Sidewinder DMX™

Application Programming Interface (API)

Version 1.2

Interactive
Technologies

Interactive Technologies, Inc.
Orlando Office
3509 Mount Berwick Drive
Apopka, FL32712-4742 USA

Toll Free: 888-248-1851
Phone: 407-880-8180
Fax: 407-880-8280

Web Site: http://www.interactive-online.com/
Email: info@interactive-online.com
Technical Support: support@interactive-online.com
Product Info: sidewinder@interactive-online.com

Printed in the United States of America.

Table of Contents

Introduction

The Sidewinder DMX "Virtual DMX Console" is a DMX processor with on-board features similar to many mid-range DMX consoles. These functions do not have any physical controls on the Sidewinder hardware, but are instead entirely accessed through the Sidewinder's RS-232 port. This document describes the method by which external devices can control most of Sidewinder's functions.

At the time this document was prepared, the Sidewinder firmware is still going through some revisions. Most notably, the master/submaster function calls and the accelerated non-ASCII function calls are not documented. Also, there are several private function calls reserved for use by Interactive Technologies that aren't documented.

Since this document is still considered draft quality, please contact sidewinder@interactive-online for any additional technical support needed.

Also, since our products are mostly driven by customer demand, please let us know of any feature requests, bug reports or other comments you may have.

The Figment/Sidewinder Development Team

Sidewinder Serial Port

The serial port on the Sidewinder hardware is configured as a standard RS-232 serial port wired as a host controller (terminal), just like a desktop computer. This was done to make the connection to a Palm OS® compatible handheld device as easy as possible.

Sidewinder provides a Male DB-9 connector for connection with external equipment.

To connect Sidewinder to a desktop computer (or other standard serial device), a “Null Modem” serial cable is required. Null Modem cables typically have Female DB-9 cable connectors on each end of the cable.

Electrically the serial port meets or exceeds EIA-232-F, can accept +/- 30V input levels and is ESD protected to 2000 V per MIL-STD-883, Method 3015.

Logically, the serial communication format uses 8-data bits, 1 stop bit and no parity (8-1-N). The baud rate (bits per second) are software selectable. In normal operation, Sidewinder’s bootstrap code sends and receives low-level boot data at 9600 baud. The Figment application communicates with Sidewinder at 57600 baud.

Startup Sequence

Basic Startup

When power is applied to Sidewinder, its bootstrap firmware executes. This small application program is responsible for initializing the hardware, validating the Flash ROM and executing the main Flash application.

When the bootstrap application starts up, it sets the serial port to 9600 baud, 8 data bits, 1 stop bit and no parity (8-1-N). Also, the Red Host Status LED is illuminated to demonstrate that the bootstrap application is running. A short bootstrap message is sent to the serial port:

```
Sidewinder Bootstrap Build 15(B3)
```

After initialization is complete and the bootstrap application has determined that a valid application program is available in the main Flash ROM, it launches the main application. The bootstrap application acknowledges this event by displaying a message similar to:

```
Flash OK, Jumping to $E080
```

Next, the main Sidewinder application begins running. The serial port is switched to its default buad rate of 57600. Now, the main application sends its startup message:

```
Sidewinder DMX(1.2.0) Ready
```

At this point, Sidewinder is running and ready to accept commands from the host.

Firmware Loading

If the Sidewinder is powered-up with both the M1 and M4 buttons held down (the Triangle and X keys on the portable Sidewinder), then the bootstrap application enters a special Firmware Download mode.

In this mode, the Red Host Status LED continues to glow and the following message appears:

```
Begin S-Record Download...
```

A this point, a Motorola “S19” object file may be sent to the bootstrap application, which it will decode and re-flash into the main application Flash ROM area. The bootstrap application responds with messages that indicate its progress with each step of decoding, erasing flash, programming the new software, etc.

Please note that the process of downloading firmaware to Sidewinder is very likely to change in the future. For best results, please check for specific firmware download instructions with each software release as they become available.

When a successful firmware download and re-flash has completed, the bootstrap application then executes the main program.

Erasing User Memory

One of the first functions of the main Sidewinder application is to check the state of the X button. If it is held down then the main user flash memory is erased. This function only erases downloaded show file information (resource data) and not the main software application. While this process completes, the Host Status LED glows Orange.

This function is useful if the show file (resource data) becomes corrupted in some way that causes the Sidewinder application to not function properly. Since the user memory erase function occurs first, the show data can be deleted before it is accessed.

Automatic Show Execution

If the currently downloaded show contains instructions to execute a “start-up macro”, then the Sidewinder application executes these instructions.

See the Resource Manager section for the format of the various resource chunks that comprise a working “stand-alone” show file.

Command Format

Sidewinder accepts commands in two different formats.

The first is the ASCII Command Format, which uses strings of printing characters that are available from a standard computer keyboard. These are alphabetic, numeric and punctuation characters. Although these commands are easy to enter, they require more bytes to be sent between the Sidewinder and Host to complete a command and/or response. If you are communicating with Sidewinder from a terminal program, you will use this format.

The second format is the Binary Command Format, which uses binary data strings to communicate between the Sidewinder and Host. This format requires software at the host to produce and receive these un-typeable command strings. At this time the Binary Command Format commands are undocumented as they are changing.

ASCII Command Format

All ASCII-based commands and responses sent to or from the Sidewinder are in the following general format. For commands sent to Sidewinder:

```
<(Command)(Parameters)(Checksum)>
```

And for responses sent from Sidewinder:

```
<(Response)(Parameters)(Checksum)>
```

Each command string starts with a “less-than” symbol (<), a few characters that determine the command, the appropriate command data, a checksum and then the “greater-than” symbol (>). A command response from Sidewinder is formatted in a similar way. For example, the “Ping” command is:

```
<?06F>
```

The command is “?0” and the checksum is “6F” (there are no parameters). When Sidewinder receives this command, a response similar to the following is returned:

```
<?0000002F>
```

Here, the response is “?0” with the data “0000” and a checksum of “2F”. The full description of this command and many others are described later in this document.

Calculating ASCII Checksums

Each command contains a two-character checksum just before the ">". It is computed by taking the sum of the ASCII values of each character after the "<" and before the checksum. The lowest 8 bits of the resulting sum are converted into ASCII hexadecimal and inserted as the checksum. For example:

Command	=	?0
Checksum	=	ASCII('?') + ASCII('0')
	=	3F + 30
	=	6F
Final CommandString	=	<?06F>

Clearly, it is relatively easy for software to compute and append these checksum characters to commands, but it is somewhat difficult to do by hand. In the case where the user doesn't want to use checksums two asterisk characters can be inserted instead. For example:

Command	=	?0
No Checksum	=	<?0**>

This method bypasses the necessity to do checksum computation, but it disables Sidewinder's ability to determine if a command string has been corrupted in any way.

Sidewinder will accept only command strings that either have a properly computed checksum or the "**" checksum. Otherwise, Sidewinder will reject the command and briefly flash the Red status LED.

Binary Command Format

At this time the alternate binary format used for several commands is not documented because the format is expected to change in an upcoming release of the Sidewinder firmware. Please contact Interactive Technologies if you need additional information not provided here.

Sidewinder Commands

The following pages document Sidewinder's public command set. This version of this document only documents basic Sidewinder commands. Others are either not fully implemented at this time or are not being published. If additional information is needed, please contact sidewinder@interactive–online.com.

Device Control Commands

Device control commands typically set or monitor low-level functions of the Sidewinder hardware. These functions include validating the serial connection, setting modes, serial port functions, determining firmware versions, serial numbers and more.

Ping Command

Command: <?0[mm](**)>
Response: <?0(mm)(bb)(**)>
Parameters: (mm) <> Mode Flags (optional in command, always in response)

$01	=	Diagnostic Mode Enable
$02	=	DMX Output Off
$04	=	DMX Output Full-Speed
$08	=	reserved
$10	=	Invert DMX Output Signal
$20	=	Invert DMX Input Signal
$40	=	reserved
$80	=	Stand-Alone Mode (read only)

(bb) < Button Flags

$01	=	M1 pressed (closed)
$02	=	M2 pressed (closed)
$04	=	M3 pressed (closed)
$08	=	M4 pressed (closed)
$10	=	M5 pressed (closed)
$20	=	M6 pressed (closed)
$40	=	M7 pressed (closed)
$80	=	M8 pressed (closed)

(**) <> Checksum

This command "pings" the device. A simple response is returned if the device is connected and operating properly. The parameters in the response convey basic information about the operation of the device, such as if the diagnostic mode is running, if stand-alone is running, if pins are swapped and if buttons have been pressed.

The **Mode Flags** are optional when the command is sent from the host. If the host wishes to set the flags, the byte can be sent. If the host does not want to change the flags but just receive their current state, then the byte can be omitted.

The **Diagnostic Mode Enable** bit is used to enable the special diagnostic mode, which is needed for several of the special diagnostic functions (described in the Diagnostic Commands section).

The **DMX Output Off** bit turns of the DMXOutput port when set. The **DMXOutput Full-Speed** bit disables the "dynamic update rate" feature, causing the DMXOutput port to update at its fastest rate possible (given the current processor load).

The **Invert DMX Output** and **Invert DMX Input** bits enable hardware pin-swapping circuitry that reverses the polarity of the Data + / - pins on either the DMXInput or DMXOutput ports.

The **Stand-Alone Mode** bit is a read-only bit that indicates if an internal playback fader is running a "stand-alone" cue from internal memory.

In the **Button Flags** response byte, each bit corresponds to one of the physical buttons on the front panel of the Sidewinder (or dry-contact closure inputs). When a button is currently pressed (or the contact is closed), the corresponding bit is set in the mask byte.

The physical buttons on Sidewiner automatically send a Button Transition response string to the host when a button is pressed. This allows the host to be notified of button presses as they happen, instead of requiring that the host continue to poll the device using the Ping command.

Version Command

Command:	<?1(**)>		
Response:	<?1(mm)(n)(r)(s)(bbb)(**)>		
Parameters:	(mm)	<	Major feature level ($00 - $FF)
	(n)	<	Minor feature level ($0 - $F)
	(r)	<	Revision level ($0 - $F)
	(s)	<	Release stage
			$0 = Development
			$1 = Alpha
			$2 = Beta
			$3 = Release
	(bbb)	<	Build number ($000 - $FFF)
	(**)	<>	Checksum

Returns the current firmware version of the main Sidewinder application.

For example, version 3.5.15b12 results in the following response:

```
<?1035F200CF0>
```

Please note that earlier versions of Sidewinder used a different scheme for communicating the firmware verison. The older scheme returned 9 ASCII characters in the parameter section of the command (15 total characters) instead of the 8 ASCII characters (14 total characters) in the present scheme.

Serial Number Command

Command:	<?2(**)>		
Response:	<?2(ssssss)(**)>		
Parameters:	(ssssss)	<	ASCII Serial Number
	(**)	<>	Checksum

Returns the serial number of the Sidewinder device in a 6-character ASCII string.

For example, if the unit's serial number is 870101, then the response from this command will be:

<$2870101A2>

Battery Voltage Command

Command:	<?3(**)>		
Response:	<?3(vv)(**)>		
Parameters:	(vv)	<	Battery Voltage (00 - FF)
	(**)	<>	Checksum

Returns the current power supply voltage. A return value of $00 is equal to 0.00 volts. A return value of $FF is equal to 5.00 volts. Each step in value is equal to approximately 19.6 mV.

Battery operated Sidewinder devices typically return a value between 0.50 and 3.50 volts. Sidewinder devices that are line powered (or are using an external AC adaptor) typically return a value above 4.00 volts.

Baud Rate Command

Command:	<?4(bbbb)(**)>		
Response:	<?4(bbbb)(**)>		
Parameters:	(bbbb)	<>	Baud Rate Divisor
	(**)	<>	Checksum

Changes the baud rate of the RS-232 port to a rate as determined by the **Baud Rate Divisor**.

The command string is sent to Sidewinder and the response is received from Sidewinder in the original baud rate. After the response is returned to the host, the device switches its baud rate to the newly requested baud rate after a 250 ms (0.25 second) pause.

The Baud Rate Divisor is calculated by the following equation:

```
Baud Rate  =  16000000 / ( Divisor * 16 )
```

For convenience, the following table lists several common baud rate divisors:

Divisor	Baud Rate				
0D05	300.03	bps	(similar to	300	baud with 0.01% error)
0341	1200.48	bps	(similar to	1200	baud with 0.04% error)
01A1	2398.08	bps	(similar to	2400	baud with 0.08% error)
00D0	4807.69	bps	(similar to	4800	baud with 0.16% error)
0068	9615.38	bps	(similar to	9600	baud with 0.16% error)
0034	19230.77	bps	(similar to	19200	baud with 0.16% error)
001A	38461.54	bps	(similar to	38400	baud with 0.16% error)
0011	58823.53	bps	(similar to	57600	baud with 2.08% error)
0009	111111.11	bps	(similar to	115200	baud with 3.68% error)

Feature Code Command

Command:	<?5(cccccccc)(**)>		
Response:	<?5(rr)(**)>		
Parameters:	(cccccccc)	>	Feature Code ($00000000 - $FFFFFFFF)
	(rr)	<	Result ($00 = success, all others fail)
	(**)	<>	Checksum

Stores the given feature code into Sidewinder's nonvolatile memory. The result code will return a non-zero value if an error occurred in processing or storing the feature code.

Please note that if this command returns a zero ($00) result code, it does not necessarily mean that the feature code matched the device's selectable feature. After successfully sending a feature code to Sidewinder, you should check the availability of the feature by using a command that accesses the specific feature you intended to enable.

Button Transition Response

Command:	n/a		
Response:	<?B(bb)(**)>		
Parameters:	(bb)	<	Button Flags
			$01 = M1 press (close) transition detected
			$02 = M2 press (close) transition detected
			$04 = M3 press (close) transition detected
			$08 = M4 press (close) transition detected
			$10 = M5 press (close) transition detected
			$20 = M6 press (close) transition detected
			$40 = M7 press (close) transition detected
			$80 = M8 press (close) transition detected
	(**)	<>	Checksum

This response is automatically sent from the Sidewinder whenever a physical button (or contact closure input) makes a transition from released to pressed (or open to closed).

Note that this response is not sent to the host when a physical button is mapped to execute a macro by the data in a show file stored in Sidewinder's internal memory. In this case, Sidewinder executes the macro and does not generate a Button Transition Response.

Live Buffer Commands

The Live Buffer commands operate on Sidewinder's live control buffer, which is used to modify the DMX Output of the device with the highest possible priority. Any channel levels set in this highest-priority buffer will entirely mask any channel levels coming from a playback fader, submaster or even any channel levels present at the DMXInput port.

These commands provide the most direct control over DMX Output channels. Any levels set using this method will always override any other source of levels for a particular channel.

Live Buffer Set

Command:	<L(ccc)(nnn)(vv..vv)(**)>		
Response:	<L(ccc)(nnn)(**)>		
Parameters:	(ccc)	<>	Starting Channel Number (zero based, $000 - $1FF)
	(nnn)	<>	Channel Count ($001 - $200)
	(vv..vv)	>	Channel Level String (each $00 - $FF)
	(**)	<>	Checksum

Sets the nnn channels in the live output buffer starting with channel ccc to the levels in the string vv..vv . These channel values will output immediately, since the live buffer has highest priority in the DMX Output stack.

The **Starting Channel Number** is zero based, so valid values are from $000 (channel 1) to $1FF (channel 512).

The **Channel Count** value can range from $001 (one channel) to $200 (512 channels).

The **Channel Level String** contains only nnn hexadecimal values.

For example, to set the four channels starting at channel 12 to the levels $AA, $BB, $CC and $DD respectively, the Live Buffer Set command is formed as follows:

<L00B004AABBCCDD**>

Live Buffer Release All

Command:	<LF(**)>		
Response:	<LF(**)>		
Parameters:	(**)	<>	Checksum

Releases all of the live buffer output channels. If any channels in the live buffer had previously been set by the Live Buffer Set command, this command releases them and allows lower priority channel levels (playback faders, submasters, DMXInput, etc.) to appear in the DMX Output.

Live Buffer Release Mask

Command:	<LE(mm..mm)(**)>		
Response:	<LE(**)>		
Parameters:	(mm..mm)	>	Mask String (64 bytes, 512 bits)
	(**)	<>	Checksum

Releases the specified channels from the live buffer. Each channel that is indicated in the **Mask String** will be released from the live buffer, channels that are not included in the **Mask String** will not be affected by the operation.

The **Mask String** is formed by converting a 64 byte bit field into ASCII hexadecimal characters. The most significant bit of the first byte corresponds to channel 1. The least significant bit of the last byte corresponds to channel 512.

The following example shows the Live Buffer Release Mask command formed to release channels 12 through 18:

<LE001FC00000 (114 more "0"s omitted) 0000**>

Live Buffer Set Mask

Command:	<LD(vv)(mm..mm)(**)>		
Response:	<LD(vv)(**)>		
Parameters:	(vv)	<>	Channel Level ($00 - $FF)
	(mm..mm)	>	Mask String (64 bytes, 512 bits)
	(**)	<>	Checksum

Sets all channels specified by the channel mask string mm..mm to the value vv.

The **Channel Level** is a single hexadecimal value from $00 to $FF.

The **Mask String** is formed by converting a 64 byte bit field into ASCII hexadecimal characters. The most significant bit of the first byte corresponds to channel 1. The least significant bit of the last byte corresponds to channel 512. This function operates on each channel with a 1 in its corresponding position in the Mask String.

Playback Fader Commands

The Playback Fader commands manipulate the internal playback faders. Each fader is capable of of controlling any group of channels on the stage and they can be used to perform dipless crossfades from their previous channel levels to the new destination channel levels.

Playback Go

Command:	<P(f)0(mm..mm)(vv..vv)(tttt)(**)>		
Response:	<P(f)0(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(mm..mm)	>	Mask String (64 bytes, 512 bits)
	(vv..vv)	>	Channel Level String (each $00 - $FF)
	(tttt)	>	Crossfade time ($0000 - $FFFF)
	(**)	<>	Checksum

Writes the channels specified by the mask `mm..mm` with the given chanel levels in `vv..vv` into playback fader `f`. Only the channels specified in the mask appear in the channel level string. The time `tttt` is the desired fade time.

The **Fader Number** is a designation of which playback fader this command should operate on. At this time, only fader zero (`0`) is implemented.

The **Mask String** is formed by converting a 64 byte bit field into ASCII hexadecimal characters. The most significant bit of the first byte corresponds to channel 1. The least significant bit of the last byte corresponds to channel 512. This function operates on each channel with a 1 in its corresponding position in the Mask String.

The **Channel Level String** is a string of hexadecimal bytes that correspond to each of the channels that are indicated in the Mask String. Only channels with a 1 in the corresponding position of the Mask String appear in the Channel Level String.

The **Crossfade Time** is a 16-bit hexadecimal value from $0000 to $FFFF. The Crossfade Time is specified in 1/10ths of a second. For example, a time of `0001` is equal to 0.1 seconds and a time of `000C` is equal to 1.2 seconds. A Crossfade Time of zero (`0000`) indicates that the new scene data should appear immediately on stage.

For example, the following command plays a cue back that has channels 5, 6, 7 and 8 set to $11, $22, $33 and $44 respectively, with a fade time of 1.5 seconds. To make the example more readable, 124 zero characters were omitted from the Mask String:

<P000F00 (124 mask characters omitted)11223344000F**>

Playback Go Stand-Alone

Command:	<P(f)1(nnnn)(**)>		
Response:	<P(f)1(nnnn)(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(nnnn)	<>	Stand-Alone Cue Number ($0000 - $270F)
	(**)	<>	Checksum

Executes the specified cue `nnnn` from internal memory. Sidewinder will switch to "Stand-Alone" mode. If the cue contains timing or links to other internal cues, these functions will be activated.

All internal cue data is stored via Sidewinder's Resource Manager. See the section on the Resource Manager for information regarding using the Resource Manager commands and the format of the various data types used to specify cue data.

The **Fader Number** is a designation of which playback fader this command should operate on. At this time, only fader zero (`0`) is implemented.

The **Stand-Alone Cue Number** is a hexadecimal number from $0000 to $270F. This parameter is specified as the cue number multiplied by ten. Since cue numbers can range from 0.1 to 999.9, this results in Stand-Alone Cue Numbers from $0001 to $270F. Specifying cue number $0000 is interpreted as "the first cue", which will execute the first cue found in memory, regardless of its cue number.

For example, the following command executes Cue 2.5 from internal memory:

```
<P010019**>
```

Playback Clear

Command:	<P(f)C(tttt)(**)>		
Response:	<P(f)C(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(tttt)	>	Fade time ($0000 - $FFFF)
	(**)	<>	Checksum

Clears all channels from the specified playback fader. After this command completes, the playback fader will not be contributing to the look on stage in any way. An optional fade time may be specified that causes the clear operation to fade to "black" more slowly.

The **Fader Number** is a designation of which playback fader this command should operate on. At this time, only fader zero (`0`) is implemented.

The **Fade Time** is a 16-bit hexadecimal value from $0000 to $FFFF. The Fade Time is specified in 1/10ths of a second. For example, a time of `0001` is equal to 0.1 seconds and a time of `000C` is equal to 1.2 seconds. A Crossfade Time of zero (`0000`) indicates that the Playback Clear operation should appear immediately on stage.

Playback Halt

Command:	<P(f)D(**)>		
Response:	<P(f)D(pp)(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(pp)	<	Position of the playback fader ($00 - $FF)
	(**)	<>	Checksum

Stops the current fade in progress. Returns the current fader position.

In version 1.2 of Sidewinder, this function is not implemented yet.

Playback Set Position

Command:	<P(f)E(pp)(**)>		
Response:	<P(f)E(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(pp)	>	Position of the playback fader ($00 - $FF)
	(**)	<>	Checksum

Sets the playback fader position to pp. If a fade was in progress, it is halted.

In version 1.2 of Sidewinder, this function is not implemented yet.

Playback Resume

Command:	<P(f)F(**)>		
Response:	<P(f)F(**)>		
Parameters:	(f)	<>	Fader Number (0)
	(**)	<>	Checksum

Resumes a halted fade.

In version 1.2 of Sidewinder, this function is not implemented yet.

DMX Input Commands

The DMX Input commands are used to directly access DMXdata that is present at the DMXInput port of the Sidewinder interface. Sidewinder automatically merges any DMXInput data that is connected to the interface with the DMXOutput data that it is generating.

DMX Input Read

Command:	<I(ccc)(nnn)(**)>		
Response:	<I(ccc)(nnn)(vv..vv)(**)>		
Parameters:	(ccc)	<>	Starting Channel Number (zero based, $000 - $1FF)
	(nnn)	<>	Channel Count ($001 - $200)
	(vv..vv)	<	Channel Level String (each $00 - $FF)
	(**)	<>	Checksum

Returns the current values vv..vv from the input buffer of the nnn channels starting with channel ccc .

The **Starting Channel Number** is zero based, so valid values are from $000 (channel 1) to $1FF (channel 512).

The **Channel Count** value can range from $001 (one channel) to $200 (512 channels). The returned Channel Count value may be lower (or even zero) if the current DMX Input does not contain the requested channels or if no DMXInput is present.

The **Channel Level String** contains the returned Channel Count hexadecimal bytes, representing the values of each of the requested channels.

Resource Manager Commands

The Resource Manager is used to store, change and retireve data from Sidewinder's internal user-memory. In the current model of Sidewinder, the user-memory is Flash based, which allows for non-volatile storage while Sidewinder is turned off, but requires that the entire user-memory be erased before new (or changed) data can be recorded into it.

A Resource is a small block of data that is tagged with a resource type and resource ID in order for the data block to be easily organized and retrieved from memory when needed.

Sidewinder (and other applications) define several resource types specifically for use with Sidewinder's stand-alone playback functions. These resource types store cue data, macro data, show data and more. See the Resource Types section for information about the format of several pre-defined resource types.

Memory Erase

Command:	<ME(**)>				
Response:	<ME(rr)(**)>				
Parameters:	(rr)	<	Result		
			$00	=	Successful
			$7B	=	Memory Erase Failed
			$7F	=	Feature Not Available
	(**)	<>	Checksum		

Erases the internal non-volatile Flash memory. Although resource data can be added to the Flash memory at any time (until the memory space is full), no resource data can be changed once it is stored. To change resource data, the entire memory must be erased and re-loaded with updated information.

The **Result** returned will indicate if the operation was successful or if some sort of error occurred.

Memory Store

Command:	<MS(tt)(nnnn)(dd..dd)(**)>		
Response:	<MS(tt)(nnnn)(rr)(**)>		
Parameters:	(tt)	<>	Resource Type (two-character identifier)
	(nnnn)	<>	Resource ID ($0000 - $FFFF)
	(dd..dd)	>	Resource Data (hexadecimal bytes)
	(rr)	<	Result
			$00 = Successful
			$7A = Resource Manager Error
			$7C = Resource Already Exists
			$7E = Memory Full
			$7F = Feature Not Available
	(**)	<>	Checksum

Writes the specified resource data into non-volatile memory. Although resource data with any combination of type and ID may be stored into Sidewinder's internal memory, several specific types are used by Sidewinder (and other applications) to specifically store show data for stand-alone playback. See the Resource Types section for additional information.

The **Resource Type** is a two-character ASCII code that identifies the type of resource data that is being stored. For example, "CU" is used for cue data and "MA" is used for macro data.

The **Resource ID** is a hexadecimal number from $0000 to $FFFF that identifies the individual resource of the specified type. For example, cue data resources use the Resource ID to represent the individual cue numbers of each cue.

The **Resource Data** is a string of hexadecimal bytes that represent the data that is to be stored into non-volatile memory.

The **Result** indicates if the operation was successful or not. $00 indicates success and all others indicate a failure of some kind.

For example, to store a resource of type "EX", ID 1234 with the data $3A4B5C6D:

```
<MSEX12343A4B5C6D**>
```

And, if the command completed successfully, Sidewinder will return:

```
<MSEX123400**>
```

Memory Recall

Command:	<MR(tt)(nnnn)(**)>		
Response:	<MR(tt)(nnnn)(rr)(dd..dd)(**)>		
Parameters:	(tt)	<>	Resource Type (two-character identifier)
	(nnnn)	<>	Resource ID ($0000 - $FFFF)
	(rr)	<	Result
			$00 = Successful
			$7A = Resource Manager Error
			$7D = Resource Not Found
			$7E = Memory Full
			$7F = Feature Not Available
	(dd..dd)	<	Resource Data (hexadecimal bytes)
	(**)	<>	Checksum

Locates the specified resource data in the non-volatile memory and returns its contents. Although resource data with any combination of type and ID may be stored into Sidewinder's internal memory, several specific types are used by Sidewinder (and other applications) to specifically store show data for stand-alone playback. See the Resource Types section for additional information.

The **Resource Type** is a two-character ASCII code that identifies the type of resource data that is being stored. For example, "CU" is used for cue data and "MA" is used for macro data.

The **Resource ID** is a hexadecimal number from $0000 to $FFFF that identifies the individual resource of the specified type. For example, cue data resources use the Resource ID to represent the individual cue numbers of each cue.

The **Result** indicates if the operation was successful or not. $00 indicates success and all others indicate a failure of some kind.

The **Resource Data** is a string of hexadecimal bytes that represent the data that is recalled from non-volatile memory.

For example, to retireve a resource of type "EX", ID 1234:

<MREX1234**>

And, if the command completed successfully, Sidewinder will return (assuming the the data was stored from the Memory Store example:

<MREX1234003A4B5C6D**>

Memory Play Macro

```
Command:     <MP(nnnn)(**)>
Response:    <MP(nnnn)(rr)(**)>
Parameters:  (nnnn)      <>  Macro Number ($0000 - $00FF)
             (rr)        <   Result
                                 $00  =  Successful
                                 $7A  =  Resource Manager Error
                                 $7D  =  Resource Not Found
                                 $7E  =  Memory Full
                                 $7F  =  Feature Not Available
             (**)        <>  Checksum
```

Locates the Macro resource (resource type "MA") with the specified ID and executes it.

The **Macro Number** is the number of the macro resource to locate and execute.

The **Result** indicates if the operation was successful or not. $00 indicates success and all others indicate a failure of some kind.

Memory Play Macro Vector

```
Command:     <MV(vvvv)(**)>
Response:    <MV(vvvv)(rr)(**)>
Parameters:  (nnnn)      <>  Macro Vector Number ($0000 - $0008)
                                 $0000   =   Start-Up Macro
                                 $000x   =   Physical Button Macro
             (rr)        <   Result
                                 $00     =   Successful
                                 $7A     =   Resource Manager Error
                                 $7D     =   Resource Not Found
                                 $7E     =   Memory Full
                                 $7F     =   Feature Not Available
             (**)        <>  Checksum
```

Locates the Show Information resource (resource type "SI", ID 0000) and fetches the macro number that the show specifies should be executed when one of the physical macro buttons is pressed or when the Sidewinder starts-up. When this macro number is acquired, then the command locates and executes the macro.

The **Macro Vector Number** is the number of the macro vector to locate the actual macro number to execute (see the table above).

The **Result** indicates if the operation was successful or not. $00 indicates success and all others indicate a failure of some kind.

Resource Types

Sidewinder defines several resource types for use in conjunction with its Stand-Alone Playback feature and for use with several other external applications (like Figment DMX for Palm OS).

This section describes several resource types that are pre-defined.

Please note that user applications can store any type of data in Sidewinder, but Interactive Technologies reserves the use of all resource types that are composed of two capital letters from AA to ZZ. Any other combination of two characters with either lower-case letters, numbers or punctuation marks are available for use in your own applications.

Show Information Resource

Resource Type:	SI	
Resource ID:	0000	
Format:	(nn..nn)(ffffffff)(m0)(m1)(m2)(m3)(m4)(dddddddd)	
Fields:	(nn..nn)	Show Name (16 bytes)
	(ffffffff)	Flags (4 bytes)
	(m0)	Start-Up Macro Vector (1 byte)
	(m1)	Macro 1 Vector (1 byte)
	(m2)	Macro 2 Vector (1 byte)
	(m3)	Macro 3 Vector (1 byte)
	(m4)	Macro 4 Vector (1 byte)
	(dddddddd)	Modification Date (4 bytes)

Show Name is the name of the show file. Each character of the name is converted to its hexadecimal representation in the string to send to the Memory Store command.

Flags contain individual flag bits for the show file.

Start-Up Macro Vector contains the macro number to execute when Sidewinder starts-up.

Macro x Vector contains the macro numbers to execute when one of the physical macro buttons are pressed.

Modification Date is the date and time that the show was last modified. This number is represented in the number of seconds since January 1, 1904.

Cue Resource

Resource Type:	CU	
Resource ID:	from 0001 to 270F , representing the cue number	
Format:	(nn..nn)(00)(ff)(iiii)(tttt)(cccc)	
	(aaaa)(rrrrrr)(mm..mm)(vv..vv)	
Fields:	(nn..nn)	Cue Name (16 bytes)
	(00)	Reserved, Must Be Zero (1 byte)
	(ff)	Flags (1 byte)
	(iiii)	Cue Number (2 bytes)
	(tttt)	Fade Time (2 bytes)
	(cccc)	Follow Cue (2 bytes)
	(aaaa)	Follow Time (2 bytes)
	(rrrrrr)	Reserved, Must Be Zero (3 bytes)
	(mm..mm)	Channel Mask (64 bytes)
	(vv..vv)	Channel Levels (variable length, 512 bytes max)

Cue Name is the name of the cue. Each character of the name is converted to its hexadecimal representation in the string to send to the Memory Store command.

Flags contain individual flag bits for the cue.

Cue Number is the cue number, from $0001 to $270F.

Fade Time is the fade time for the cue, represented in tenths of a second, where $0000 is immediate.

Follow Cue is the cue number of the cue to follow to after this cue is complete.

Follow Time is the follow time for the cue, represented in tenths of a second, where $0000 is immediate.

Channel Mask is a 64-byte bit field, where the most significant bit of the first byte represents channel 1 and the least significant bit of the last byte represents channel 512. A 1 in any channel's corresponding position indicates that the channel is included in the cue.

Channel Levels is a string of hexadecimal bytes, one for each channel indicated in the Channel Mask. Only channels with a 1 in their corresponding bit in the Channel Mask are included in the string of Channel Levels.

Macro Resource

Resource Type:	MA	
Resource ID:	from 0001 to 00FF , representing the macro number	
Format:	(nn..nn)(0000)(iiii)(cccc)(rr..rr)(kk..kk)	
Fields:	(nn..nn)	Macro Name (16 bytes)
	(0000)	Reserved, Must Be Zero (2 bytes)
	(iiii)	Macro Number (2 bytes)
	(cccc)	Key Count (2 bytes)
	(rr..rr)	Reserved, Must Be Zero (6 bytes)
	(kk..kk)	Keys (variable length, 64 bytes max)

Macro Name is the name of the macro. Each character of the name is converted to its hexadecimal representation in the string to send to the Memory Store command.

Macro Number is the macro number, from $0001 to $00FF.

Key Count is the number of keys in the macro definition (maximum 64).

Keys is an array of the hexadecimal values of ASCIIcharacters for each of the macro's keys (maximum of 64 bytes).