, Str, Len, Instring, Left, Right, Mid
Random number	Rand
Trigger input	Trig1, Trig2, ...
Trigger assertion	Close1, Close2, ...
Trigger removal	Open1, Open2, ...
Trigger word	Inputs
DMX output	DMX1, DMX2, ...
System status	STATUS
Timer	Tmode, TIMER...
Real-time clock	CLOCK
Audio track playing	?P1, ?P2, ...

Category	Examples / Identifiers
Audio track % complete	?T1, ?T2, ...
DMX track playing	?D1, ?D2, ...
DMX frame rate	Framerate
Audio inputs (rPod only)	MICL, MICR, LINL, LINR
Multimover (MS2 only)	?MMOVER, MMCMD, MMDATA

Command Description Index

This section provides detailed descriptions, syntax, parameters, and examples for all system commands. Unless otherwise noted, all commands listed here can be incorporated directly into a sequence program.

> **NOTE:** *All direct commands from this index are also available as sequence commands.*

File Access Commands

Change Drive

Syntax: `cd "drive name"`

Parameters:

`"drive name"` — Drive letter: "a" (CF slot 1) or "b" (CF slot 2).

Both CF card slots are active, with slot 1 assigned as drive a: and slot 2 as drive b:. To select drive b:

Example:

```
cd b
cd a    ; To return to drive a:
```

On the rPod8.4, changing the drive also redirects control to the associated processor, allowing both the Console and DTE to control the same processor if desired.

Display Directory

Syntax: `dir`

Displays the contents of the currently selected directory on the active drive.

Example:

```
dir
```

Change Directory

Syntax: `Chdir "directory name"`

Parameters:

`"directory name"` — Name of the directory to navigate to.

Opens the specified directory on the current drive. Use dot notation (..) to move to the parent directory. Typing Chdir with no parameter prints the current directory path. The current path is also accessible through the system string variable WORKDIR.

Example:

```
Chdir "Mydir\sound files"
Play "Mydir\sound files\sound1"    ; Play a file in the directory
Chdir ..                           ; Move to parent directory
$a = WORKDIR                       ; Store the current path in $a
```

Make Directory

Syntax: `Makedir "directory name"`

Parameters:

`"directory name"` — Name of the new directory to create.

Creates a new directory on the currently selected drive.

Example:

```
Makedir "sound files"
```

Remove Directory

Syntax: `Remdir "directory name"`

Parameters:

`"directory name"` — Name of the directory to remove.

Removes the specified directory from the currently selected drive.

Example:

```
Remdir "sound files"
```

Delete File

Syntax: `del "file name"`

Parameters:

`"file name"` — File name on the CF card. The drive prefix (a:\ or b:\) and dot extension are optional.

Deletes the specified file from the currently selected drive.

Example:

```
del Flim
```

Rename File

Syntax: `Rename "original file name" as "new file name"`

Parameters:

`"original file name"` — Existing file name. Drive prefix and dot extension are optional.

`"new file name"` — New file name. Drive prefix and dot extension are optional.

Renames the specified file on the currently selected drive.

Example:

```
Rename Flim as Flimbackup
```

Open File

Syntax: `filenumber = Openfile "file name" for "filemode"`

Parameters:

`"filenumber"` — A file reference variable returned by the command. Must be a previously defined local or global variable.

`"file name"` — File name on the CF card. Drive prefix and extension are optional.

`"filemode"` — Access mode flag (see table below).

Opens an existing file or creates a new file for reading or writing. A companion Closefile command must be issued for every Openfile. The returned file number is subsequently used by Closefile, Writefile, Readfile, and Seekfile.

File Mode	Description
"r"	Open existing file for reading. Pointer at beginning.
"r+"	Open existing file for reading and writing. Pointer at beginning.
"w"	Create new file or clear existing file for writing. Pointer at beginning.
"w+"	Open for reading and writing; create if absent or clear if present. Pointer at beginning.
"a"	Open for appending (write to end). Creates file if absent. Pointer at end.
"a+"	Open for reading and appending. Creates file if absent. Pointer at end.

Example — creating a new file for writing:

```
Define Fn
Fn = Openfile data.txt for w
```

Close File

Syntax: `Closefile "filenumber"`

Parameters:

`"filenumber"` — File reference number returned by Openfile.

Closes a previously opened file. A Closefile must be issued for every Openfile.

Example:

```
Closefile Fn
```

File Configuration (Delimiter)

Syntax: `Config FDELIM = "character list"`

Parameters:

`"character list"` — Comma-delimited list of ASCII codes (decimal or hex), local variables, or global variables.

Sets the delimiter string used to separate data in Writefile and Readfile commands. The default delimiter is a comma (ASCII 44).

Example:

```
Config FDELIM = 9, 0x20     ; Change delimiter to tab and space
Config FDELIM = 44          ; Reset to comma
```

Write File

Syntax: `Writefile "filenumber" "data list"`

Syntax: `Writefile # "filenumber" "data list"`

Parameters:

`"filenumber"` — File reference number returned by Openfile.

`"data list"` — Comma-delimited list of numbers, variables, and/or strings.

Writes data to a previously opened file. The file must not be opened in read-only mode (r).

Writefile uses the system delimiter between values, making them compatible with Readfile.

Writefile # writes raw binary data without delimiters. Placing a # before a variable causes it to be written as a single byte instead of a 4-byte word.

Example:

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

Example — saving channel volume settings to a binary file:

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

> **NOTE:** *FDELIM is the system string variable for the current delimiter. FERR holds the result of the last file operation: 0 = success, 2 = failure.*

Read File

Syntax: `Readfile "filenumber" "variable list"`

Syntax: `Readfile # "filenumber" "variable list"`

Syntax: `Readfile $ "filenumber" "string variable" <count>`

Parameters:

`"filenumber"` — File reference number returned by Openfile.

`"variable list"` — List of local or global variables (or strings) to receive data.

`"string variable"` — String variable for the $ variant.

`"count"` — Optional number of characters to read ($ variant). If omitted, null termination is used.

Reads data from a previously opened file. The file must not be opened in write-only mode (w or a).

Readfile: reads using the system delimiter.

Readfile $: reads raw string data into a single string variable.

Readfile #: reads raw binary data; a # before a variable reads 1 byte, a plain variable reads 4 bytes.

Example:

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

Example — reading binary volume settings:

```
Define volume1
Define Volume2
Fn = Openfile config.hex for r
Readfile # Fn #Volume1, #Volume2
Closefile Fn
Print volume1, " ", volume2
70 110
```

NOTE: *FERR = 0 (success), FERR = 1 (success, end-of-file detected), FERR = 2 (failed).*

Seek File

Syntax: `"position" = Seekfile "filenumber" "offset" <R> {CURR/BEG/END}`

Parameters:

`"position"` — Returns the file character position after the seek.

`"filenumber"` — File reference number returned by Openfile.

`"offset"` — Byte offset relative to CURR, BEG, or END.

`<R>` — Optional record flag. Uses FDELIM to locate a specific record offset.

`CURR / BEG / END` — CURR (default) = current position, BEG = beginning, END = backward from end.

Positions the file pointer in a previously opened file. If the file "data" contains the records: 254, 32, "this is some text", 850, 2000 — then the following commands show typical usage:

Example:

```
Define Fn
Define a
Define p
Fn = Openfile data for r
p = Seekfile Fn 1 R BEG     ; Position to second record
Print p
4                           ; File position returned
Readfile Fn a               ; Read second record
```

```
Print a
32                          ; Value of second record
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

The pre-loading feature loads sound and DMX files into memory buffers in advance. Playback begins upon the next Start command, enabling sample-accurate synchronization of multiple files and eliminating time lag from buffer cueing.

Pre-load a Sound

Syntax: `Ldsnd "file name" on "ch # list" T "track # list" X "cross-fade list"`

Syntax: `Ldsndm "file name" on "ch # list" T "track # list" X "cross-fade list"`

Parameters:

`"file name"` — Sound file name on the CF card. Drive prefix and extension are optional.

`"ch # list"` — Output channel number list. Multiple channels separated by commas.

`T "track # list"` — Optional: manually assign specific tracks.

`X "cross-fade list"` — Optional: list of currently playing tracks to cross-fade with.

Pre-loads a sound file into memory without starting playback. Ldsndm pre-loads in mono mode. The .wav extension may be included or omitted.

Using the X flag causes the specified currently playing tracks to decay out when the new tracks are started.

Example	Description
`Define ch5 = 5`	Variable definition
`Ldsnd Flim on 5`	Pre-load stereo on channel 5
`Ldsndm Flim on 5`	Pre-load mono on channel 5
`Ldsnd Flim on ch5`	Pre-load using variable reference

Pre-loading to multiple channels and starting synchronized playback:

```
Define ch5 = 5
Define ch7 = 7
Ldsnd "Bell Tree" on 1,3    ; Pre-load to channels 1 and 3
```

```
Ldsnd "Bell Tree" on ch5, ch7 ; Pre-load using variables
Start                        ; Begin synchronized playback
```

Optional track assignment — manually specify track numbers:

```
Ldsnd "Bell Tree" on 1,3,5 T 1,2,3
```

Optional cross-fade — fade out tracks 1, 2, 3 while fading in new tracks 4, 5, 6:

```
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
; ... after playing for a while ...
Ldsnd "Bell Tree" on 1,3,5 T 4,5,6 X 1,2,3
Start
```

Unload a Sound Track

Syntax: `Unload "track # list"`

Parameters:

`"track # list"` — Tracks to unload. Tracks may be referenced by local or global variables.

Releases pre-loaded sound tracks from memory. Unloaded tracks will not play when a Start command is issued.

Example:

```
Unload 1    ; Remove pre-loaded track 1
```

Play Sound

Syntax: `Play "file name" on "ch # list" T "track # list" X "cross-fade list"`

Syntax: `Playm "file name" on "ch # list" T "track # list" X "cross-fade list"`

Parameters:

`"file name"` — Sound file name.

`"ch # list"` — Output channel number list.

`T "track # list"` — Optional: manual track assignment.

`X "cross-fade list"` — Optional: tracks to cross-fade with.

Loads and begins playback of the specified sound file immediately. Playm plays in mono mode. If a stereo file is specified, it plays on the specified channel and the next channel.

Example	Description
`Define ch5 = 5`	Variable definition
`Play Flim on 5`	Stereo playback on channel 5
`Playm Flim on 5`	Mono playback on channel 5
`Play Flim on ch5`	Playback using variable reference

Play on multiple channels using variables:

```
Define dog = 1
Define cat = 3
Define pig = 5
Play "Bell Tree" on dog, cat, pig
```

Optional track assignment:

```
Play "Bell Tree" on 1,3,5 T 1,2,3
```

Optional cross-fade — cross fade "Bell Tree" with sounds on tracks 2, 3, and 4:

```
Play "Bell Tree" on 1 X 2,3,4
```

Loop Sound

Syntax: `Loop "file name" on "ch # list" T "track # list" X "cross-fade list"`

Syntax: `Loopm "file name" on "ch # list" T "track # list" X "cross-fade list"`

Parameters:

`"file name"` — Sound file name.

`"ch # list"` — Output channel list.

`T / X` — Optional track assignment and cross-fade flags (see Play Sound).

Loads and continuously loops the specified sound file. Loopm operates in mono mode.

Example:

```
Loop Flim on 1       ; Stereo loop
Loopm Flim on 1      ; Mono loop
Loop Flim on 1 T2    ; Loop with manual track assignment
```

Un-loop Track

Syntax: `Unloop "track # list"`

Parameters:

`"track # list"` — Track(s) to convert from looping to non-looping mode.

Cancels looping mode. The track will stop when it reaches the end of the sound.

Example:

```
Unloop 1,2
```

Convert to Looping

Syntax: `Makeloop "track # list"`

Parameters:

`"track # list"` — Track(s) to convert to looping mode.

Converts a non-looping track to looping mode.

Example:

```
Makeloop 3
```

Append Sound

Syntax: `Append "file name" on "track #"`

Parameters:

`"file name"` — Sound file to append.

`"track #"` — Active track number to append to.

Queues the specified sound file to begin playing when the current file completes. The track must be active. Appended sounds should have the same sample rate and playback style (mono/stereo).

Example:

```
Append "Bell Tree" on 1
```

Stop Sound Tracks

Syntax: `Stop "track # list"`

Syntax: `Stop all`

Parameters:

`"track # list"` — List of actively playing tracks to stop.

Stops playback on the specified tracks. If a track has a non-zero decay setting, it decays to zero before stopping.

Example:

```
Stop 1,2
Stop all
```

Pause / Resume Sound Tracks

Syntax: `Pause "track # list"`

Syntax: `Pause all`

Syntax: `Resume "track # list"`

Syntax: `Resume all`

Parameters:

`"track # list"` — List of actively playing tracks.

Pause freezes playback on the specified tracks while others continue. Resume restores playback from the paused point.

Example:

```
Pause 2,3    ; Pause tracks 2 and 3 while 1 and 4 continue
Pause all
Resume 2,3   ; or
Resume all
```

Mirror Sound Tracks (rPod Only)

Syntax: `Mirror "track # list" on/off`

Parameters:

`"track # list"` — List of actively playing tracks.

Allows a single file and track to play back on multiple channels controlled by the same processor (P1: channels 1–4; P2: channels 5–8).

Example:

```
Mirror 1 on
Play tone on 1 T1    ; Plays on channels 1, 2, 3, and 4 when mirrored
```

Channel mirroring behavior by file type and play command:

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

Position Sound Track

Syntax: `Pos "track # list" to F/S/T "position reference"`

Syntax: `Skip "track # list" <</>>/F/S/T "position reference"`

Parameters:

`"track # list"` — List of track numbers.

`F / S / T` — F = frames (1/30 sec), S = samples (44,100/sec), T = time in ms (default).

`"position reference"` — Position value. For Skip, use >> (forward) or << (backward).

Pos repositions playback from the beginning of the file. Skip repositions relative to the current position.

Example	Description
Pos 1,3 to F300	Position tracks 1 and 3 at frame 300 (10 seconds)
Pos 1,3 to S441000	Position at sample 441,000 (10 seconds at 44.1K)
Pos 1,3 to T10000	Position at 10,000 ms (10 seconds)
Skip 1 << F1890	Skip backward 1890 frames (63 seconds)
Skip 1 << S2778300	Skip backward 2,778,300 samples (63 seconds)
Skip 1 << 1:03	Skip backward 1 minute 3 seconds

Pitch Sound Track

Syntax: `Pitch "track # list" to "percent x 10" <in "time (ms)">`

Parameters:

`"track # list"` — List of actively playing tracks.

`"percent x 10"` — Playback rate as a percentage × 10. Range: 0–2000. 1000 = normal, 2000 = 2x, 500 = half.

`"time (ms)"` — Optional ramp time in milliseconds. If omitted, the change is immediate.

Alters the playback sample rate of the specified tracks. Cannot be used when a track is timer-locked.

Example:

```
Pitch 1,2 to 1100 in 10000    ; Pitch to 110% over 10 seconds
```

Set Global Playback Sample Rate

Syntax: `Config SAMPLE "sample rate"`

Parameters:

`"sample rate"` — Sample rate in samples per second. Default is 44100.

Sets the audio sample rate for all channels. Not all sample rates can be achieved; the selected rate will be returned.

Example:

```
Config SAMPLE 48000
```

Sound Volume Control

Channel volumes are initialized at boot in the following order: (1) default value of 64 is applied; (2) saved Cvol settings in the .ini file are applied; (3) if a codec configuration file is present, its Cvol settings override the .ini values. After boot, any changes via sequence or console take effect immediately.

Channel Volume Control

Syntax: `Cvol "ch # list" = "volume"`

Parameters:

`"ch # list"` — Output channel list. Multiple channels separated by commas.

`"volume"` — Volume from 0 (off) to 127 (max), or in decibels (–90 dB to 0 dB).

Sets the hardware output volume for the specified channels using the codec gain control.

Example	Description
`Cvol 1,2,3,4 = 64`	`Set channels 1-4 to volume 64`
`Define vol1 = 64  /  Cvol 1,2,3,4 = vol1`	`Using a variable`
`Cvol ch1, ch2, ch3 = vol1`	`Using channel variables`
`Cvol 6,7 = -10dB`	`Set channels 6 and 7 to -10 dB`

Channel relationships by model:

rPod10.2 Primary	rPod10.2 Sub	rPod8.4 Primary	rPod8.4 Sub	MS-II Primary	MS-II Sub
1, 2	11	1, 2	9	1, 2	3
3, 4	12	3, 4	10		
5-10	none	5, 6	11		
		7, 8	12		

Track Volume Control

Syntax: `Tvol "track # list" = "volume"`

Parameters:

`"track # list"` — Track number list.

`"volume"` — Volume from 0 (off) to 127 (max), or in dB (–90 dB to 0 dB).

Sets the software volume for the specified tracks. Track volume is scaled in software and is affected by channel volume and any codec DSP gains or attenuations.

Example:

```
Tvol 1,5 = 54
Tvol 1,5 = -3dB
```

Ducking Volume Setting

Syntax: `Dvol "track # list" = "volume"`

Parameters:

`"track # list"` — Track number list.

`"volume"` — Duck volume from 0 to 127 or in dB.

Sets the ducking volume level for the specified tracks.

Example:

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

Duck Track Control

Syntax: `Duck "track # list"`

Parameters:

`"track # list"` — Track number list.

Sets the volume of the specified tracks to the ducking volume level. Overrides track volume until Unduck is received.

Example:

```
Duck 1,2
```

Un-Duck Track Control

Syntax: `Unduck "track # list"`

Parameters:

`"track # list"` — Track number list.

Restores the specified tracks from ducking volume to normal track volume (Tvol).

Example:

```
Unduck 1,2
```

Input Volume Setting

Syntax: `Ivol "ch # list" = "volume"`

Parameters:

`"ch # list"` — Input channel list. rPod: MICL, MICR, LINL, LINR. MS-II: INL, INR. Use "all" for both L/R.

`"volume"` — Volume from 0 to 127.

Sets the input gain for the specified analog input channels.

Example:

```
Ivol MICL, MICR, LINL, LINR = 80    ; rPod
Ivol all = 80                        ; MS-II
```

Patch Assignment (rPod10.2 and rPod8.4 Only)

Syntax: `Patch A/B = "input ch list"`

Parameters:

`A/B` — Patch assignment A or B.

`"input ch list"` — Input channels: MICL, MICR, LINL, LINR.

Assigns source input channels to the specified patch. The system supports two patches (A and B).

Example:

```
Patch A = MICL
Patch B = LINL, LINR
```

Input Mapping (rPod10.2 and rPod8.4 Only)

Syntax: `Patch A/B to "output ch# list"`

Parameters:

`A/B` — Patch assignment A or B.

`"ch # list"` — Output channel list.

Assigns destination output channels to the specified patch.

Example:

```
Patch A to 1,3,5    ; Odd channels to patch A
Patch B to 2,4,6    ; Even channels to patch B
```

Input Control (rPod10.2 and rPod8.4 Only)

Syntax: `Patch A/B on/off`

Parameters:

`A/B` — Patch assignment A or B.

`on/off` — on = connect; off = disconnect.

Connects or disconnects the specified patch.

Example:

```
Patch A on
```

The diagram below illustrates a complete patch configuration routing MICL to channels 5 and 7 via Patch A, and MICR to channels 6 and 8 via Patch B.

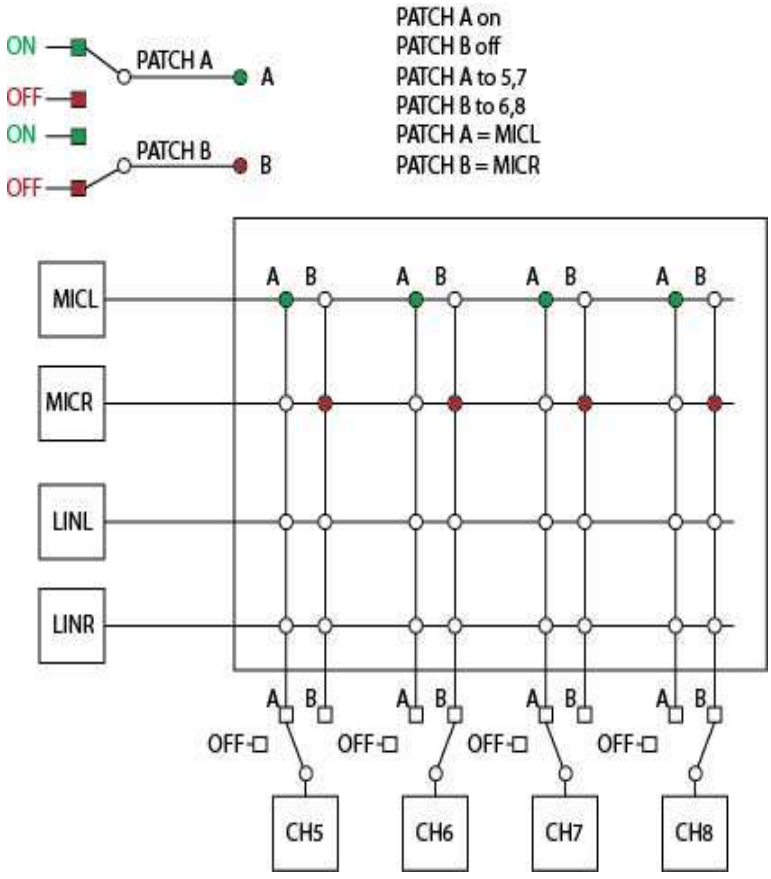

Figure 11 — Patch signal routing block diagram showing Patch A (MICL) connected to output channels 5 and 7, and Patch B (MICR) connected to channels 6 and 8. Patch A is enabled; Patch B is disabled in this example.

Input Control (MS-II and rPod8.4)

Syntax: `Ain = MIC/LIN/all` (MS-II)

Syntax: `Ain = ADAT` (rPod8.4)

Syntax: `Ain = ANALOG` (rPod8.4)

Syntax: `Ain on/off`

Parameters:

`MIC` — Select mono microphone input.

`LIN` — Select stereo line level input.

`all` — Select line level and microphone (default).

`ADAT` — Select RASR board ADAT optical interface (rPod8.4).

`ANALOG` — Select onboard A/D analog inputs (rPod8.4).

Selects and enables/disables the audio input source. Analog inputs are disconnected at power-up.

Example:

```
Ain on       ; Connect input
Ain off      ; Disconnect input
Ain = MIC    ; Select microphone input only
```

Boost Microphone Gain

Syntax: `Boost "input ch" "gain"`

Parameters:

`"input ch"` — rPod: MICL, MICR, or all. MS-II: INL, INR, or all.

`"gain"` — rPod: 0, 20, 40, or 60 dB (rPod8.4 r1.2+: 0–60 dB in 3 dB steps). MS-II: 0, 3, 6, 9, or 12 dB.

Sets the microphone input gain for the specified input channel.

Example:

```
Boost MICL 40db     ; rPod10.2, rPod8.4 — 40 dB gain
Boost MICL 0db      ; MS-II — 0 dB additional gain
```

Mute / Un-Mute

Syntax: `Mute`

Syntax: `Unmute`

Mute silences all channels. Unmute restores all channels.

Example:

```
Mute
Unmute
```

Track Attack Time

Syntax: `Atk "track # list" = "time"      ; Playback volume attack`

Syntax: `Datk "track # list" = "time"     ; Ducking volume attack`

Parameters:

`"track # list"` — Track number list.

`"time"` — Time in frames, ms, or 1/10 second depending on TIMEREF setting.

Sets the attack duration for volume increases and un-ducking. Default is 0.

Example:

```
Atk 1,2 = 3000    ; 3 seconds in ms mode
Define ch1 = 1
Define ch2 = 2
Define ramp1 = 30  ; 1 second in frames mode
Atk ch1, ch2 = ramp1
```

Track Decay Time

Syntax: `Dek "track # list" = "time"     ; Playback volume decay`

Syntax: `Ddek "track # list" = "time"    ; Ducking volume decay`

Parameters:

`"track # list"` — Track number list.

`"time"` — Time in frames, ms, or 1/10 second (0–127, max 12.7 sec).

Sets the decay duration for volume reductions and ducking. Default is 0.

Example:

```
Dek 5 = 10    ; 1 second in 30fps frames mode
```

Amplifier Control (MS-II Only)

Syntax: `Amp on/off/stby/mute`

Parameters:

`on` — Turn amplifier on.

`off` — Turn amplifier off.

`stby` — Place amplifier in standby mode.

`mute` — Mute the amplifier.

Controls the speaker amplifier on the MS-II. To avoid a power-on pop, place the amplifier in standby before turning it on.

Example:

```
Amp off
Amp stby
Amp on
```

Clip-Based Metaphors

Clip metaphors simplify managing audio playback parameters by combining multiple commands into a single object. A clip references a sound file and specifies playback behavior, channel and track assignments, and volume settings. Multiple clips can reference a single audio file, each with different properties.

Load Clip

Syntax: `Ldclip "file name"`

Parameters:

`"file name"` — Clip file on the CF card. Drive prefix and .clip extension are optional.

Loads the clip into memory without executing the specified playback options. The clip remains in memory after execution and can be unloaded using Freeclip.

After starting the clip with a Start command, the assigned track number can be retrieved using Tvar:

Example:

```
Define tr
Define pc
Ldclip showsnd
Start C showsnd
tr = showsnd.t1.track    ; Get the assigned track number
pc = Tvar tr percent     ; Get the percent complete for track showsnd.t1
Stop tr                  ; Stop playback of track showsnd.t1
Stop showsnd.t2.track    ; Stop playback of track showsnd.t2
```

Execute Clip

Syntax: `Clip "file name"`

Parameters:

`"file name"` — Clip file on the CF card. Drive prefix and .clip extension are optional.

Loads the clip and immediately executes the specified playback options. The clip remains in memory and can be restarted with Start or unloaded with Freeclip.

Example:

```
Clip entrysnd
```

Unload Clip

Syntax: `Freeclip "file name"`

Parameters:

`"file name"` — Clip file name. Must have been previously loaded.

Removes the specified clip from memory. Example showing a multi-clip show:

Example:

```
Clip Intro              ; Play the Intro clip
Ldclip scene1           ; Pre-load three show clips
Ldclip scene2
Ldclip scene3
L1
    if Close1 == 1
        Start C scene1  ; Start the 1st pre-loaded clip
    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      ; Un-load all clips
    Freeclip scene1
    Freeclip scene2
    Freeclip scene3
```

The Clip database uses Microsoft Access (.accdb) format. A blank database is provided with the Clip Manager program. Do not modify database field names or delete any fields, as doing so could compromise Clip operations. The database may be duplicated or renamed as needed.

Clip Manager Program

The Clip Manager application provides a user interface for managing the Clip Database. It supports adding, editing, and deleting records, and exports individual clips as .clip files for use by Mini-Sam or rPod units. The application does not create new databases; it operates on existing .accdb files.

The figure below shows the application at startup, before any database connection has been established.

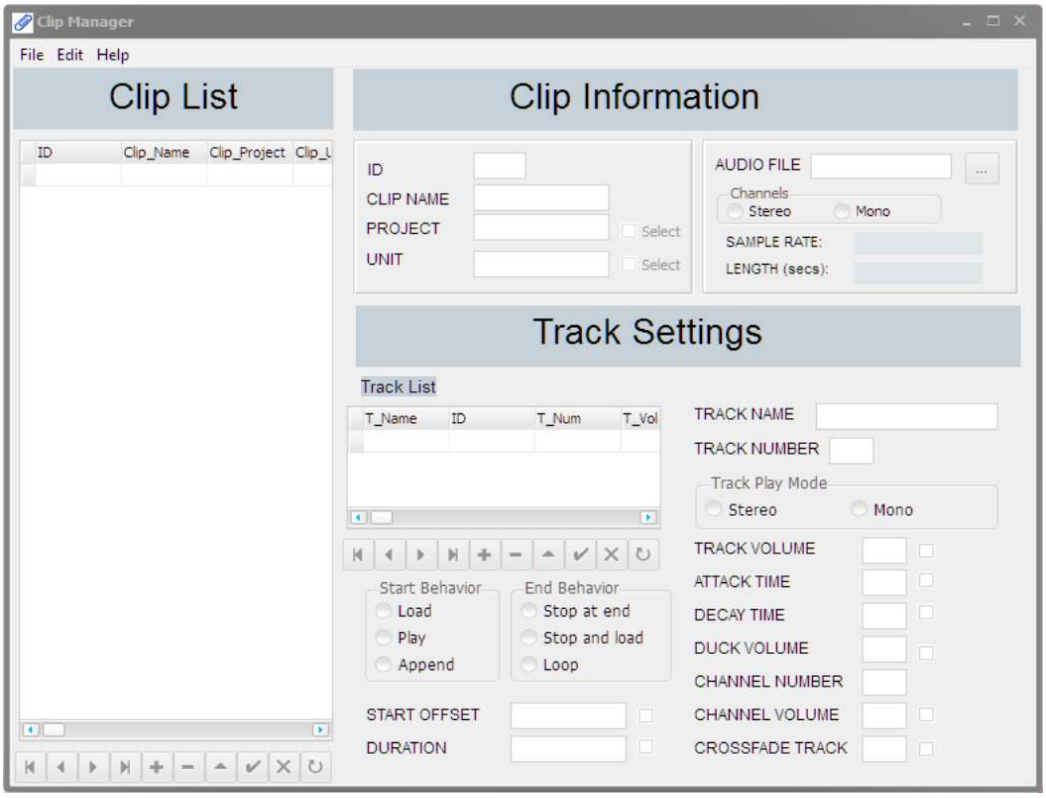

Figure 2 — Clip Manager at startup. No database is connected; the Clip List is empty and all Clip Information and Track Settings fields are blank.

Connecting to a Clip Database

To connect to a Clip database, select File → Open Connection from the menu bar.

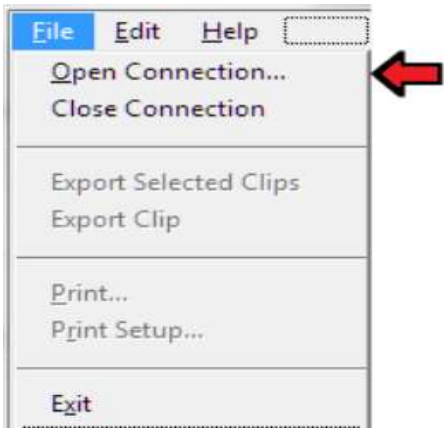

Figure 3 — File menu. Select Open Connection to establish a database link.

The Open dialog will appear. Navigate to the desired .accdb database file and click Open.

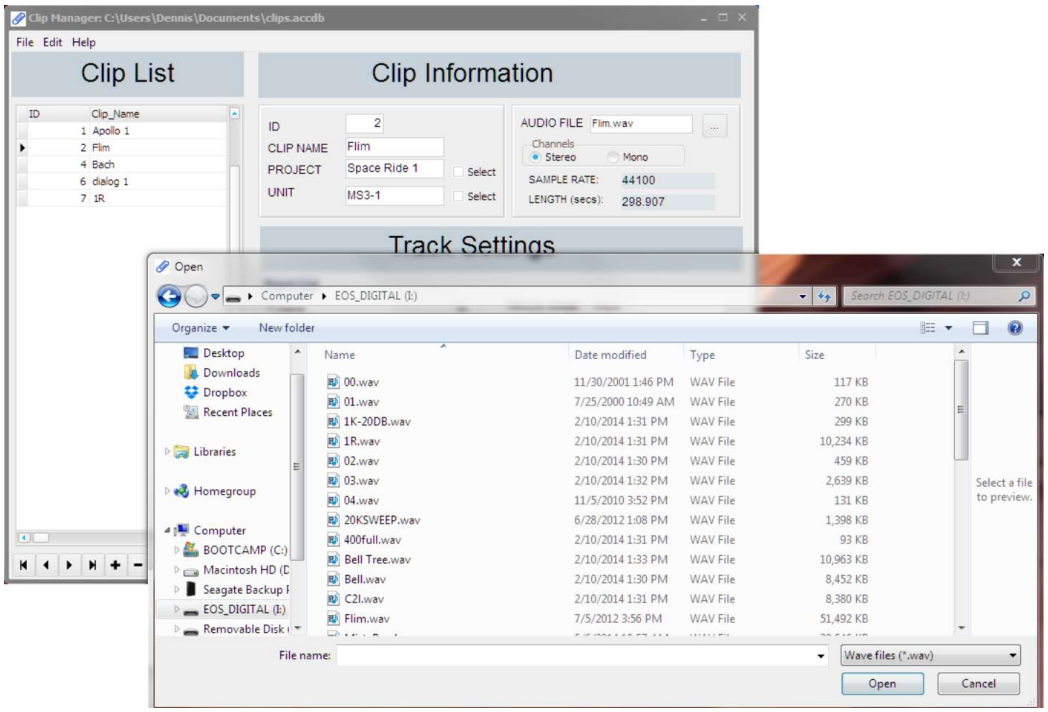

Figure 4 — Open file dialog for database selection. Navigate to a .accdb file and click Open. The Clip Manager title bar will update to reflect the connected database path.

Once connected, the Clip List is populated and the selected clip's information is displayed in the Clip Information and Track Settings panels.

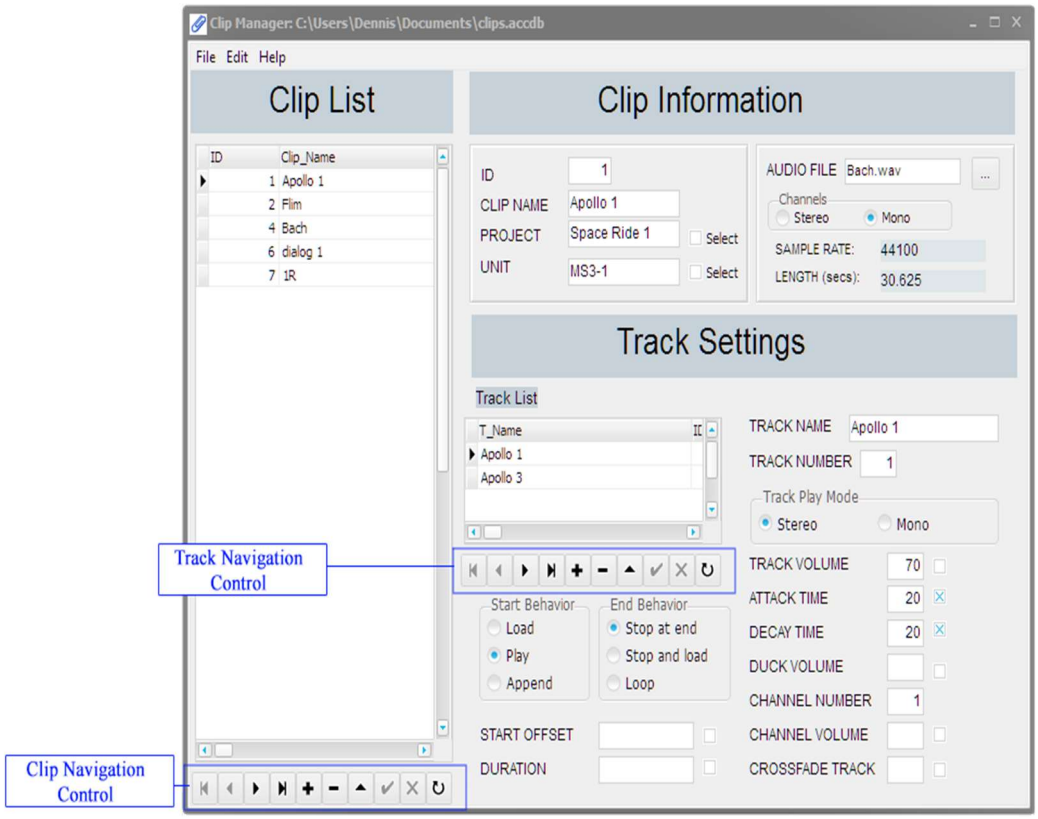

Figure 5 — Clip Manager with an open and connected database. The Clip List (left panel) displays all available clips. The Clip Information panel (upper right) shows the fields for the selected clip record. The Track Settings panel (lower right) shows the associated track parameters. Navigation controls appear at the bottom of both the Clip List and Track List panels.

Navigating the Clip Database

Two independent navigation control bars are provided — one for the Clip List and one for the Track List. Both bars offer identical functions applied to their respective record sets. The figure below identifies each button in the navigation control bar.

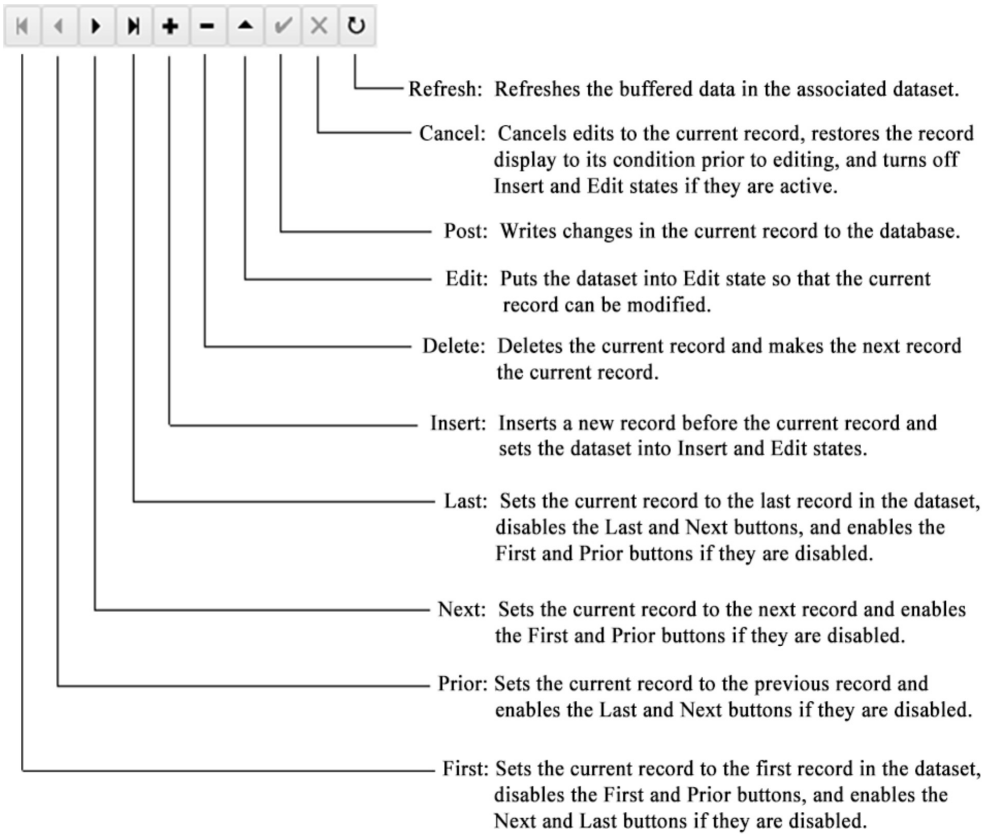

Figure 6 — Navigation control button reference. From left to right: First, Prior, Next, Last, Insert, Delete, Edit, Post, Cancel, and Refresh.

Clip Database Fields

The Clip Information panel contains the fields that define the clip record. The auto-generated ID links the clip record to its associated track records. The annotated diagram below identifies each field and its purpose.

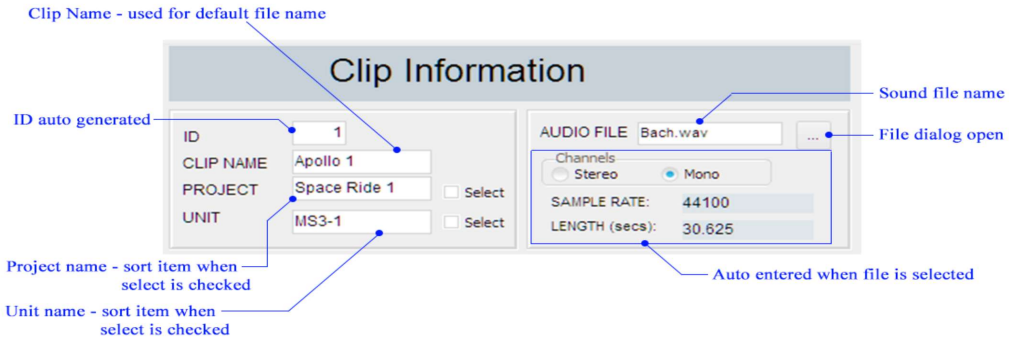

Figure 7 — Clip Information panel field reference. The ID field is auto-generated by the database and serves as the relational key to the Track List. The CLIP NAME is used as the default export file name. PROJECT and UNIT are optional filter fields. The AUDIO FILE, Channels, SAMPLE RATE, and LENGTH fields are populated automatically when a sound file is selected via the file dialog open button.

Track Database Fields

Each clip may have one or more associated track records shown in the Track List. The fields below define the playback parameters for each track. Fields with a check box are applied to playback only when the check box is enabled.

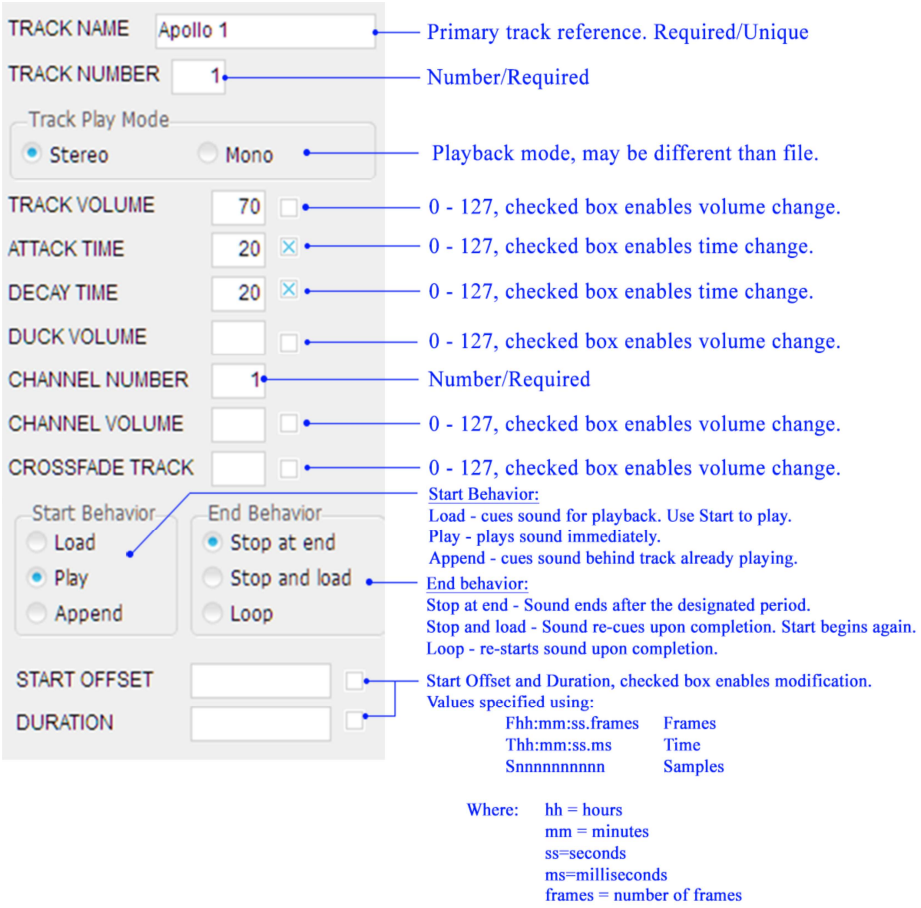

Figure 8 — Track Settings field reference. TRACK NAME and TRACK NUMBER are required fields. TRACK PLAY MODE may differ from the file format. Volume, timing, and channel fields are applied on execution only when their associated check box is selected. START BEHAVIOR controls how the clip initiates (Load, Play, or Append). END BEHAVIOR controls what occurs at the end of playback (Stop at end, Stop and load, or Loop). START OFFSET and DURATION accept time values in frames (Fhh:mm:ss.frames), milliseconds (Thh:mm:ss.ms), or samples (Snnnnnnnnnn).

Exporting Clips

A clip must be exported to a .clip file before it can be used on an rPod or Mini-Sam unit. Select the desired clip from the Clip List, then select File → Export Clip from the menu.

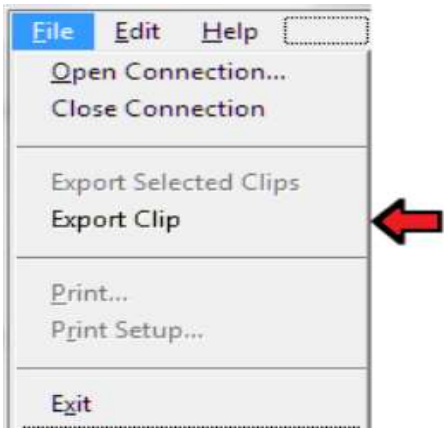

Figure 9 — File menu showing the Export Clip option. A Save dialog will appear with the clip name pre-populated as the default file name. Select a destination path, confirm or modify the file name, and press Enter to save. Once exported, transfer the .clip file to the CF media for use by the unit.

DMX/RS485 and Output Control Commands

These commands provide bit-level, byte-level, and dimmer control for DMX and RS485 output channels.

- Channels are numbered 1 to maxdmx.
- Bits are numbered 0 to 7.
- Control outputs are numbered 1 to 4096.
- DMX output values range from 0 to 255.
- Time units are milliseconds or frames depending on the TIMEREF setting.

Bit Level Control — Output

Syntax: `Output "control # list" on/off/toggle`

Parameters:

`"control # list"` — Output control number list (1–4096). On Mini-Sam and rPod, the first 16 outputs map to output connectors corresponding to DMX channels 1 and 2.

Sets, clears, or toggles the specified control outputs. Toggle inverts the current state.

Example:

```
Output 1,3 on      ; Turn on control outputs 1 and 3
Output 4,8,9 off   ; Turn off control outputs 4, 8, and 9
```

Bit Level Control — DMX Bit

Syntax: `Dmx "ch #" - "bit #" on/off`

Parameters:

`"ch #"` — DMX output channel (1 to maxdmx).

`"bit #"` — DMX output bit (1–8; 1 = LSb, 8 = MSb).

Sets or clears a specific bit within a DMX channel.

Example:

```
Dmx 5-4 on      ; Turn on bit 4 of DMX channel 5
Dmx 1-8 off     ; Turn off bit 8 (MSb) of DMX channel 1
```

Byte Level Control

Syntax: `Dmx "ch #" = "output value"`

Parameters:

`"ch #"` — DMX output channel (1 to maxdmx).

`"output value"` — Byte value 0–255 (0 = all bits off, 255 = all bits on).

Sets all 8 bits of the specified DMX channel to the given value.

Example:

```
Dmx 22 = 255    ; All bits on
Dmx 14 = 3      ; Bits 0 and 1 on only
```

Dimming Control

Syntax: `Dimto "output value" in "time (ms)" on "ch # list"`

Parameters:

`"output value"` — Target DMX byte value (0–255).

`"time"` — Dimming duration in ms, frames, or 1/10 sec depending on TIMEREF.

`"ch # list"` — DMX output channels (1 to maxdmx).

Smoothly transitions DMX channel output values to the target over the specified time.

Example:

```
Dimto 120 in 1000 on 1,2    ; Dim channels 1 and 2 to 120 in 1 second
```

DMX/RS485 Output Driver Control

Syntax: `Dmx DRIVE`

Syntax: `Dmx RELEASE`

Enables and disables the DMX/RS485 differential driver. RELEASE places the driver output in a high-impedance state. Used for half-duplex RS485 or non-standard bidirectional DMX control.

Example:

```
Dmx DRIVE     ; Enable driver
Dmx RELEASE   ; Disable driver
```

DMX File Playback

DMX playback files are generated using the Programmer or DMXedit software applications, or recorded using the Recdmx command. Before playback begins, the system checks whether the file is already mounted on a track. If so, the loading stage is skipped. Leaving tracks mounted after playback reduces CF card bandwidth requirements.

Pre-load DMX File

Syntax: `Lddmx "file name" on "track #"`

Parameters:

`"file name"` — DMX file on the CF card. Drive prefix and extension are optional.

`"track #"` — DMX track number.

Pre-loads a DMX file to the specified track without starting playback. Used with the Start command to synchronize DMX and audio playback.

Example:

```
Lddmx light1DMX on 1    ; Pre-load to track 1
Lddmx light2DMX on 3    ; Pre-load to track 3
Lddmx light3DMX         ; Pre-load to auto-assigned track
Start                   ; Begin synchronized playback
```

Unload DMX Track

Syntax: `Unloaddmx "track # list"`

Syntax: `Unloaddmx all`

Parameters:

`"track # list"` — DMX tracks to unload.

Releases pre-loaded DMX tracks from memory.

Example:

```
Unloaddmx 1    ; Remove pre-loaded track 1
```

Play DMX File

Syntax: `Playdmx "file name" on "track #"`

Parameters:

`"file name"` — DMX file on the CF card.

`"track #"` — DMX track number.

Mounts and begins playback of the DMX file on the specified track. The file remains mounted after playback.

The file can only be unmounted via the Unmount command, redefinition, reboot, or hardware reset.

To play a temporary file (auto-unmounted after playback), omit the track number.

Example:

```
Playdmx light1DMX on 1    ; Permanent mount: file stays in memory after
playback
Start C light1DMX         ; Restart the already-mounted file
Playdmx light7DMX         ; Temporary mount: file removed after playback
```

Loop DMX File

Syntax: `Loopdmx "file name" on "track #"`

Parameters:

`"file name"` — DMX file on the CF card.

`"track #"` — DMX track number.

Mounts and continuously loops the DMX file. Stop with Stopdmx.

Example:

```
Loopdmx light1DMX on 1
```

Mount DMX File

Syntax: `Mount "file name" on "track #"`

Parameters:

`"file name"` — DMX file on the CF card.

`"track #"` — DMX track number.

Mounts a DMX file to the specified track without starting playback. The file remains in memory until unmounted or redefined.

Note: this command does not cause playback of the file.

Example:

```
Mount light1DMX on 5          ; Mount without playing
Playdmx light1DMX on 2        ; Now play the mounted file on channel 2
```

The second Playdmx command above does not require CF card access because the file was already mounted. When playback completes, the file remains mounted on track 5.

Un-mount DMX Files

Syntax: `Unmount "track # list"`

Parameters:

`"track # list"` — DMX track numbers to unmount.

Unmounts files from the specified DMX tracks. This command is not accepted when the specified tracks are busy.

Example:

```
Unmount 1,2,3,4
Unmount all
```

Stop DMX Tracks

Syntax: `Stopdmx "track # list"`

Syntax: `Stopdmx all`

Parameters:

`"track # list"` — List of actively playing DMX tracks.

Stops playback on the specified DMX tracks.

Example:

```
Stopdmx 1,2
```

```
Stopdmx all    ; Also used to stop all sounds and DMX
```

Position DMX Playback

Syntax: `Posdmx "track # list" to F/T "position reference"`

Syntax: `Skipdmx "track # list" to F/T "position reference"`

Parameters:

`"track # list"` — DMX track numbers.

`F / T` — F = frames (1/30 sec), T = time in ms (default).

`"position reference"` — Position value. Skip may be positive or negative.

Posdmx positions from the beginning; Skipdmx positions relative to the current playback position.

Example:

```
Posdmx 1,3 to F300    ; Position at frame 300 (10 seconds)
Posdmx 1,3 to T10     ; Position at 10 ms
Skipdmx 1 F-1890      ; Skip backward 1890 frames (63 seconds)
Skipdmc 1 T-1:03      ; Skip backward 1 minute 3 seconds
```

DMX File Capture

Record DMX

Syntax: `Recdmx "file name" from "ch # (start)" to "ch # (end)"`

Parameters:

`"file name"` — Output file name to save on the CF card.

`"ch # (start)"` — First DMX input channel to record.

`"ch # (end)"` — Last DMX input channel to record.

Records the specified range of DMX input channels to the named file.

Example:

```
Recdmx light10DMX from 1 to 10
```

Pre-load Record DMX

Syntax: `Ldrecdmx "file name" from "ch # (start)" to "ch # (end)" T "linked sound track #"`

Parameters:

`"file name"` — Output file name.

`"ch #"` — Input DMX channel range.

`T "linked sound track #"` — Optional: links recording to a sound track. Recording stops when the sound file ends.

Prepares the system for synchronous DMX recording with simultaneous audio/DMX playback. Recording begins when the Start command is issued.

Example:

```
Lddmx light1DMX on 11          ; Pre-load DMX file for playback on track 11
Ldsnd Flim on 1 T1             ; Pre-load audio file for playback
Ldrecdmx light10DMX from 1 to 10 T1  ; Pre-load record, linked to sound track
1
Start                          ; Begin synchronized playback and recording
```

> **NOTE:** *The T1 flag in the Ldrecdmx command links DMX recording to sound track 1. Recording will stop automatically when the sound file "Flim" completes playback.*

Start Pre-loaded Recording

Syntax: `Start`

Initiates synchronized playback and recording for all pre-loaded sound, DMX, and record DMX commands.

Example:

```
Start
```

Stop Recording DMX

Syntax: `Stop all`

Syntax: `Stopdmx all`

Either command stops all active DMX recording.

Example:

```
Stop all
Stopdmx all
```

Ethernet

Configure Ethernet

Syntax: `Config ENET <"ip address"> : <"port"> / <"subnet address">`

Parameters:

`"ip address"` — Standard 32-bit IP address in dot notation (e.g., 192.168.1.10). Optional.

`"port"` — UDP port destination filter. Optional.

`"subnet address"` — Standard 32-bit subnet mask in dot notation. Optional.

Sets the IP address, port, and subnet mask for the unit. Settings are stored in the .ini file on the CF card. Each parameter can be set independently or together.

Example:

```
Config ENET 192.168.1.5                    ; Change IP address only
Config ENET :1000                          ; Change port only
Config ENET 192.168.1.5:1000/255.255.0.0   ; Change IP, port, and subnet
```

Configuring the Gateway

Syntax: `Config GATEWAY "ip address"`

Parameters:

`"ip address"` — Standard 32-bit gateway IP address in dot notation (e.g., 192.168.1.1).

Sets the default gateway address for the unit. The gateway address is stored on the resident Compact Flash card in the .ini file. The setting can be applied by issuing the Config GATEWAY command or by editing the .ini file directly with a text editor.

Example:

```
Config GATEWAY 192.168.1.1
```

Ethernet Command Status Reporting

Syntax: `Verbose ENET on/off`

Parameters:

`on` — Enable command status feedback to the requesting device.

`off` — Disable command status feedback (default).

Enables or disables command feedback for Sendcmd/Sendcmds operations.

Example:

```
Verbose ENET on
Verbose ENET off
```

Configure FTP Password

Syntax: `Config PASSWORD "string"`

Parameters:

`"string"` — Password string constant.

Changes the FTP password. The default password is "SKE".

Example:

```
Config PASSWORD MyNewPass
```

Configure UDP

Syntax: `Config UDP Manual/Auto`

Parameters:

`Auto (default)` — All raw UDP data is forwarded to the Console.

`Manual` — Use with Read UDP or Peek UDP commands.

Sets the UDP data handling mode.

Example:

```
Config UDP Manual
```

IP Referencing (Bind)

Syntax: `Bind "ip address" to "bind number"`

Parameters:

`"ip address"` — IP address of the target network device in dot notation.

`"bind number"` — Arbitrary reference number (1–255). Use 255 for broadcast.

Assigns a short-hand numerical reference to an IP address for use with Send and Sendcmd.

Example:

```
Bind 169.254.144.60 to 1        ; Bind IP to reference 1
Bind 169.254.144.255 to 255     ; Network broadcast bind
```

UDP Control (Send Command)

Syntax: `Sendcmd "bind number": "constant string"`

Syntax: `Sendcmds "bind number": "variable string"`

Parameters:

`"bind number"` — Destination bind number (1–255).

`"constant string"` — A command string to be executed by the receiving device.

`"variable string"` — A local, global, or public string variable containing the command.

Sends any valid console command to another Mini-Sam or rPod unit on the network.

When using Sendcmds on the P1 side of an rPod8.4, use a Public string variable. P2 may use a public, global, or local string.

Example:

```
Sendcmd 1: "Play Flim on 1"
Sendcmd 1: "Lddmx lights on 3"
Sendcmd 1: "Start"
; To simulate via a UDP terminal (e.g., Hercules): send just the command
string without Sendcmd or bind number.
; Sendcmds variable string example:
Define $cmd
$cmd = "Play Flim on 1"
Sendcmds 1: $cmd
```

UDP Sending Data

Syntax: `Send "bind number".<"packet number">-<"message type">|"string or variable list"`

Syntax: `Printe "bind number"|"string or variable list"`

Parameters:

`"bind number"` — Destination bind number (1–255).

`"packet number"` — Optional 32-bit packet number assigned by the user (default 0).

`"message type"` — Optional 32-bit message type identifying data format (default 0).

`"string or variable list"` — Comma-delimited list of variables, strings, or constants.

Sends structured data to UDP-compatible devices. Printe is similar but omits packet number and message type, making it suitable for non-Mini-Sam/rPod devices. Use with Recv on the receiving device.

Example:

```
; message type 1 = ack(1)/nack(0), type 2 = variable packet
Send 25.103-2|x, $a, $b, z    ; Send to bind 25, packet 103, type 2
; Printe example:
Printe 1|"status", x           ; Send without packet/type headers
```

Receive UDP Data

Syntax: `Recv "string or variable list"`

Syntax: `Read UDP`

Syntax: `Peek UDP`

Syntax: `Config ENET Term "Term method"`

Parameters:

`"string or variable list"` — Variables to receive the incoming data.

`Term method` — String termination: NULL (default), CR, CRLF, TAB, or NONE.

Recv assigns received data to local or global variables. System variables updated after receipt: ?ETYPE, ?EORIG, ?ECOUNT, ?EPKT, ?ENET.

Read UDP updates ?ENET = 2 and replaces the UDP system variable with the new packet data, then discards the packet.

Peek UDP behaves like Read UDP but does not discard the packet.

Config ENET Term changes the string termination character for received data.

Example:

```
; Recv example — receive and acknowledge a variable packet:
if ?ENET == 1              ; Is there a UDP data packet?
    if ?ETYPE == 2         ; Is it a variable packet (type 2)?
        Recv x, $a, $b, z  ; Grab the data
        y = ?EORIG         ; y = source bind number
        Send y|1           ; Send back an acknowledgment
    endif
endif
Config ENET Term CR        ; Set string termination to carriage return
```

System Commands

> **NOTE:** *The system constants ?P and ?T return playback status when used in sequences. They are separate from the ?P and ?T commands described below.*

Re-Boot

Syntax: `Boot`

Causes the system to re-initialize. Equivalent to pressing the hardware reset button.

Example:

```
Boot
```

Set Time Base

Syntax: `Config TIMEREF ms/frames/mixed`

Parameters:

`ms` — Sets time measurement to milliseconds.

`frames` — Sets time measurement to frames (1/30 second).

`mixed` — Legacy mode using a mixture of time bases.

Sets the fundamental time reference for the system. Best set from the Console with a CF card installed, as the setting is saved to the .ini file and restored at power-up. The table below shows how each setting affects time-related operations:

Example:

```
Config TIMEREF ms
```

Operation	ms	frames	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`

Syntax: `Reload`

Syntax: `Remove "variable list"/all`

Parameters:

`Update` — Saves the current values of all cosmic variables to boot.ini on the CF card.

`Reload` — Reloads and restores cosmic variables from boot.ini.

`Remove "variable list"` — Converts specified cosmic variables to regular global variables. Requires a subsequent Update.

Cosmic variables are created with Define Cosmic <variable>. They are not automatically saved until Update is issued.

At power-up or boot, the system calls Reload to restore cosmic variables from boot.ini. Issuing Reload manually can overwrite current values.

Example:

```
Update       ; Save cosmic vars to CF card
Reload       ; Restore cosmic vars from CF card
Remove x, y  ; Convert x and y to global vars
Update       ; Apply the removal
```

Start Playback

Syntax: `Start <T "track # list" D "channel # list" C "clip name list">`

Parameters:

`T "track # list"` — Optional: start only the specified pre-loaded sound tracks.

`D "channel # list"` — Optional: start only the specified pre-loaded DMX channels.

`C "clip name list"` — Required to start clips. Case-sensitive. Uses clip names, not file names.

Begins synchronized playback of all pre-loaded sound, DMX, and clip files.

Example:

```
Start                          ; Start all pre-loaded sounds and DMX
Start T 1,3,5                  ; Start only tracks 1, 3, and 5
Ldsnd sound2 on 1 T2 X1
Start T 1,2                    ; Include track 1 (fading out) in the start list
for cross-fade
Start T 1,3 D 1,2              ; Start specific tracks and DMX channels
Start C Apollo                 ; Start the clip named "Apollo"
```

Query Playing Tracks

Syntax: `?P`

Prints the current playback status. Returns "0 tracks playing" if idle, or the total count followed by an itemized list.

Example:

```
?P
; If no tracks are playing:
0 tracks playing:
; If tracks are playing:
2 tracks playing: 1, 2
```

Query Track Status

Syntax: `?T "track # list"`

Parameters:

`"track # list"` — Track numbers to query.

Prints the status of the specified tracks.

Example:

```
?T 1,2
; If no tracks are active:
T1: UNUSED
T2: UNUSED
; If tracks are playing:
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
```

Print

Syntax: `Print(port) "variable list"`

Syntax: `Print(port) # "variable list"`

Parameters:

`(port)` — Optional: c=Console, d=DTE, e=Ethernet, x=DMX (serial), m=MIDI (serial), n=Network.

`"variable list"` — Numbers, variables, and strings. Placing # before a variable prints it as an ASCII control code.

Displays variables and strings to a serial port. The standard Print command adds a CR/LF before and after the output plus the drive prompt. Print # outputs only the specified values without CR/LF or prompt.

Example:

```
Context = 1
Define x
x = 5
x = x + 3
Print "x = ", x
x = 8               ; Displayed result
; Hexadecimal example:
x = 0x14
Print "0x14 is ", x
0x14 is 20          ; Displayed result
; Print # with control characters:
Define LF = 10
Define CR = 13
Print # #CR, #LF, "Hello", 10, 13, "a:>"
```

```
; Print to specific ports:
Printd "Hello"     ; Print to DTE port
Printc "Hello"     ; Print to Console port
```

Monitor Serial Port

Syntax: `Monitor Console/DMX/DTE/Midi on/off  (rPod8.4 r1.2)`

Syntax: `Monitor Console/DMX/DTE on/off  (rPod8.4 r1.0a)`

Syntax: `Monitor Console/DMX on/off  (MS2e)`

Syntax: `Monitor all off`

Enables or disables monitoring of the specified serial port transmit and receive signals via the Status LEDs.

MS2e: Tx on Status 1, Rx on Status 2.

rPod: Tx Console/DMX on Status 1, Rx Console/DMX on Status 2, Tx DTE/Midi on Status 3, Rx DTE/Midi on Status 4.

It is possible to monitor the Console and DTE ports simultaneously.

Example:

```
Config DMX RS485    ; Set the DMX port to RS485 mode
Monitor DMX on      ; Monitor DMX transfers
Monitor Console on  ; Restore Console monitoring
```

Configure Serial Port

Syntax: `Config Console/DTE/DMX = "baud rate", "#bits", "#stop bits", "parity"`

Syntax: `Config Midi = "baud rate", "#bits", "#stop bits", "parity"  (rPod8.4 r1.2)`

Syntax: `Config Console/DTE Term = "character list"`

Syntax: `Config Console/DTE Term on/off`

Syntax: `Config DMX Standard/RS485`

Syntax: `Config DMX Receive on/off`

Parameters:

`"baud rate"` — Communication speed.

`"#bits"` — Data bits.

`"#stop bits"` — Stop bits.

`"parity"` — Parity setting.

Modifies the operating characteristics of the RS-232 ports. Default: 115,200 baud, 8 bits, 1 stop bit, no parity.

The DMX port on the MS2e and rPod8.4 r1.2 can be reconfigured as an RS485 port. When in RS485 mode, the DMX receiver should be disabled.

Example:

```
Config DMX Receive on         ; Enable DMX receiver
Config DMX Receive off        ; Disable DMX receiver
```

```
Config Console = 9600          ; Set Console to 9600 baud
Config DTE = 115200, 7, 2, 0   ; 7 bits, 2 stop bits
Config Console Term = 10, 13   ; Termination: linefeed + carriage return
Config Console Term off        ; Disable termination
Config Console Term = 13       ; Set single-char termination
Config Console Term on         ; Re-enable termination
```

Null Character Substitution

Syntax: `Config Console/DTE nullsub on/off`

Parameters:

`on` — Convert incoming null characters (0x00) to spaces (0x20).

`off` — Pass null characters without conversion (default).

Prevents premature string termination when receiving binary data that contains embedded null characters.

Example:

```
Config Console nullsub on
```

Read Serial Port

Syntax: `Read Console/DTE #/& "count"`

Syntax: `Read Console/DTE Lock`

Syntax: `Read Console/DTE Release`

Parameters:

`#` — Optional: convert data to decimal string representation.

`&` — Optional: convert data to hexadecimal string representation.

`"count"` — Number of characters to read (term off mode only).

`Lock` — Lock the Console in Read mode; disable command line processing.

`Release` — Release the Console from Lock mode.

Captures an input string from the specified serial port. When string termination is on, data is transferred only after the termination character is detected. When termination is off, exactly count characters are read immediately from the buffer.

Example:

```
Read Console           ; Read with termination on (waits for terminator)
Print Console
Hello                  ; Displayed result if "Hello" was typed
; Reading with termination off:
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"`

Parameters:

`#` — Optional decimal conversion.

`&` — Optional hexadecimal conversion.

`"count"` — Number of characters to read.

Identical to Read Serial Port but does not remove the data from the receive buffer.

Example:

```
Peek Console 4
```

Silent Mode

Syntax: `Silent <Console/DMX/DTE/MIDI/VMSKE/all> on/off`

Parameters:

`on` — Suppress all status output to the specified ports.

`off` — Enable status output (default).

Enables or disables all status data printed to the specified ports. Affects prompts, error messages, and splash screens, but not explicitly directed Print commands. The silent mode setting is saved to the .ini file and restored at power-up.

Example:

```
Silent DTE on     ; Suppress all status on DTE port
```

Command Feedback (Verbose)

Syntax: `Verbose on/off`

Parameters:

`on` — Enable command feedback from the active port.

`off` — Disable command feedback (default at power-up).

Enables or disables command result feedback from the controller to the terminal. Useful for custom serial control or adapting to an existing serial protocol.

Example:

```
Verbose on
```

Install Operating System

Syntax: `Install "file name"`

Parameters:

`"file name"` — Operating system file on the CF card. Drive prefix and .ldr extension are optional.

Installs a new operating system from the CF card. Once the command starts, removing power will result in an incomplete installation; the BootLoader method must then be used to restore operation. Refer to the Hardware Manual for detailed procedures.

Example:

```
Install rpodP1a056
```

Controlling Status LEDs

Syntax: `Status "output list" on/off`

Parameters:

`"output list"` — One or more of: 1, 2, 3, 4 (Status LEDs 1–4), P (Processor OK), A (Audio OK), all.

Turns the specified status indicator LEDs on or off.

Example:

```
Status all on
Status 1,2 off
Status P on
```

Configure Triggers

Syntax: `Config TRIG "trigger list" = "de-bounce time"`

Parameters:

`"trigger list"` — List of trigger inputs to configure.

`"de-bounce time"` — De-bounce time in milliseconds.

Sets the de-bounce time for the specified triggers. Default is 20 ms. Decreasing it improves response but increases sensitivity; increasing it reduces sensitivity.

Example:

```
Config TRIG 1,2 = 2    ; Set 2ms de-bounce on triggers 1 and 2
```

Logging

Logging commands automatically record device activity, commands, responses, and I/O to the CF card. Log files are named LOGxxxxxx.txt where the suffix reflects the date based on the log mode. Issue an Update command periodically to flush the write buffer and prevent data loss on power failure.

Log Activation / Deactivation

Syntax: `Log on`

Syntax: `Log off`

Log on activates system logging. Configure Log replace and Log mode before issuing Log on. Log timestamp can be changed at any time.

Log off deactivates logging. A Log update is automatically issued as part of the command.

Example:

```
Log on
```

```
Log off
```

Log Update

Syntax: `Log update`

Forces the flushing of the log write buffer. Issue periodically in a sequence program to ensure data integrity on unexpected power loss.

Example:

```
Log update
```

Log File Clearing

Syntax: `Log clear`

Clears the current log file. Logging can be enabled or disabled. The log mode must be the same as when the original file was created.

Example:

```
Log clear
```

Log Listing

Syntax: `Log list`

Lists the current log file to the command input device. Logging is temporarily paused during the listing.

Example:

```
Log list
```

Log Mode

Syntax: `Log freq none/hour/day/month`

Parameters:

`none` — Create only one file; no additional files beyond the initial.

`hour/day/month` — Create a new file at each interval transition.

Sets the file creation rate. If replace mode is set, the previous file is erased before a new one is created.

Example:

```
Log freq hour
```

Log File Handling

Syntax: `Log replace true/false`

Parameters:

`true` — Erase the existing file before creating a new one.

`false` — Retain existing files; add new files at each interval (default).

Defines how the system handles log files at interval transitions.

Example:

```
Log replace true
```

Log Timestamp

Syntax: `Log timestamp on/off`

Parameters:

`on` — Prefix each log entry with a timestamp.

`off` — No timestamp prefix (default).

Enables or disables timestamp prefixing for log entries. Can be changed at any time while logging is active.

Example:

```
Log timestamp on
```

Sequence Programming Commands

Sequences provide programmable logic control for full automation and advanced control. Programs are text files with a .seq extension. The file startup.seq is reserved for auto-execution at power-up. Up to 16 sequences of 2048 lines each can run simultaneously.

Comments begin with a semicolon (;). Jump labels mark program positions for Goto instructions and debug commands. Example:

```
; This is the program body
main    do    ; Start of main loop
```

Setting Sequence Context

Syntax: `Context "sequence # (1-16)"`

Parameters:

`"sequence #"` — Sequence number from 1 to 16.

Defines the active sequence, providing visibility to its local variables from the Console or DTE port. The following example shows two separate instances of variable x, one per sequence:

Example:

```
Context  1
SEQ 1 Loaded      ; Status from the unit
Define x
x = 5
Context 2
SEQ 2 Empty       ; Status from the unit
Define x
x = 10
Print x
x=10              ; Displayed value
Context 1
SEQ 1 Loaded      ; Status from the unit
```

```
Print x
x=5                ; Displayed value
```

Local Variable Definition

Syntax: `Define <$>"variable"`

Parameters:

`"variable"` — Variable name. Prefix with $ for string variables.

Creates a local variable in the active sequence. Local variables are unique to the sequence and not shared between sequences.

Example:

```
Define pause32         ; Creates an integer variable
Define $mystring       ; Creates a string variable
```

Global / Public Variable Definition

Syntax: `Define Global <$>"variable"`

Syntax: `Define Public <$>"variable"` (rPod8.4 only)

Parameters:

`"variable"` — Variable name.

Creates a global variable accessible to all sequences. Public variables (rPod8.4 only) are also shared between both processors.

Example:

```
Define Global main_count
Define Global $astring
main_count = 50
$astring = "Hello"
Print main_count
(G)main_count=50       ; Displayed value (G = global)
Print $astring
Hello                  ; Displayed value
```

Group Enumeration

Syntax: `Formgroup "file name" as "group #"`

Parameters:

`"file name"` — Sequence file on the CF card.

`"group #"` — Group reference number (1–500).

Assigns a numeric reference to a group file. If the file show1.seq contains these three commands:

Example:

```
Play Flim on 1         ; Content of show1.seq
Playdmx lights on 3
Dmx 1-1 on
```

```
; Enumerate show1 as group 1:
Formgroup show1 as 1
; Then execute it:
Playgroup 1          ; or
Playgroup x          ; where x = 1
```

Group Playback

Syntax: `Playgroup "file name"`

Syntax: `Playgroup "group #"`

Syntax: `Playgroup "variable"`

Parameters:

`"file name"` — Sequence file to play directly.

`"group #"` — Group number (1–500).

`"variable"` — Variable containing the group number.

Plays a group file. Group files contain direct executable code only — branching, looping, timer, and delay commands are invalid.

Example:

```
Playgroup show1.seq   ; Play by filename
Playgroup 1           ; Play by group number
Playgroup x           ; Play using a variable (where x is previously defined)
```

Load Sequence File

Syntax: `Ldseq "file name" {on "sequence # (1-16)"}`

Parameters:

`"file name"` — Sequence file on the CF card.

`"sequence #"` — Optional sequence slot (1–16). If omitted, the first free slot is used.

Loads, compiles, and places the sequence into resident memory. The file startup.seq is automatically loaded and executed at power-up using slot 1.

Example:

```
Ldseq Show1.seq       ; Load to first available slot
Ldseq Show1 on 1      ; Load to slot 1
Ldseq Show1 on x      ; Load using a variable for the slot number (where x =
1)
```

Play Sequence Program

Syntax: `Playseq "sequence # (1-16)"`

Syntax: `Playseq "file name" {on "sequence # (1-16)"}`

Syntax: `Startseq "sequence # list"`

Syntax: `Startseq "file name list"`

Syntax: `Startseq all`

Syntax: `Go`

Parameters:

`"sequence #"` — Sequence slot number (1–16).

`"file name"` — Sequence file name.

Starts execution of a sequence program. Playseq always restarts from the beginning. Startseq, Startseq all, and Go resume paused sequences from the last-executed instruction. Go resumes only the sequence referenced by the current context.

Example:

```
Playseq 2              ; Start sequence 2
Playseq x              ; Start using a variable (where x = 2)
Playseq show1 on 2     ; Load and start show1 in slot 2
Startseq 1,2           ; Resume paused sequences 1 and 2
Startseq all
Go
```

Stop Running Sequences

Syntax: `Halt "sequence # list (1-16)"`

Syntax: `Halt "file name list"`

Syntax: `Halt all`

Parameters:

`"sequence # list"` — Sequence slot numbers (1–16).

`"file name list"` — Sequence file names.

Stops execution of the specified sequence programs. Use Go or Startseq to resume; use Playseq to restart from the beginning.

Note: Halt does not stop any audio or control files that are currently playing.

Example:

```
Halt 1,2    ; Stop sequences 1 and 2
Halt all    ; Stop all sequences
```

Step Through a Sequence

Syntax: `Step "# of steps"`

Parameters:

`"# of steps"` — Optional number of program lines to advance. Default is 1. Blank and comment lines count as steps.

Advances a stopped sequence by the specified number of steps. The sequence must be active and stopped. The sequence to step is set by the Context command.

Example:

```
Step         ; Advance 1 step
Step 1000    ; Advance 1000 steps
```

Set Program Breakpoints

Syntax: `Break "line #/label/variable list"`

Syntax: `Break "line #/label/variable list" off`

Syntax: `Break all off`

Parameters:

`"line #/label/variable"` — Program line number, jump label, or variable containing the line number.

Sets a stop position in a sequence program. The sequence is defined by the Context command.

Example:

```
Break Loopstart, programend  ; Set breakpoints by label
Break 15, 20, 37             ; Set breakpoints by line number
Break Loopstart off          ; Remove a single breakpoint
Break all off                ; Remove all breakpoints
```

List a Sequence

Syntax: `List "Starting line #/label" {to "end line #/label" / :"line count"}`

Parameters:

`"line #/label"` — Optional start and end of the listing range.

`:"line count"` — Number of lines to print (overrides end line if specified).

Displays the sequence to the Console or DTE ports. The sequence is defined by the Context command. A > character marks the current position when stopped. An * character marks breakpoints.

Example:

```
List                         ; List the entire sequence
List Mainloop to 27          ; List from label to line 27
List Mainloop :10            ; List 10 lines from label
List branch1 to branch5      ; List between two labels
```

Program Flow

Program flow commands are available only within sequence programs and are not valid for direct entry from the Console or DTE ports.

Timer Delay

Syntax: `Wait "time"`

Syntax: `Wait "min time" - "max time"`

Parameters:

`"time"` — Delay duration in ms, frames, or 1/10 sec depending on TIMEREF.

`"min time" - "max time"` — Random delay within the specified range.

Pauses execution of the current sequence for the specified duration.

Example:

```
Wait 1000              ; Wait 1 second
```

```
Wait t1                 ; Wait using a variable (where t1 = 1000)
Wait 1000 - 5000        ; Wait between 1 and 5 seconds randomly
```

Branch Immediate (Goto)

Syntax: `Goto "line label"`

Parameters:

`"line label"` — Destination jump label. May not be a reserved word.

Redirects program flow to the specified jump label. The following shows an alternate method for looping a sound file (the more efficient Loopsnd command would normally be used instead):

Example:

```
Define status
Main    Play Flim on 1 T1
        do
            status = ?P1
        while status <> 0
        Goto Main
```

Branch Conditional (if/else/endif)

Syntax: `if "operation"`

Syntax: `...`

Syntax: `else`

Syntax: `...`

Syntax: `endif`

Parameters:

`"operation"` — See the Table of Operations for valid forms.

Executes the block after if when the operation is true. Conditional constructs can be nested to multiple levels.

Example:

```
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 (select/case/endsel)

Syntax: `select "variable"`

Syntax: `case n`

Syntax: `...`

Syntax: `break`

Syntax: `endsel`

Parameters:

`"variable"` — The variable to compare.

`case n` — A constant value to compare against the variable.

The following example monitors trigger 1 and cycles through playback of three separate messages on each trigger press:

Example:

```
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 (do/while)

Syntax: `do`

Syntax: `...`

Syntax: `while "operation"`

Parameters:

`"operation"` — Condition evaluated at the end of each iteration.

Executes the block at least once, then repeats while the operation is true.

Example:

```
Define Count
Count = 0
Play Flim on 1 T1
do
    Count = Count + 1
while ?P1 <> 0    ; Loop until track 1 finishes playing
```

For Loop (for/next)

Syntax: `for "variable" = "operand1" to "operand2"`

Syntax: `...`

Syntax: `next`

Parameters:

`"variable"` — The loop incrementing variable (local or global).

`"operand1"` — Initial value.

`"operand2"` — Terminal value. Loop ends when variable equals operand2.

Increments the variable on each pass and executes the block while the variable has not yet reached operand2.

Example:

```
Define Count
for Count = 1 to 100
    Play Flim on 1 T1
    do
    while ?P1 <> 0    ; Wait for each play to finish
next
```

Event Programming

Event programming provides timer-based show control using the system timer. See the Timer-Based Actions section for conceptual background.

Event List Header and Footer

Syntax: `Eventlist "name"`

Syntax: `Endlist`

Parameters:

`"name"` — Constant string event list name. Up to 64 uniquely named event lists per system.

Defines the start and end of a timer event list. Timer cues are placed between these markers.

Example:

```
Eventlist MainShow
    [00:01:00.00] Play Flim on 1
    [00:02:30.15] Play Bells on 2
Endlist
```

Event Command

Syntax: `EVENT on`

Syntax: `EVENT off`

Syntax: `EVENT = "name"`

Syntax: `EVENT offset "timer/constant/variable"`

Parameters:

`"name"` — The Eventlist name to activate.

`"timer/constant/variable"` — Positive or negative frame offset from the system timer.

The EVENT command must appear in the sequence program body to specify the active event list. The following example shows event selection with trigger-based control and time-of-day scheduling:

Example:

```
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 3 PM offset
EVENT on                  ; Activate the event list
J1
    if Close1 == 1        ; Trigger 1 closes: stop the event list
        EVENT off
    endif
    if Open1 == 1         ; Trigger 1 opens: activate the event list
        EVENT on
    endif
    if Close2 == 1        ; Trigger 2 closes: change the show
        EVENT = sideshow
    endif
    if Open2 == 1         ; Trigger 2 opens: restore the main show
        EVENT = mainshow
    endif
    Goto J1
```

Event Sensitivity

Syntax: `TIMER window "timer/constant/variable"`

Syntax: `TIMER setback "timer/constant/variable"`

Parameters:

`window` — Timer changes greater than this value trigger a full event list recalculation.

`setback` — Allowable event timing jitter. Values less than expected jitter may cause multiple triggering.

Both window and setback are also readable as system variables.

Example:

```
TIMER window 30      ; Any jump > 30 frames causes full recalculation
TIMER setback 2      ; Allow 2-frame timing jitter
x = window           ; x will contain the current window setting
x = setback          ; x will contain the current setback value
```

Cue List Commands

Syntax: `Cue "timer/constant/variable" "command"`

Syntax: `Cue group "group number" "timer/constant/variable" "command"`

Parameters:

`"timer/constant/variable"` — Timer value in frames. A time-formatted constant like [00:14:23.25] is typical.

`"command"` — Any valid sequence instruction or console directive.

`"group number"` — Group number index (1–64).

Loads a timed command into the cue list.

Example:

```
Cue [00:00:05.00] Play Flim on 1
Cue group 1 [00:00:10.00] Play Bells on 2
```

Cue Relative

Syntax: `Cue group "group number" now`

Syntax: `Cue group "group number" never`

Parameters:

`"group number"` — Group number index (1–64).

Cue group now sets the group offset time to the current timer value. All subsequent Cue group commands for this group add this offset to the specified time.

Cue group never clears the offset to zero, restoring absolute timing for that group.

Example:

```
Cue group 1 now    ; Set group 1 offset to current timer value
```

Cue Clear

Syntax: `Cue clear all`

Syntax: `Cue clear group "group number"`

Parameters:

`"group number"` — Group number (1–64).

Purges pending cues from the cue buffer.

Example:

```
Cue clear all
Cue clear group 1
```

Cue Print

Syntax: `Cue print<c/d/e/m/x>`

Syntax: `Cue print<c/d/e/m/x> next`

Syntax: `Cue print<c/d/e/m/x> group "group number"`

Parameters:

`print suffix` — c=Console, d=DTE, e=Ethernet, m=MIDI, x=DMX. Default: originating port.

`next` — Print only the next pending cue.

`group` — Print only cues for the specified group.

Prints the current cue buffer to the specified port.

Example:

```
Cue print          ; Print all cues to originating port
Cue printc         ; Print to Console
Cue print next     ; Print next pending cue
```

Timer Synchronization

Multiple units can be synchronized over UDP Ethernet using port 11001. The protocol uses a single master and multiple slaves. Timer accuracy is guaranteed to within one frame (1/30 sec); typical wired networks achieve ±1ms synchronization.

Timer Sync Commands

Syntax: `Sync on`

Syntax: `Sync off` (default)

Syntax: `Sync rate 10000` (default 10 seconds)

Syntax: `Sync now`

Syntax: `Sync block`

Syntax: `Sync allow`

Syntax: `TIMER master`

Syntax: `TIMER slave` (default)

Parameters:

`Sync on/off` — Enable or disable periodic synchronization.

`Sync rate` — Set the synchronization interval in milliseconds.

`Sync now` — Force immediate synchronization to this unit's timer.

`Sync block` — Disable reception of timer synchronization from the network.

`Sync allow` — Re-enable reception of timer synchronization.

`TIMER master` — Designate this unit as the synchronization master.

`TIMER slave` — Designate this unit as a synchronization slave (default).

At power-up, all units default to slave mode. Exactly one master must be designated for coordinated timer commands to propagate across the network.

Example:

```
TIMER master                ; Set this unit as master
Sync on                     ; Enable periodic sync
Sync rate 5000              ; Synchronize every 5 seconds
```

Synchronizing Audio to the Master Timer

Audio tracks can be synchronized to the main timer using the Lock command. Playback is throttled using a pitch algorithm to maintain synchronization. Tracking accuracy is approximately ±3 ms. If more than ±1 second out of sync, the track abruptly skips to the correct position. Stopping the master timer also stops locked tracks.

Lock / Unlock Sound Tracks

Syntax: `Lock "track # list" to "offset time (frames)"`

Syntax: `Unlock "track # list"`

Parameters:

`"track # list"` — List of tracks to lock or unlock.

`"offset time (frames)"` — Optional frame offset from the main timer.

Locks the specified tracks to the main timer. If a sound is started before the timer + offset is reached, it is pre-loaded and held until the target time.

Example:

```
Define toffset
toffset = TIMER
toffset = toffset + 300     ; Add 10-second offset (300 frames)
Lock 1 to toffset           ; Lock track 1 to timer + 10 seconds
Play sound1 on 1 T1
; Later, to unlock:
Unlock 1
```

Custom Audio Processing

This section describes how to configure and customize the Sigma DSP-based audio processing on the MS-2 and rPod models. Sigma Studio (available free from Analog Devices) is required.

System Description

The MS-2, rPod10.2, and rPod8.4 use multiple DSP processors to generate audio from analog inputs or CF card sources. The Blackfin main processor manages system coordination, track volume, and polyphony. Each Sigma DSP processor receives two-channel audio data and performs additional processing:

- Channel volume control
- Multi-band equalization
- Subwoofer crossover filtering
- Dynamics processing (compression, expansion, midnight control)
- Spatial processing (fat stereo, dynamic bass)

- Line delay for speaker positioning and phase correction

Finally, each Sigma processor converts the 2.1 audio data to analog signals using the onboard triple digital-to-analog converters.

The figure below illustrates the complete rPod8.4 audio signal processing architecture, showing the relationship between the two Blackfin processors, the four AD1953 output codec/processors, the ADAU1701 input processor, and the inter-processor dual-port memory interface.

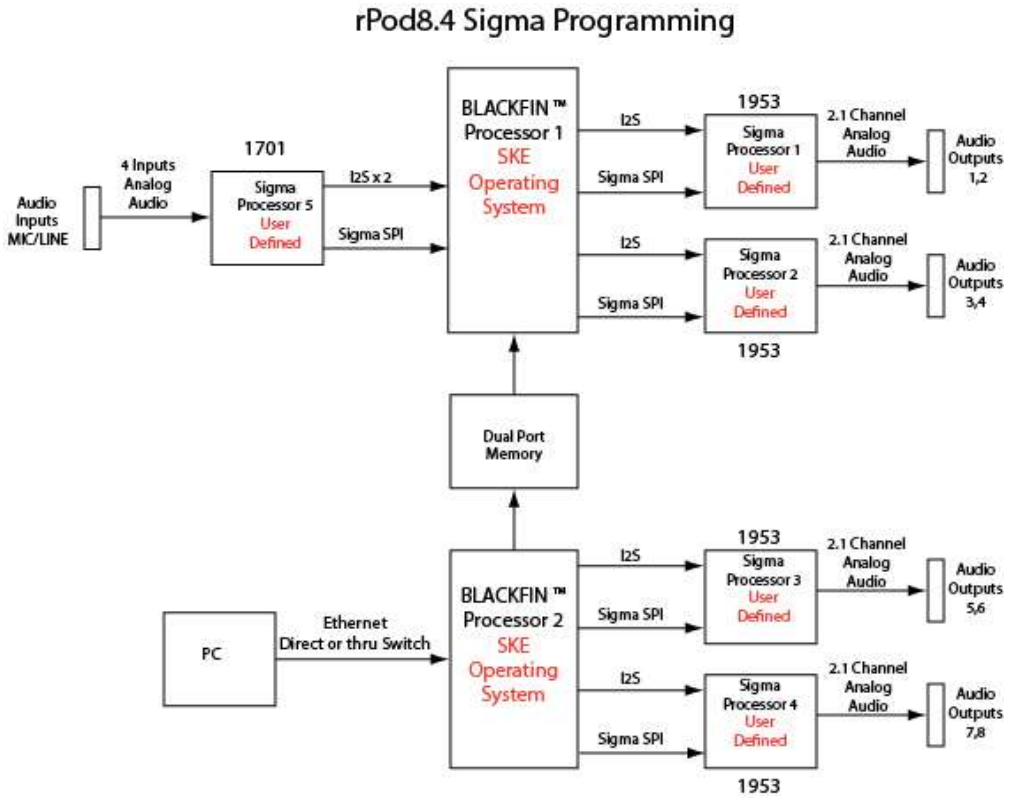

Figure 12 — rPod8.4 Sigma DSP programming architecture. Analog audio inputs are processed by the ADAU1701 (Sigma Processor 5, user-defined) before being passed to the two Blackfin processors via I2S. Each Blackfin processor drives two AD1953 Sigma output processors via Sigma SPI, producing four stereo (2.1) analog output pairs on channels 1–2, 3–4, 5–6, and 7–8. The PC communicates with all processors via Ethernet through either a direct connection or a network switch.

Using the Default Model

Each Sigma processor can use a default program model or a user-defined model. The design models reside on the CF card and are downloaded at power-up. For the AD1953 audio codecs, a default model is contained in onboard ROM and transferred to program memory at power-up. No custom files are required if using the default model.

Creating a Custom Design

If a custom design is required, the program and parameter setup files are placed on the CF card and automatically transferred by the SKE operating system at boot. An equivalent Sigma Studio model of the default design is provided by SKE (contained in the 1953 Model folder). It includes a seven-band EQ stage, crossover filtering, master volume control, spatial processing, dynamics processing, and a sub-channel re-injection stage.

The figure below shows the default SKE Sigma Studio schematic for the AD1953 output codec.

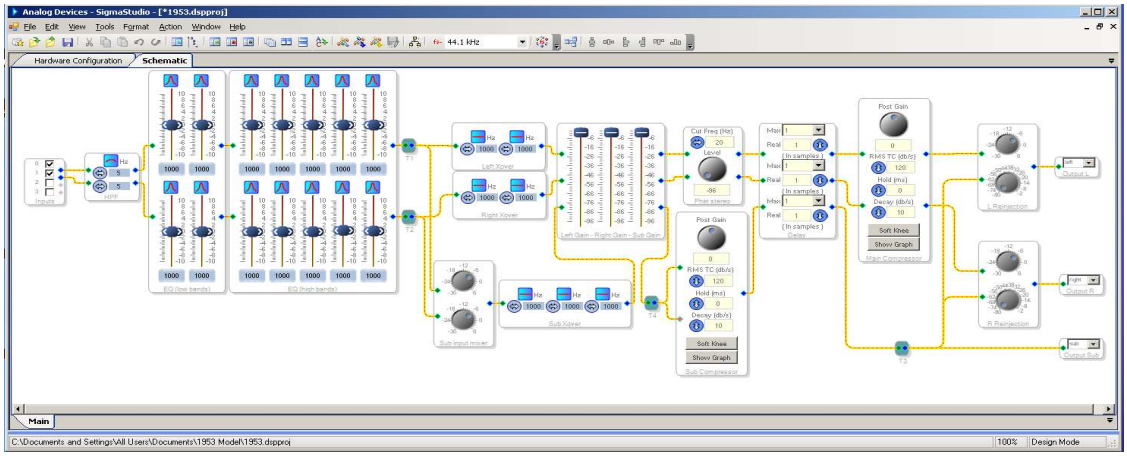

Figure 13 — Default SKE Sigma Studio schematic (1953 Model). The signal chain includes a high-pass filter (HPF) input stage, low-band and high-band seven-band EQ sections, left/right crossover filters, a sub-channel input mixer, master volume control (Left Gain, Right Gain, Sub Gain), Phat Stereo spatial processing, main compressor, sub compressor, RMS time-constant processing, and left/right re-injection output stages. This model can be customized by the user within Sigma Studio.

An alternate example model is also provided in the 1953 example 1 folder, demonstrating an alternative signal chain configuration.

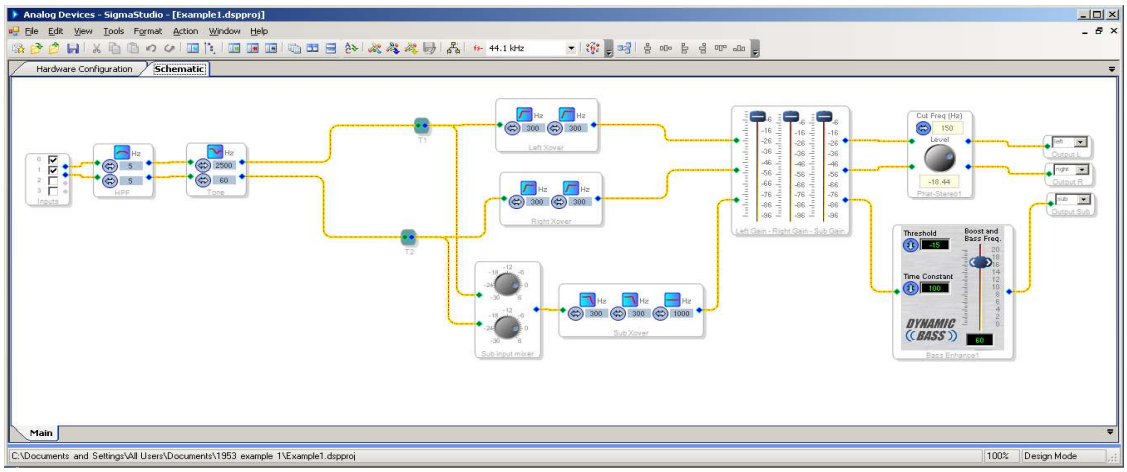

Figure 14 — Alternate Sigma Studio schematic (Example 1). This simplified model includes an HPF input stage, a Tone control stage, left/right and sub crossover filters, Left Gain/Right Gain/Sub Gain master volume, Phat Stereo spatial processing, and a Dynamic Bass Enhance stage. This design may be used as a starting point for a custom implementation.

> **NOTE:** *All design elements must be compatible with the AD1953 processor. Users are free to develop their own schematics using Sigma Studio, within the constraints of the AD1953 device.*

Input DSP Processing (rPod8.4 r1.2 and Later)

The rPod8.4 r1.2 and later feature an input DSP that allows Sigma Studio to process signals from MICL, MICR, LINEL, and LINER:

- Select ADAU1701 as the DSP Processor in Sigma Studio (instead of AD1953).
- Generated files must be named ProgD1.hex and SetupD1.hex.
- Place these files on CF Card 1.

Sigma Studio Setup and Ethernet Connection

To connect a Mini-Sam or rPod to Sigma Studio, a communications channel must be established in the Hardware Configuration tab. Two channel types are available: TCPIP1953 targets an output audio codec processor (AD195x), and TCPIP1701 targets the input audio processor (ADAU1701). The DSP service uses a default Ethernet port of 8086, which must match the port configured in Sigma Studio.

Enable and disable the TCP server connection from the unit's console:

```
Sigma on                ; Enable connection to Sigma Studio
Sigma off               ; Disable connection
Sigma port <port #>     ; Set TCP server port (default: 8086)
```

The Hardware Configuration tab must include a TCPIP communications channel connected to the appropriate processor IC. The figures below show the correct configuration for each processor type.

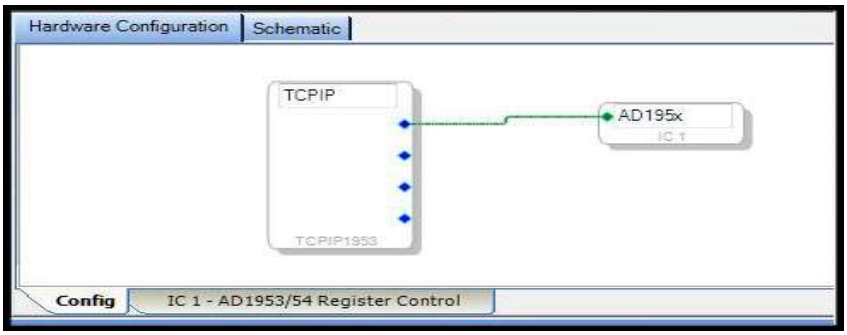

Figure 15 — Sigma Studio Hardware Configuration tab for output codec programming. The TCPIP1953 communications channel is connected to an AD195x processor IC. The Config tab at the bottom identifies the IC as an AD1953/54 Register Control target.

The Sigma Studio Tree ToolBox lists the available processor ICs and communication channel types. When configuring hardware, select the appropriate IC type and channel from this list.

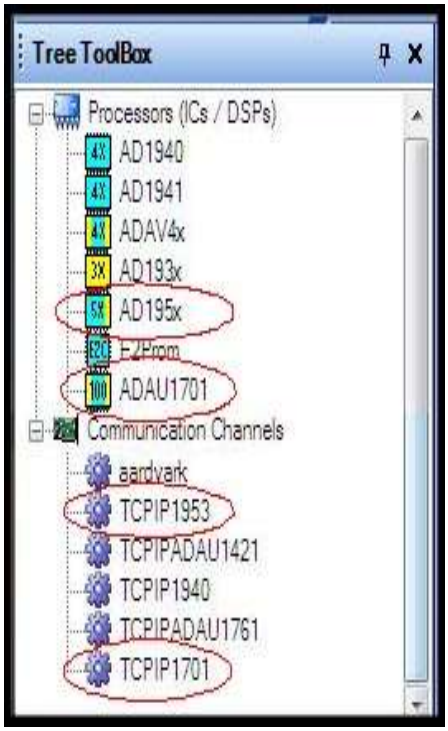

Figure 16 — Sigma Studio Tree ToolBox. The Processors section lists available IC types including AD195x (output codec) and ADAU1701 (input processor). The Communication Channels section lists TCPIP1953 and TCPIP1701 for connecting to each respective processor type.

For input DSP processing using the ADAU1701, the Hardware Configuration tab should use a TCPIP1701 channel connected to an ADAU1701 IC.

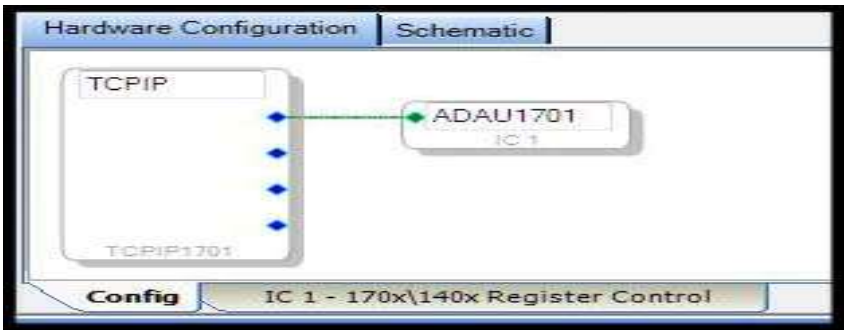

Figure 17 — Sigma Studio Hardware Configuration tab for input processor programming. The TCPIP1701 communications channel is connected to an ADAU1701 IC. The Config tab identifies the IC as a 170x/140x Register Control target.

Mapping Processors to a Sigma Studio Design

This command applies to the AD1953 output codec/processors on the rPod. It maps the Ethernet Streaming Connection nodes to the rPod DSP SPI connections. All four processors must have an assignment. Assignments can be made to non-existing Sigma Studio ICs. The ADAU1701 input processor connections are differentiated by protocol and do not require mapping.

```
Sigma map 1,2,3,4    ; Default: each processor to its own Sigma IC
Sigma map 1,1,1,1    ; Map IC1 to all 4 codecs (channels 1-8)
Sigma map 2,2,1,1    ; IC2 to channels 1-4; IC1 to channels 5-8
```

Establishing the Sigma Studio Connection

After issuing the Sigma on command, the unit opens a TCP server socket and waits for a connection from Sigma Studio. The console will confirm the connection when it is established. The figure below shows a typical console output sequence during connection.

```
Codec program loaded.
Codec program loaded.

============================================================
Simon-Kaloi Engineering Ltd.rpodP2 Audio and Control System
Revision 1.69a, File:rpodP2a069 21OCT2015 44.1K/16bit
============================================================
DATE: Thu 10/22/15 02:55:14 AM
File System Version: HCC_FAT_LFN ver:2.60
MFG: Fri 05/16/14 03:54:41 PM  Serial #13351307290023 F4:1E:26:04:00:44
IP Address: 169.254.144.60:11000 Subnet Mask: 255.255.255.0
Sigma TCP Port: 8086
WARNING:No startup sequence found.
b:>Sigma on
b:>TCP Server Socket Open
b:>TCP Client connected
b:>_
```

Figure 18 — Console output during a successful Sigma Studio connection sequence. After the codec program is loaded and Sigma on is issued, the TCP Server Socket opens and a TCP Client connected message confirms the Sigma Studio session is active.

If Sigma Studio does not connect automatically, right-click the TCPIP communications channel node in the Hardware Configuration tab and open the TCPIPForm dialog. Verify that the IP address matches

the unit and that the port number is 8086 (or the value set by Sigma port). Click Open Connection to establish the link.

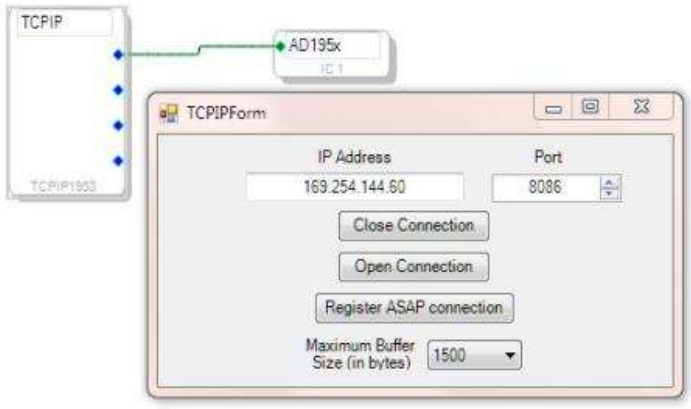

Figure 19 — TCPIPForm connection dialog in Sigma Studio. Enter the unit IP address and confirm the port number matches the value configured with the Sigma port command (default: 8086). Click Open Connection to initiate the TCP session.

Saving the Design

Extracts the program and filter data from the specified processor and writes configuration files to the CF card. The designs must be loaded through Sigma Studio before executing these commands.

```
Sigma save all          ; rPod: save all processors
Sigma save 1            ; Save codec 1 (channels 1-2)
Sigma save 2            ; Save codec 2 (channels 3-4)
Sigma save 3            ; Save codec 3 (channels 5-6)
Sigma save 4            ; Save codec 4 (channels 7-8)
Sigma save 5            ; Save input processor configuration
Sigma save              ; Mini-Sam version
```

ASCII Table

The following table lists the standard 7-bit ASCII character set.

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

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
(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)	26	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

Quick Reference Guide

This section provides a concise command reference for rapid lookup. All commands listed here are described in full detail in the Command Description Index.

File Access

Command	Syntax
Change drive	cd "drive name"
Display directory	dir
Change directory	Chdir "directory name"
Make directory	Makedir "directory name"
Remove directory	Remdir "directory name"
Delete file	del "file name"
Rename file	Rename "original" as "new name"
Open file	filenumber = Openfile "file name" for "mode"
Close file	Closefile "filenumber"
File delimiter config	Config FDELIM = "character list"
Write file (delimited)	Writefile "filenumber" "data list"
Write file (raw)	Writefile # "filenumber" "data list"
Read file (delimited)	Readfile "filenumber" "variable list"
Read file (raw)	Readfile # "filenumber" "variable list"
Read file to string	Readfile $ "filenumber" "string var" <count>
Seek file	position = Seekfile "filenumber" "offset" <R> {CURR/BEG/END}

Sound File Playback

Command	Syntax
Pre-load stereo	Ldsnd "file" on "ch list" T "track list" X "xfade list"
Pre-load mono	Ldsndm "file" on "ch list" T "track list" X "xfade list"
Unload tracks	Unload "track # list"
Play stereo	Play "file" on "ch list" T "track list" X "xfade list"
Play mono	Playm "file" on "ch list" T "track list" X "xfade list"
Loop stereo	Loop "file" on "ch list" T "track list" X "xfade list"
Loop mono	Loopm "file" on "ch list" T "track list" X "xfade list"

Command	Syntax
Un-loop tracks	Unloop "track # list"
Convert to looping	Makeloop "track # list"
Append sound	Append "file" on "track #"
Stop tracks	Stop "track # list"
Stop all	Stop all
Pause tracks	Pause "track # list"
Pause all	Pause all
Resume tracks	Resume "track # list"
Resume all	Resume all
Mirror tracks (rPod)	Mirror "track # list" on/off
Position track	Pos "track # list" to F/S/T "position"
Skip position	Skip "track # list" <</>>/F/S/T "position"
Pitch track	Pitch "track # list" to "pct x10" <in "time (ms)">
Set sample rate	Config SAMPLE "rate"

Sound Volume Control

Command	Syntax
Channel volume	Cvol "ch list" = "volume"
Track volume	Tvol "track list" = "volume"
Duck volume setting	Dvol "track list" = "volume"
Duck track	Duck "track list"
Un-duck track	Unduck "track list"
Input volume	Ivol "ch list" = "volume"
Patch assign (rPod)	Patch A/B = "input ch list"
Input mapping (rPod)	Patch A/B to "output ch list"
Input control (rPod)	Patch A/B on/off
Input mapping (MS-II)	Ain MIC/LIN/all
Input control (MS-II)	Ain on/off
Boost mic gain	Boost "input ch" "gain"
Mute all	Mute
Un-mute all	Unmute

Command	Syntax
Track attack time	Atk "track list" = "time"
Duck attack time	Datk "track list" = "time"
Track decay time	Dek "track list" = "time"
Duck decay time	Ddek "track list" = "time"
Amplifier (MS-II)	Amp on/off/stby/mute

Clips

Command	Syntax
Load clip	Ldclip "file name"
Execute clip	Clip "file name"
Unload clip	Freeclip "file name"

DMX/RS485 and Output Control

Command	Syntax
Bit control (output)	Output "control # list" on/off/toggle
Bit control (DMX)	Dmx "ch #" - "bit #" on/off
Byte control	Dmx "ch #" = "output value"
Dimming control	Dimto "value" in "time" on "ch list"
Driver enable	Dmx DRIVE
Driver release	Dmx RELEASE

DMX File Playback

Command	Syntax
Pre-load DMX	Lddmx "file" on "track #"
Unload DMX (list)	Unloaddmx "track # list"
Unload DMX (all)	Unloaddmx all
Play DMX file	Playdmx "file" on "track #"
Loop DMX file	Loopdmx "file" on "track #"
Mount DMX file	Mount "file" on "track #"
Unmount DMX file	Unmount "track # list"
Stop DMX tracks	Stopdmx "track # list"

Command	Syntax
Stop all DMX	Stopdmx all
Position DMX	Posdmx "track list" to F/T "position"
Skip DMX position	Skipdmx "track list" to F/T "position"

DMX File Capture

Command	Syntax
Record DMX	Recdmx "file" from "ch#" to "ch#"
Pre-load record DMX	Ldrecdmx "file" from "ch#" to "ch#" T "sound track #"
Start recording	Start
Stop recording (1)	Stop all
Stop recording (2)	Stopdmx all

Ethernet

Command	Syntax
Configure Ethernet	Config ENET <"ip addr"> : <"port"> / <"subnet">
Configure gateway	Config GATEWAY "ip address"
Status reporting	Verbose ENET on/off
FTP password	Config PASSWORD "string"
UDP mode	Config UDP Manual/Auto
IP binding	Bind "ip address" to "bind number"
Send command (const)	Sendcmd "bind #": "command string"
Send command (var)	Sendcmds "bind #": "string variable"
Send data	Send "bind #".<"pkt #">-<"type">|"data list"
Print to Ethernet	Printe "bind #"|"data list"
Receive data	Recv "variable list"
Read UDP	Read UDP
Peek UDP	Peek UDP
UDP termination	Config ENET Term "method"

System Commands

Command	Syntax
Re-boot	Boot
Set time base	Config TIMEREF ms/frames/mixed
Save cosmic vars	Update
Restore cosmic vars	Reload
Remove cosmic var	Remove "variable list"/all
Start playback	Start <T "track list" D "ch list" C "clip list">
Query playing tracks	?P
Query track status	?T "track # list"
Print	Print(port) "variable list"
Print raw	Print(port) # "variable list"
Monitor port (r1.2)	Monitor Console/DMX/DTE/Midi on/off
Monitor all off	Monitor all off
Config serial port	Config Console/DTE/DMX = baud, bits, stop, parity
Config termination	Config Console/DTE Term = "char list"
Termination control	Config Console/DTE Term on/off
DMX config	Config DMX Standard/RS485
MIDI config	Config Midi Standard/RS232
DMX receiver	Config DMX Receive on/off
Null substitution	Config Console/DTE nullsub on/off
Read serial port	Read Console/DTE <#/&> <count>
Peek serial port	Peek Console/DTE <#/&> <count>
Silent mode	Silent <port/all> on/off
Verbose mode	Verbose on/off
Install OS	Install "file name"
Status LEDs	Status "output list" on/off
Configure triggers	Config TRIG "trigger list" = "de-bounce time"

Logging

Command	Syntax
Log on	Log on
Log off	Log off
Log update (flush)	Log update
Log clear	Log clear

Command	Syntax
Log list	Log list
Log mode	Log freq none/hour/day/month
Log file handling	Log replace true/false
Log timestamp	Log timestamp on/off

Sequence Programming Direct Commands

Command	Syntax
Set context	Context "sequence # (1-16)"
Define local int	Define "variable"
Define global int	Define Global "variable"
Define local string	Define $"variable"
Define global string	Define Global $"variable"
Define public string	Define Public $"variable" (rPod8.4)
Group enumeration	Formgroup "file" as "group #"
Play group (file)	Playgroup "file name"
Play group (number)	Playgroup "group #"
Play group (variable)	Playgroup "variable"
Load sequence	Ldseq "file" on "sequence # (1-16)"
Play sequence	Playseq "sequence # (1-16)"
Start sequence (list)	Startseq "sequence # list"
Start all sequences	Startseq all
Start (Go)	Go
Halt sequence	Halt "sequence # list"
Halt all	Halt all
Step sequence	Step "# of steps"
Set breakpoint	Break "line #/label/var list"
Clear breakpoint	Break "line #/label/var list" off
Clear all breakpoints	Break all off
List sequence	List "start" {to "end" / :"count"}

Program Flow

Command	Syntax
Timer delay	Wait "time (ms)"
Random delay	Wait "min" - "max"
Branch immediate	Goto "line label"
Branch conditional	if / else / endif
Branch on selection	select / case / break / endsel
Do loop	do / while "operation"
For loop	for "var" = "op1" to "op2" / next

Event Programming

Command	Syntax
Event list start	Eventlist "name"
Event list end	Endlist
Activate event list	EVENT = "name"
Event on/off	EVENT on/off
Event time offset	EVENT offset "frames"
Timer window	TIMER window "frames"
Timer setback	TIMER setback "frames"
Cue a command	Cue "time" "command"
Cue with group	Cue group "group #" "time" "command"
Cue group now	Cue group "group #" now
Cue group never	Cue group "group #" never
Clear all cues	Cue clear all
Clear group cues	Cue clear group "group #"
Print cue list	Cue print <c/d/e/m/x>
Print next cue	Cue print next
Print cues by group	Cue print group "group #"

Timer Synchronization

Command	Syntax
Sync on/off	Sync on/off
Sync rate	Sync rate "ms"

Command	Syntax
Sync master	TIMER master
Sync slave	TIMER slave
Sync now	Sync now
Sync block/allow	Sync block / Sync allow
Lock audio track	Lock "track list" to "offset (frames)"
Unlock audio track	Unlock "track list"

Frequently Asked Questions (FAQ)

How do I read an input trigger and play a sound when the trigger occurs?

The following sequence continuously monitors trigger 1 and plays the sound "Flim" on channel 1 when trigger 1 is asserted:

```
J1  if Close1 == 1
        Play Flim on 1 T1
    endif
    Goto J1
```

How do I cross-fade two sound files?

Set the attack and decay times for both tracks, start the first sound, then use the X flag to cross-fade. The following example cross-fades "Flim" on track 1 with "Music 1" on track 2:

```
Atk 1,2 = 30            ; 1-second attack on tracks 1 and 2
Dek 1,2 = 30            ; 1-second decay on tracks 1 and 2
Play Flim on 1 T1       ; Start playback of Flim
; ... after playing for a while ...
Play "Music 1" on 1 T2 X1   ; Cross-fade: fade out track 1, fade in track 2
; ... to cross-fade back to Flim:
Play Flim on 1 T1 X2         ; Fade out track 2, fade in track 1
```

How do I start a sound file 10 seconds into the file?

Pre-load the file to the desired track, use Pos to position the playback pointer at 10 seconds, then issue the Start command:

```
Ldsnd Flim on 1 T1
Pos 1 to T10            ; Position to 10 ms — NOTE: use T10000 for 10 seconds
in ms mode
Start
```

How do I patch microphone inputs to analog outputs on the rPod10.2 and rPod8.4?

Use the three variants of the Patch command. The following example patches MICL to channels 5 and 7, and MICR to channels 6 and 8, preserving left/right channel consistency:

```
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 channels 1 and 3
Patch B to 2,4           ; Connect right mic to channels 2 and 4
Patch A on               ; Enable the A connection
Patch B on               ; Enable the B connection
.
.
Patch A off              ; Disable the A connection when done
```

```
Patch B off              ; Disable the B connection
```

The diagram below illustrates the signal routing for the configuration above — MICL assigned to Patch A, MICR to Patch B, with Patch A routed to channels 5 and 7, and Patch B routed to channels 6 and 8.

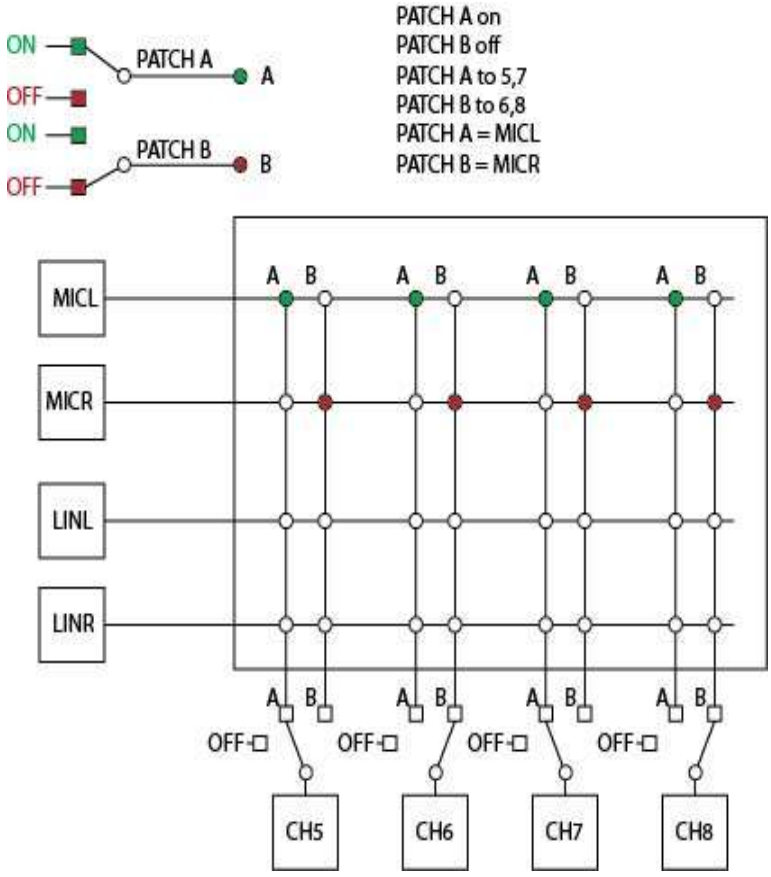

Figure 10 — Software block diagram showing MICL patched to channels 5 and 7 via Patch A, and MICR patched to channels 6 and 8 via Patch B. Patch A is enabled (on); Patch B is disabled (off) in this configuration.

How do I play shows sequentially using a single trigger?

Use a counter variable to track which show has played. The following full example from startup.seq implements five shows with a keep-alive ambient loop. Trigger 1 starts the next show; additional triggers are locked out until the show completes:

```
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     ; Advance to next show
    if showplayed > 4               ; Wrap around to show1?
        showplayed = 0
    endif
    do                             ; Wait until show is complete
    while ?P1 <> 0
endif
Goto J1
```

How do I play random shows using a single trigger?

Use the Rand function and group files. The following streamlined example implements a three-show system using only nine compiled lines in the main sequence. All show commands reside in group files on the CF card:

Main program file (startup.seq):

```
Playgroup init          ; Run initialization from init.seq
J1
    if Close1 == 1      ; Is trigger 1 closed?
        x = Rand 1 3    ; Get a random number from 1 to 3
        Playgroup x     ; Play the random show
        do              ; Wait for track 1 to complete
        while ?P1 <> 0
    endif
    Goto J1
```

Group file init.seq (initialization):

```
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 group 1
Formgroup show2 as 2    ; Enumerate Show2.seq as group 2
Formgroup show3 as 3    ; Enumerate Show3.seq as group 3
```

Group file show1.seq:

```
Ldsnd show1a on 1 T1    ; Load sound a for show1
Ldsnd show1b on 3 T2    ; Load sound b for show1
Lddmx show1dmx on 1     ; Load DMX file for show1
Start                   ; Start show1
```

Group files show2.seq and show3.seq follow the same pattern. Each can be individually tested from the Console:

```
Playgroup show1    ; or
Playgroup show2    ; or
Playgroup show3
```

Can any 8-bit value be stored in a string variable?

Yes. Embedded control characters (such as STX (0x02) or CR (0x0D)) are supported through direct assignment. The following example creates a string containing "Hello", a carriage return and line feed, and "There":

```
CR = 13
LF = 10
$a = "Hello"
$b = "There"
$a = $a + CR
$a = $a + LF
$a = $a + $b
Print $a
Hello
There
```

Embedded control characters can also be printed directly to the port:

```
Printd # 0x02, "1104", 0x0d    ; STX + data + CR to DTE port
```

What are the comment delimiters?

The semicolon (;) is the comment delimiter. It may be placed on its own line or at the end of a command or operation:

```
; This is a full-line comment
Play Flim on 1   ; This is an inline comment
```

Does Val convert the string "HB" into 0x8A (0x48 + 0x42)?

Yes. Val sums the ASCII values of the characters in the string. For "HB": H = 0x48, B = 0x42, sum = 0x8A = 138:

```
$a = "HB"
x = $a
Print x
138             ; 0x8A
```

Will Val convert the string "SC11104" into 0x18D?

Yes. Val sums all ASCII values in the string: S(0x53) + C(0x43) + 1(0x31) + 1(0x31) + 1(0x31) + 0(0x30) + 4(0x34) = 397 (0x18D). To limit the result to a single byte (0x8D = 141):

```
$a = "SC11104"
x = $a
Print x
397             ; 0x18D
x = x & 0xff    ; Mask to 8 bits
Print x
141             ; 0x8D
```

Is $a = $a + $b legal?

Yes. Self-referencing string concatenation is fully supported:

```
$a = $a + $b    ; Appends $b to $a
```