



## **Showflow**

### **Integrated Show Control and Playback System**

MediaMation Incorporated  
2461 West 205th. Street, Ste: B100  
Torrance, Ca, 90501  
Phone:(310) 320-0696 Fax: (310) 320-0699

<b>WHATS NEW!</b> .....	<b>4</b>
<b>CHANGES FOR VERSION 3.XX</b> .....	<b>4</b>
COMPILING .....	4
GUI.....	4
INCLUDE FUNCTION.....	4
TIMING.....	5
LASERDISK/DVD IMPROVEMENTS .....	5
IF AND TRAP STATEMENTS.....	5
<b>NEW COMMANDS</b> .....	<b>5</b>
TIME.....	5
DATE.....	5
DAC FILTER.....	5
KEY CODE .....	5
ONE.....	5
ZERO.....	5
UNSCHEDULE.....	6
MUL.....	6
SIN .....	6
TIMECODE.....	6
ANALOG OUTPUT.....	6
CD/DVD .....	6
MOOG / MOOGRS .....	6
OPTOMUX .....	6
PARALLEL.....	6
FILE.....	6
BINARY.....	6
ESCAPE CHARACTERS.....	7
MIDI SHOW CONTROL .....	7
MIDI MACHINE CONTROL.....	7
ICVERIFY .....	7
SHIFT4.....	7
START/STOP DEVICES.....	7
MULTIPLE LISTEN.....	7
MIDI / DAC.....	7
PORT.....	7
USER CONST.....	7
DISPLAY .....	7
<b>INTRODUCTION</b> .....	<b>9</b>
<b>CONCEPT</b> .....	<b>10</b>
<b>TECHNICAL SPECIFICATIONS</b> .....	<b>10</b>
<b>BASIC KEYBOARD COMMANDS</b> .....	<b>10</b>
<b>RELATED FILES</b> .....	<b>11</b>
<b>UPPER AND LOWER CASE</b> .....	<b>13</b>
<b>GRAPHICAL USER INTERFACE SPECIFICATIONS</b> .....	<b>13</b>
<b>GUI PRIMITIVES (OBJECTS)</b> .....	<b>14</b>
TEXT NAME (COLOR, "STRING", "FONT (OPTIONAL)", X, Y).....	14

RECT NAME (COLOR, PEN WIDTH, RADIUS, X, Y, W, H).....	14
BORDER NAME (COLOR, PEN WIDTH, X, Y, W, H).....	15
LED NAME (COLOR, TYPE, , X, Y, W, H).....	15
BUTTON NAME (COLOR, "TEXT", "FONT (OPTIONAL)",TYPE, , X, Y, W, H).....	15
HOTSPOT NAME (X,Y,W,H).....	15
MBOX NAME (X,Y,W).....	15
LINE NAME (COLOR, PEN WIDTH,X,Y,X2,Y2).....	16
POLYGON NAME (COLOR, PEN WIDTH, X, Y, X2, Y2,.....XN,YN).....	16
OVAL NAME (COLOR, PEN WIDTH, X, Y, W, H).....	16
IMAGE NAME ("FILE.PCX", X,Y,W,H).....	16
.GUI AND WINDOWS SUMMARY .....	16
<b>SHOWFLOW LANGUAGE SPECIFICATIONS .....</b>	<b>18</b>
VARIABLES .....	18
<i>Output Variables</i> .....	19
DIGITAL OUTPUTS.....	19
ANALOG OUTPUTS.....	20
MIDI OUTPUTS.....	20
<i>Input Variables</i> .....	21
DIGITAL INPUTS.....	21
ANALOG INPUTS.....	22
MIDI INPUTS.....	22
USER VARIABLES.....	22
GROUP VARIABLES .....	23
TIME CODE .....	23
<b>DEVICES .....</b>	<b>23</b>
PIONEER LASER DISK PLAYERS.....	24
MOOG ELECTRIC MOTION BASE.....	24
PIONEER FILM PROJECTOR CONTROLLER .....	25
SINGLE TRACK MIDI SEQUENCERS .....	26
SEQUENCER MULTITRACK.....	26
GENERIC SERIAL DEVICES.....	27
<b>DVAR'S.....</b>	<b>27</b>
PIONEER LASER DISK PLAYERS.....	28
MOOG ELECTRIC MOTION BASE.....	28
SEQUENCER DVARs.....	28
GENERIC DEVICES .....	28
<b>NODE STATEMENTS.....</b>	<b>29</b>
SUBROUTINES.....	29
TRAPS.....	30
NODE PARAMETERS.....	32
FUNCTIONS.....	33
NULL().....	33
COLOR_LED( LEDNAME, COLOR).....	33
BLINK_LED( LEDNAME, COLOR ).....	33
COLOR_BUTTON( BUTTONNAME, COLOR).....	33
BLINK_BUTTON( BUTTONNAME, COLOR ).....	34
TRAP_BUTTON ( BUTTONNAME) NODENAME.....	34
UNTRAP_BUTTON( BUTTONNAME ).....	34
MSG( MSGNAME, STRING ) and LOGMSG( MSGNAME, STRING ).....	34
SPIT( MSGNAME, VARNAME ) and LOGSPIT( MSGNAME, VARNAME ).....	35
WINDOW( WINDOWNAME ).....	35
CALL( NODENAME ).....	35
WAIT( NUMBER ).....	35

<i>SCHEDULE( NUMBER, NODENAME)</i> .....	36
<i>FLUSH( )</i> .....	36
<i>EXIT( )</i> .....	36
<i>RESTART( )</i> .....	37
<i>TRAP( VARNAME, BOOLEAN FUNCTION) NODENAME</i> .....	37
<i>UNTRAP( VARNAME )</i> .....	38
<i>RESUME( )</i> .....	38
<i>CLEAR( VARNAME )</i> .....	39
<i>SET( VARNAME [EXPRESSION] )</i> .....	39
<i>INC( VARNAME [EXPRESSION] ) and DEC( VARNAME [EXPRESSION] )</i> .....	39
<i>BLINK_VAR( VARNAME )</i> .....	39
<i>LOAD( FILENAME )</i> .....	39
<i>HOME( )</i> .....	39
<i>SYNC(option)</i> .....	39
<i>MOUSE( BOOLEAN )</i> .....	40
<i>RAMP( BOOLEAN )</i> .....	40
<i>KINEMATICS( BOOLEAN )</i> .....	40
<i>START_SHOW( )</i> .....	40
<i>STOP_SHOW( )</i> .....	40
<i>MIDI_OUT( MIDISTREAM )</i> .....	40
<i>SERIAL_OUT( PORT, STRING )</i> .....	40
<i>SLIDE ( BYTE1, BYTE2, VALUE1, VALUE2, TIME )</i> .....	41
<i>Branch Nodes (IF (&lt;expression&gt;) NODE)</i> .....	41
<i>CHANNELIZE</i> .....	41
SEQUENCES.....	42
<i>SEQUENCER SINGLETRACK NAME</i> .....	42
<i>SEQUENCER MULTITRACK NAME</i> .....	42
DEVICES.....	42
<i>PLAY(NAME)</i> .....	43
<i>LOAD(NAME, FILE)</i> .....	43
<i>STOP(NAME)</i> .....	43
<i>CMD(NAME, ENGAGE)</i> .....	43
<i>CMD(NAME, DEVSTOP)</i> .....	43
<i>CMD(NAME, RUN)</i> .....	43
<i>CMD(NAME, FILE, XXX)</i> .....	43
<i>CMD(NAME, INHIBIT)</i> .....	43
<i>CMD(NAME, RESET)</i> .....	43
<i>CMD(NAME, DISABLE)</i> .....	43
<i>CMD(NAME, ESTOP)</i> .....	43
<i>CMD(NAME, STATUS)</i> .....	43
<i>CMD(NAME, PARK)</i> .....	43
<i>CMD(NAME, ASCII, "XXX")</i> .....	44
<i>CMD(NAME, STATUSON)OFF</i> .....	44
<i>CMD(NAME, STATUSON)</i> .....	44
<i>CMD(NAME, STATUSOFF)</i> .....	44
<i>CMD(NAME , QUERY)</i> .....	44
<i>start(NAME)</i> .....	44
<i>stop(NAME)</i> .....	44
<i>seek(NAME, 00:00:00:00)</i> .....	44
<b>CONCLUSION</b> .....	<b>45</b>

# SHOWFLOW VER 3

## Whats New!

ShowFlow version 3 is finally ready to go! We thank you for your purchase and/or upgrade. I'm sure that these changes will enhance your speed and ability to create custom applications for your show, ride, or exhibit. Some of the new features are simply upgrades that will improve the operability of the software in general and require no changes. Others are major enhancements. Also, the graphic editor for ShowFlow, called ShowCase, that replaces the need to write text is now shipping.

## Changes for Version 3.xx

If you are a current ShowFlow user and simply want to know what the new features are, we have included this section to allow you to quickly start using the new features. If you are new to ShowFlow, you may want to skip this section until you have a better understanding of the program and its functions.

### ***Compiling***

ShowFlow now has a separate compiler called SHOWBLD.EXE. This allows you to build or "compile" your script and turn it into a file that the program can load and run much faster than previous versions. Besides being much faster on bootup, this new system allows you to remove the SHOWBLD application and thus prevent anyone from making any changes to the program!

### ***GUI***

ShowFlow version 3.xx no longer automatically loads your .GUI file. In fact, you no longer need to have a separate GUI file! (See new INCLUDE function below). You can have one if you want to (a good suggestion), but you can also simply have your GUI descriptions as part of your main script. Additionally, there have been a lot of improvements to the GUI functions. You can now define the width (pen size) and color of lines, rectangles, rounded rectangles, and any other GUI object. You can also now define polygons of up to 22 interconnected lines. You can make LED's and buttons in several different shapes, blink ANY object on the screen (except imported images), and hide or display items at will from the program! Lastly, the new graphics drivers support full 32 bit colors, larger screen resolutions, and a much wider variety of SVGA driver chips which gives us graphics support on most PC's.

### ***INCLUDE function***

A very useful new feature is the "include" statement. It allows your ShowFlow script to include other scripts as part of your main script. You can now break up large scripts into several files. This can make troubleshooting and editing a lot easier. You can also include separate GUI files, or any other files (as long as they are actually compatible text files). This allows you to create libraries of sub-routines, user windows, etc. that can be called from different scripts.

## ***TIMING***

There have been several upgrades to the ShowFlow timing routines. First and foremost, all timed functions such as "wait", "schedule", and "slide" use milliseconds as their timing argument. Other timing improvements are internal and require no change. The execution speed of the program is now about 250% faster! Also analog inputs update 2X faster to keep them compatible with the HRBox features.

## ***LaserDisk/DVD Improvements***

The integration with Pioneer LaserDisk players and the new DVD 7400 series has been improved. You can now issue any non-supported commands directly to the player without interfering with the timing of the automatic status commands etc. Also, you can tell it to play to a stop marker point with an argument to the play command.

## ***IF and TRAP statements***

There is a syntax change in "if" and "trap" commands. The branch node name **MUST** be outside the parenthesis now. In previous versions, it could be inside or outside. This inconsistency has been changed. You can also perform multiple traps on a single variable, and individually "untrap" these traps. This allows you to monitor a single variable, and break to one of several different routines based upon the value of that single variable.

## **New Commands**

There are several new commands that have been added. For detailed explanations, see the following manual. Below is a quick reference to the new commands and their functions:

### ***TIME***

Allows you to monitor and trap on the system's internal time clock.

### ***DATE***

Allows you to monitor and trap on the system's internal calendar.

### ***DAC Filter***

Allows you to modify and monitor the global DAC filter scaler under program control

### ***Key Code***

Allows you to monitor and trap the keys on the computer keyboard and use them within your program for various functions.

### ***one***

Allows you to use the word "one" to mean the number 1 or on.

### ***zero***

Allows you to use the word "zero" to mean the number 0 or off.

### ***unschedule***

Allows you to unschedule, or flush, a single scheduled process without interfering with other currently scheduled processes.

### ***mul***

Adds the ability to multiply a variable by another variable or number.

### ***sin***

Adds the ability to calculate the angular sine value of a number or variable.

### ***timecode***

Allows you to monitor incoming timecode without having to synchronize to it. Multiple timecodes can be read from multiple Midi ports. Also allows you to define variables in SMPTE format (hrs:min:sec:fr) and have them display in the variable windows as such.

### ***analog output***

Allows you to define your built in, Moog, or Midi DAC analog outputs as variables, thus allowing you to monitor and set them under program control

### ***cd/dvd***

Adds full support for standard ATAPI CD ROM's and DVD ROM's installed in the system. Includes search by track, SMPTE time (created automatically from the timing bits), and ability to synchronize Midi Sequences, motion files, and Cuelists to these devices for playback.

### ***moog / moogrs***

Adds support for both the Moog Electric Motion Bases with ride storage and those with direct control. A full set of bi-directional communications and status feedback are supported for both models.

### ***optomux***

Adds full support for the Opto 22 "optomux" series of RS485 networked Solid State Relay (SSR) boards and interfaces.

### ***parallel***

Allows you to send data to the various parallel ports on the system for communications and printing.

### ***file***

Allows you to open, modify, and save text files on the systems storage medium or hard drive.

### ***BINARY***

Allows you to send, receive, and monitor (LISTEN) in binary format for any serial device that requires it.

### ***Escape characters***

You can now use what are known as “escape characters” to embed characters such as “line feed”, “return”, or even Hex numbers within standard ASCII strings when communicating via the serial or parallel ports.

### ***midi show control***

Adds support for the Midi Show Control specification. You can now define a MSC device and then control it with standard MSC commands such as “cue” and “stop”.

### ***midi machine control***

Adds support for the Midi Machine Control specification. You can now define a MMC device and then control it with standard MMC commands such as “play” and “stop”.

### ***ICVerify***

Adds support for the popular IC Verify credit card processing system.

### ***Shift4***

Adds support for the popular Shift 4 / Dollars in the Bank credit card processing system.

### ***start/stop devices***

Changes the start and stop show commands to start and stop all devices. Any device in the system that supports the play and stop command will respond to this command simultaneously.

### ***multiple LISTEN***

Adds the ability to “listen” for multiple responses simultaneously on a serial port that has been defined as a generic serial device.

### ***MIDI / DAC***

The HRBox engine can now define analog outputs as virtual outputs that generate Midi commands rather than have physical analog output cards built into the system. This allows you to have all the benefits (setable variables, filters, graphic sliders, span, offsets, etc.) of built in analog outputs, but utilize external Midi devices such as our AO-16 for more cost effective and/or flexible control options.

### ***port***

Allows you to instruct a Midi sequence to play out of a particular Midi port or cable.

### ***user const***

Allows you to define user constants that are available to the program, but cannot be changed nor do they display on the Variables screen. This is handy for things such as creating a variable called “Noon” and setting it as a user constant to 12:00:00:00. You can then simply refer to Noon rather than the exact time each time you need it.

### ***display***

Allows you to “hide” or display ANY graphic object under program control.



## Introduction

Most shows involve more than a simple sequence playing back. Decisions about which show, when to play it, monitoring of safety inputs, interactive requirements, and PLC type logic need to be made to fully integrate a show and bring all the control, audio, video, and operator interface together. Until the advent of ShowFlow, these various items were accomplished by using several different devices from different manufacturers, a small army of programmers and installers, and a large checkbook! ShowFlow brings a single, easy to program, totally integrated solution to any situation where more than simple playback is required.

The ShowFlow program acts as a “logic shell” that wraps around our HRBox controller with no additional hardware required. Utilizing the direct analog, digital, serial, and Midi I/O functions of the HRBox, ShowFlow has complete access to all your external devices, sensors, and controls. Imagine combining your Show Control, Show Monitor, and Operator Interface into a single unit! ShowFlow allows you to quickly and easily program these I/O's to operate and interact in any way you want. The SVGA video output, with a friendly graphical user interface, allows you to monitor the precise state of your entire show. In addition, you have the option of overlaying this data with custom touch-screen "Control Panels" of your own, interfacing with real consoles, or any combination of the two, for easy use by even an inexperienced operator.

The element that ties everything together is the language of ShowFlow, a language developed especially for handling the demands of a complex show control system. Imagine drawing a flowchart indicating how you would like the various components to operate. In the past, the next step would be to have a technician create the hardware, or program the appropriate circuitry, to properly control all the components. With ShowFlow, you're practically done. The language is designed to operate like the flowchart! With a simple vocabulary of a few words or commands, you can create the most sophisticated logic circuitry.

Unlike other programming languages, ShowFlow was designed from the ground up for Show Control. Here are just a few of its unique features:

- Simultaneously monitor hundreds of I/O points in real time;
- Complete support for SMPTE timecode;
- Play back any number of prerecorded shows with their own instructions to all I/O devices;
- Direct support for machines such as DVD Players, Film Projectors, Moog Motion bases, and more;
- Access to all of HRBox functions including our exclusive Universal Kinematics, Analog filtering, and new PID servo loop closure with programmable compliance;
- Import graphics and create “Hot Spots” on the screens for intuitive interactive interfaces;

All these components combine to make ShowFlow the most powerful tool available in its field. This easy programming interface also allows you to incorporate complete interactivity with your show, ride, or exhibit, while keeping things running exactly the way you want them to. Coupled with the built in filters, safety ramping, and flexibility of our HRBox, you can drive virtually any show of any size from one compact, rack mount unit.

ShowFlow provides everything you need in a show control system: Synchronized operation of different devices, an easy interface for your finished product, the flexibility to improve your system in minutes

instead of days, and a simple, intuitive environment for creating the brains of your show. With ShowFlow running things for you, you know that rampaging dinosaurs, motion bases, crowded pitch-black halls, and erupting volcanoes are all doing, effortlessly, exactly what you want them to do.

## Concept

The idea behind the ShowFlow language and environment is to offer the ability to quickly develop and implement the sophisticated logic, sequence playback, operator interface, safety monitoring, and show control systems necessary for the proper running of all the hardware components of a show.

The language is designed to flow linearly through the script (program) with branches and interrupts redirecting the flow to other routines as needed. The use of cuellists allows you to have several multitasking “flows” or paths running independently and simultaneously. Typically, the script is sitting in a short loop waiting for something to happen. This happens when an interrupt is triggered by some external event such as a button push or incoming Midi or serial data triggers a routine to run.

Interrupts are generated in ShowFlow via our extremely powerful “trap” function. Unlike most other programming languages that require you to program endless “if” routines that you have to keep track of, the “trap” function allows you to set the trap to trigger on a specific event, and then forget about it until it happens. You can have hundreds or thousands of traps simultaneously waiting to run with virtually no processing time utilized to monitor them. If you plan your script carefully, you will find this to be a very simple method to program and de-bug.

## Technical Specifications

The ShowFlow system is built around an 80xxx , PC compatible microprocessor running a high-speed specialized multitasking kernel. The application it runs consists of the ShowFlow runtime environment, two data files, and various system and support files that are included with your program disks. The program requires a minimum 486DX processor with a minimum of 8 MB of RAM. Currently, the environment operates from MS-DOS 6.2 or higher and occupies approximately 2 MB of hard disk or storage space.

The preemptive multitasking kernel executes a continual processing loop that executes a ShowFlow logic program. The processing loop also services all hardware attached to the system that it can recognize, such as asynchronous serial data, MIDI, raw digital, raw analog, a mouse or touch screen, and a graphic user interface and monitoring system.

The system loop is driven by an internal millisecond timer, allowing periodic events and external devices to be scheduled with millisecond precision. Analog voltages are sent out once per timer tick to any devices that receive voltage as an input, and digital i/o takes place once every 30 milliseconds to help eliminate “bouncing”.

## Basic Keyboard Commands

Certain keys will bring up certain screens to monitor activities and to aid in troubleshooting. These screens are AUTOMATICALLY GENERATED (!) in ShowFlow for you. These screens are invaluable for troubleshooting and testing. You can click on the Digital Output boxes and the Digital Input boxes with a mouse to toggle them On/Off . You can also click and drag on the analog sliders in the Analog Output screen to manipulate them in real time.

- U** returns to the main operator screen.
- A** Brings up the display of analog ins and outs, also shows if kinematics are engaged. Each press of the “A” key will proceed to the next screen. Depending on how many analog I/O’s you have, there may be several screens. A minimum of 2 screens (input and output) are always created. If no analogs are part of your system, these screens will be empty. Mouse draggable output sliders allow manual setting of the outputs for testing and debugging.
- O** Displays the declared digital outputs, their current status, and are mouse clickable to toggle their state. Depending on how many Digital outputs you have, there may be several screens. Each press of the “O” key will proceed to the next screen.
- I** Displays the declared digital inputs, their current status, and are mouse clickable to toggle their state. Depending on how many Digital inputs you have, there may be several screens. Each press of the “I” key will proceed to the next screen.
- S** This is the system variables screen. It displays the current file, node name and line number of that file that the program is executing. It also displays the number of subroutines that are waiting to finish, the number of trapped routines that are waiting to run, and whether the ability to run trapped routines has been re-activated. It displays this information for the main script and any cuelists that are defined. Underneath these displays, it also displays the activity of any devices you have declared, and how many future scheduled routines you currently are waiting to occur.
- V** Successively brings up screens showing any “non-physical” variable. These are any variables that do not have a physical input or output card in the system. They include all user variables, system variables such as DATE and TIME, any declared Midi I/O’s , Automatically generated Digital Variables (DVars) from any declared devices, and other variables not included on other screens.
- K** This will activate or deactivate the Kinematics.
- M** Causes the mouse cursor arrow to disappear and appear.
- Q** To quit Showflow

## Related Files

The ShowFlow language, while designed to be a visual language, is stored as a simple text file, <name>.SFL, which can be viewed, understood and edited from any text editor. We refer to these text files as the ShowFlow “*scripts*” as they describe the functionality of your application. Throughout this manual we will be referring to these scripts and to the program (ShowFlow) as distinct items. You may break your main script into as many smaller files as you want and use the *include* command to include them in the main script when the program compiles. Typically, you may want to break out your GUI files, Cuelists, and other routines and files into separate files. This can make the process of organizing your project easier and helps with troubleshooting.

The log of the system's progress is contained in an automatically generated text file called 'RECORD.TXT' which is updated by the program everytime it runs. Additional information, such as what shows were played, can be written to this file under program control. Each entry is automatically tagged with the current date and time that it occurred.

There is an encrypted file, entitled 'MMATION.CFG', that contains the hardware specifications for the system. This file **MUST** be located in the directory that contains your compiled script files (.SFO and

.SYM files). This file can only be modified by MediaMation. Attempts to change it yourself will have unpredictable results.

There are 2 optional text files called 'HRBOX.CFG' and KINEMATE.CFG. HRBOX.CFG outlines all the parameters used by HRBOX, the core digital signal processing application, embedded in ShowFlow. It is a user editable text file that loads default information for all the user controllable parameters. These include, analog input mapping, PID servo loop settings, Digital I/O note assignment, spans and offsets for every analog I/O, filter settings for every analog I/O, default SMPTE start and stop times, and more. KINEMATE.CFG is a user editable text file that contains all the information for the settings of your Universal Kinematics (See HRBox manual for more details) for all your analog I/O's. These files must also be located in the directory that contains your compiled script files.

There are 2 font files called SMALL.FNT and LARGE.FNT that must also be located in the directory that contains your compiled script files.

It is possible to have multiple programs residing on a single system. ShowFlow attempts to find and run a show file called 'SHOW.SFL', but these can be overridden at run time by giving the name of a different compiled script to ShowFlow at execution time. If you are in MS-DOS, follow the word 'SHOWFLOW' at the command prompt with the name of the desired show i.e. 'SHOWFLOW ALIENS4' will force ShowFlow to look for 'ALIENS4.SYM' and "'ALIENS4.SFO"' (the 2 files generated by the compiler) and use them in place of the defaults. It should be noted that, while different programs may reside on a single system, they will all write their progress to the same log, 'RECORD.TXT'.

Lastly, there are many different system files and .DLL files that the system requires to operate. These are all included with your software when purchased and need to be in the same directory as the main ShowFlow and ShowBld programs in order to operate.

**In Summary:**

Your ShowFlow program requires several files in order to compile and run. In general those file swill appear on your disk as:

- SHOWFLOW.EXE            *required*
- SHOWBLD.EXE            *required in order to compile scripts*
- SHOW.SFL                *or any <name>.SFL. This is the **uncompiled** script*
- SHOW.SFO                *or any <name>.SFO. This is the **compiled** script file #1*
- SHOW.SYM                *or any <name>.SYM. This is the **compiled** script file #2*
- MMATION.CFG            *required*
- HRBOX.CFG               *optional but reccommended*
- KINEMATE.CFG           *optional and only required if you have analog I/O and you want Kinematics*
- RECORD.TXT              *Automatically generated the first time the program is run*

**SYSTEM FILES REQUIRED:**

- |                              |                     |                     |
|------------------------------|---------------------|---------------------|
| <i>cw3215mt.dll</i>          | <i>cw3230mt.dll</i> | <i>hasp.dll</i>     |
| <i>hrbox.dll</i>             | <i>large.fnt</i>    | <i>metwnd.dll</i>   |
| <i>metwnd05.dll</i>          | <i>metwnd05.drv</i> | <i>metwnd07.dll</i> |
| <i>metwnd07.drv</i>          | <i>metwnd08.dll</i> | <i>metwnd08.drv</i> |
| <i>metwnd09.dll</i>          | <i>metwnd09.drv</i> | <i>mmation.exe</i>  |
| <i>mpu.dll or altech.dll</i> | <i>small.fnt</i>    | <i>textmode.dll</i> |
| <i>ui.dll</i>                | <i>winmm.dll</i>    |                     |

## Upper and Lower Case

ShowFlow *is* case sensitive. This means that “BLINK” is not the same as “blink” or “Blink”. They are all 3 different words as far as ShowFlow is concerned. **All** commands in ShowFlow are lower case. **All** GUI items are uppercase. The reason for all this is that ShowFlow reserves all words that are part of its command set and you can't use them elsewhere. By allowing you to have Upper and Lower case sensitivity, you have a lot more options as you name things. Note that text strings in quotes are treated as a single string so are somewhat exempt from this requirement. For example, you can't call an object *blink* as that is reserved. However, you can call something “*blink*” if you wanted to. You can also use the “\_” to create compound words that are treated separately, such as “blink\_Light”.

## Graphical User Interface Specifications

The GUI file associated with a given show is simply a text-based description of a series of 'Windows' that the user will view. Although it is probably the most complicated syntax portion of creating a ShowFlow script, the ShowCase Editor creates these files for you automatically utilizing a simple drag and drop interface. You can create as many different windows, that the script calls up at various times, as you wish. This is a powerful feature of ShowFlow as it can eliminate thousands of dollars worth of buttons, cables, wiring, etc. Each screen can have text, message boxes, buttons, graphics, LED's, and drawing objects to customize it's usefulness and purpose. If no windows are described, ShowFlow will default to displaying the program's variable screen.

Even though we very seldom need to concern ourselves with the actual text and syntax that are used to create a ShowFlow GUI screen, we will go over the syntax here. Whenever the SHOWBLD compiler encounters a GUI description in your script or within an included file, it generates a new screen as instructed. You can embed these files anywhere within your script that you want to, but it is recommended that you put them in their own file and include it within the main script.

Within the file itself, the syntax for a window description is:

```
gui <windowname> (x, y, width, height) {
    item(parameters)
    item(parameters)    ...
}
```

*This is the only time that the { and } symbol are utilized in the program and is probably the most “C Programming” like syntax you will be required to use. Don't panic. Just use an existing window to copy and paste to create a new window description when you need to.*

<windowname> can be any name up to 32 characters in length, as long as it starts with a letter from the alphabet. X and Y are the distance, in pixels, of the top-left corner of the window from the top-left corner of the screen, which is 640 pixels wide by 480 pixels tall. Width and height are how many pixels wide and long the window will be.

item(parameter) lines describe the various graphic objects contained in your window.

Here is an example of a window called “Main” that covers the entire 640 X 480 pixel screen, with some text displayed in it, and one button called STOP (see below for instructions on text and buttons):

```
gui main (0,0,639, 479) {
TEXT Text1 (32767, "This is some Text", "", 107, 43)           // displays some text in the window
```

```
BUTTON Stop(21140, "Stop", "", RECTANGLE, 438, 395, 82, 72) // Creates Button called EStop
}
```

Note that the upper most left corner is pixel 0,0 and that 639,479 is lowermost right corner. When counting from 0, this makes a TOTAL of 640 X 480 pixels. Also note that the double bar // indicates to ShowFlow that any text after it is ignored or “commented out” until a <RETURN> is found. This makes it very easy to add comments and information about what is happening at a particular point to your scripts. It also allows you to make portions of the script “invisible” to ShowFlow without actually cutting them and losing them for testing and debugging. You can use spaces and tabs at will as ShowFlow ignores these.

You can specify as many windows as you choose, and as many items within a window as you want. The ideal method of editing this resource file is the graphical editor, but outlined below are the different graphical primitives, or basic objects, recognized. Note that ShowFlow is case sensitive and you must be careful to use the proper upper and lower case letters when creating your ShowFlow and GUI scripts.

## GUI Primitives (Objects)

The graphic objects each need a series of parameters in order to be properly created. The most common parameters are listed her:

X	Distance from the left edge of the window
Y	Distance from the top edge of the window
W	Width in pixels of object. Distance from X
H	Height in pixels of object. Distance from Y
COLOR	Given as a color number from 0 to 32768
NAME	An object's unique name allowing ShowFlow to interact with it during runtime.
PEN WIDTH	The point size of the outside border or pen size of an object

### **TEXT Name (Color, “String”, “Font (optional)”, X, Y)**

A word or a line of text, unchangeable. Good for identifying other graphic objects. STRING is a message, enclosed in quotes, such as "Show Status".

Example:

```
TEXT MyText (32768, “This is it!”, ““, 90,90) // Puts some white text in a window at pixels 90,90
```

### **RECT Name (Color, Pen Width, Radius, X, Y, W, H)**

A rectangle object.

Pen Width is the point size of the outer border. A value of -1 will have no outer border and will instead be filled with the color.

Radius is the number of pixels of curvature that the object will have. A value of 0 will create perfectly straight edges. A value equal to the width will make a smooth arc from one side to the other.

X and Y are the coordinates of the top-left corner, while W and H are the width and height of the object.

Example:

```
RECT MyRect (32768, -1, 0, 90, 90, 45, 25) // Puts a small white rectangle at pixels 90,90.
```

### ***BORDER Name (Color, Pen Width, X, Y, W, H)***

BORDER is a stylized version of the RECT. A Border draws an empty rectangle with a border 4 pixels thick, with highlighting to appear mock-3D. It's a good idea to use one around the edge of any windows you create.

Example

**BORDER border3 (21140, 1, 90, 90, 100, 120) // makes a 3D border of pen size 1 at pixels 90,90**

### ***LED Name (Color, Type, , X, Y, W, H)***

The LED is a very versatile graphic object. LEDs function as lights on your interface. Each LED needs a unique name for referencing by ShowFlow. An LED can be colored by the program, or set to blink between two colors. There is no pen width as all LED's are completely filled with the color.

Type can be RECTANGLE, RADIUS, or ROUND. This allows you to have either a sharp edged, slightly rounded edge, or completely round edged LED

Example:

**LED ROUNDED (29, ROUND, 254, 117, 80, 71) // Creates an on screen LED called Rounded**

### ***BUTTON Name (Color, "Text", "Font (optional)", Type, , X, Y, W, H)***

BUTTON is the most versatile graphic object, and the only one that can be used for user input as well as graphical output. It has the added functionality of being able to respond to a user's pressing it via mouse or touchscreen.. It is similar to an LED except it can also be used as a pushbutton, and can also have a text label embedded in it. You can use Buttons as LED's within your program with the added benefit of having a text label displayed.

Text is the label that is centered within the button object.

Font is an optional argument to change the font style. This requires the listed font to be present within the same directory as the script file.

Type can be RECTANGLE, RADIUS, or ROUND. This allows you to have either a sharp edged, slightly rounded edge, or completely round edged Button

Example:

**BUTTON RadButton2 (1023, "Button2", "", RADIUS, 496, 126, 87, 60) //creates a button labeled Button2**

### ***HOTSPOT Name (X,Y,W,H)***

HOTSPOT is an invisible button. It has the functionality of being able to respond to a user's pressing it via mouse or touchscreen, but a graphic can be behind it without being obscured from view. This allows you to create sophisticated graphics in your desktop publishing program, place them in a ShowFlow window, and make them respond just like a button. It also allows you to create hotspots of any size and position on your screen for interactivity.

Example:

**HOTSPOT Hotspot1 (157, 218, 102, 62) //creates an invisible Hotspot at pixels 157,218**

### ***MBOX Name (X,Y,W)***

MBOX is short for MessageBox, which is similar to a TEXT object, but not restricted to a single string. This is a very powerful graphic object used to display status, instructions, etc. to the operator. The

HEIGHT is not specified by the user, but is calculated by ShowFlow based upon the size of the font. MBOX can receive and output new messages throughout the life of the program MBOXes are always black, with white text.

Example:

**MBOX ShowTimer (85, 276, 331) // Creates a message box called ShowTimer at pixels 85,276**

Note that you must specify the width of the message box. If you then try to display messages that require more width than you have assigned, you must come back to this point and extend your width. Otherwise, the right hand portion of your text will not be seen correctly on screen.

### **LINE Name (Color, Pen Width,X,Y,X2,Y2)**

LINE is a single line segment, from (X, Y) to (X2, Y2), in the color COLOR. Lines can be drawn from any 2 points on your screen in any available color.

Example:

**LINE Line1 (32768, 3, 1,1,10,15) // Draws a white line from 1,1 to 10,15**

### **POLYGON Name (Color, Pen Width, X, Y, X2, Y2,.....Xn,Yn)**

Polygon draws a series of connected lines from points X,Y, to X1,Y1, then to points Xn,Yn. It is important to make sure that your last point is the same as the first point in order to close the polygon. Pen Width is the point size of the outer border. A value of -1 will have no outer border and will instead be filled with the color.

Example:

**POLYGON Polygon2 (32768, -1, 298, 242, 322, 217, 362, 217, 298, 242) // creates a polygon with a fill color of white**

### **OVAL Name (Color, Pen Width, X, Y, W, H)**

An oval object can be shaped as an oval, ellipse, or circle within the boundary points of X,Y,H,W. Pen Width is the point size of the outer border. A value of -1 will have no outer border and will instead be filled with the color.

Example:

**OVAL Oval1 (32768, -1, 426, 224, 80, 70) //creates a white oval size 80, 70 from pixels 426, 224**

### **IMAGE Name ("File.PCX", X,Y,W,H)**

IMAGE is used to place a PCX file picture onto the screen. PCX file format is the default Paintbox format in Windows and other PC compatible programs. This allow you to create complex graphics and pictures in other programs, and then use them as part of your ShowFlow GUI interface. Used with the Hotspot object, this becomes a very powerful interactive feature.

Example:

**IMAGE Image1 ("Mmlogo.pcx", 40, 313, 199, 70) // places the beautiful MediaMation logo on screen**

## **.GUI and Windows Summary**

Your window descriptions can contain as many of these items as you like, with a few limits set on the total number of LEDs and MBOXes of no more than 255 of each per program. Since a typical program needs perhaps 30 LEDs and two or three MBOXes, this should not present a problem. And be sure to enclose the entire window description in curly brackets.

These screens allow you to quickly add new features and operator interface choices to your application. Try not to over whelm the user with the amount of information displayed on a screen. This is especially true of text messages. People respond in a much more intuitive manner to LEDs and Buttons that blink or change color rather than a slew of message boxes displaying text. Typically, you will only have 2 or 3 message boxes, and a lot of well placed LED, graphics, and Buttons on a well designed screen. Anything that makes sense to have it's own screen, simply make one. It is easy and can make the program a lot easier to operate.

At the very least, try to include and/or create an instruction screen that will tell a user what buttons to push on the keyboard to access the automatically generated screens. MediaMation will be happy to provide you with the script for the basic instruction screen we use. The beauty of this entire .GUI interface, is that the entire script is simply a text file. It can be edited and changed easily in any text editor or word processor, but using our ShowCase editor makes all the details described above invisible to the creator.

## SHOWFLOW LANGUAGE SPECIFICATIONS

The heart of the power of ShowFlow is the simple, but powerful, .SFL script. This is the text file that describes exactly what your application will do. A thorough understanding of all the various functions is essential to successfully creating a large and complicated script. Smaller scripts can be created easily with only a basic understanding of the various functions. For most computer users, 2 to 3 days of study and practice is all that is needed to master the entire program!

A ShowFlow script contains 3 main items:

- Variables
- Nodes
- Devices

### **Variables**

A variable is any Value, Memory location, I/O point, Midi message, Graphic Button, or internally generated object that can change it's value or state. An example of this is a digital input. It is either On or Off. All variables can be modified and/or monitored within your ShowFlow script.

There are 7 kinds of variables:

- **outputs** These drive some external device. See below for more information.
- **inputs** These respond to external stimuli. See below for more information.
- **user** These are used internally within your script to store values, make decisions, and display information in your windows. You can have as many of these as you need. Some user variables such as DVars (described later) are automatically generated by ShowFlow. A user variable does NOT have any physical inputs or output.
- **const user** Similar to a user variable, except that it does not display on the variable window and cannot be changed under program control. Typically used for constants that need to be referred to, but not changed. An examples would include things such as "const user White = 32768". This would allow you to refer to the color "White" rather than remember that color number 32768 is white when coloring GUI objects.
- **system variables** These variables are automatically generated by the system. They include:
 

DATE	The date according to the system clock
TIME	The time according to the system clock
Key_Code	The last key pressed on the keyboard
SMPTE_Time	The current SMPTE time in Hrs:Min:Sec:Frames. If no external sync, this time . will indicate the total time that the program has been running.
SMPTE_Start	This is a user settable variable that indicates the starting offset of sequences and . cuelists.
SMPTE_Stop	This is a user settable variable that indicates the ending time of sequences and . cuelists
DAC_Filter	This is the current value of the DAC Filter Scaler (see HRBOX manual).
- **DVar** DVar stands for Digital Variable. Every device (see below) defined by the user will automatically generate a DVar with the same name. For instance, if you declare a DVD player named "DVD1" a DVar named "DVD1DVar" will be generated. These DVars will change their value in response to feedback from the

various devices. For instance, for Pioneer® DVD 7400 units, a DVar value of 4 means “playing”. You can use these DVars to check status, retrieve information, and interact with the various devices you have in your system.

- **group** This variable allows you to group together up to 24 other variables and treat them as a single variable. This is handy for turning large quantities of things on/off at once.

ShowFlow variables must be declared in advance, usually at the top of the script or in a separate, included file. All variables are global in scope, so it is the responsibility of the creator to prevent conflicts in the assignment, naming, and testing of variables.

Variable names cannot contain any spaces, quotes, paranthesis, or +,- characters. They can contain the underscore “\_” symbol to space words apart. The general syntax for variable declaration is:

*[variable] [type] [number] [name]* and optionally for outputs and user variables *[= expression]*

- *variable* indicates whether the variable is an **input**, **output**, or **user** variable.
- The *type* field specifies what kind of variable it is, such as digital I/O, Analog I/O, MIDI message, etc.
- *Number* tells ShowFlow which I/O point, Midi message#, or analog I/O it is referring to.
- *Name* is the name or identifier given to the variable for subsequent use within your script. Once you have named your variable, you need only to refer to it by that name from then on.

## Output Variables

An output variable is used to send a particular value or state (On/Off) to an external device or internal digital I/O card. By setting this variable to a particular state or value, the appropriate output or Midi message will be set. When declaring the output, you can also set it to an initial value by adding “= x” to the end of your declaration. Otherwise, the default value is “0” or “off”. Currently there are 3 types of output variables:

1. Digital Outputs
2. Analog Outputs
3. Midi Outputs

### Digital Outputs

Since ShowFlow is essentially a logic shell that warps around our HRBOX controller, you must remember that all of the HRBOX functions are incorporated within it. This includes the ability to have multiple Digital Output cards of various types and drive capabilities. A digital output is either On or Off. Typically, these are used to drive valves, lights, relays, or other external devices that simply need to be on or off. Any digital outputs defined in the MMATION.CFG file can be declared as an output variable and then accessed from your ShowFlow script. The first digital output in your system as defined by the MMATION.CFG file is output number “0”. Each output then increases sequentially in number after that. You are not required to declare all the outputs, just the ones that you want to access from within your ShowFlow script.

If you set the output variable to “0” or “zero” or clear it, it will turn off. If you set it to any value greater than “0”, or “one”, or supply no argument, it will turn on. You can also test and monitor these outputs from within your script with the standard “if, then” or “trap” commands described later.

All digital I/O’s are also controllable via Midi note On/Off. The particular note number and channel that each output (or input) is assigned to is described in the MMATION.CFG file. These assignments can

be over ridden, however, from the HRBOX.CFG text file that is user editable (see the HRBOX Manual). Regardless, the important thing to remember is that besides having control of them directly from your ShowFlow script, they can also be turned On or Off from external Midi data. Additionally, since ShowFlow treats internally playing Midi sequences the same as externally generated Midi data, you can have ShowFlow play a sequence that controls these outputs, regardless of whether or not you have declared them in your ShowFlow script as digital outputs.

Example:

```

output digital 3 EStop_Lite=1      // declare the 4th digital output as EStop light driver and
                                     // turn it on initially
set (EStop_Lite)                  // Turns on the Estop Light
blink_var (EStop_Lite)           // Begins flashing the output at 1/2 second intervals
clear (EStop_Lite)               // turns off light and/or stops blinking

```

### **Analog Outputs**

If your system is equipped with analog outputs, you can declare them in your ShowFlow script and set them to particular values at will. Just like digital outputs, these outputs also respond to Midi data so there is no guarantee that they will remain at the value you set them to if incoming data is present. Analog outputs have 12 bit resolution. You can set them to any value between 0 and 4095.

Example:

```

output analog 0 Servo_1           //declare the first analog output as Servo #1
set (Servo_1, 2048)              // set the analog output to 1/2 position

```

### **Midi Outputs**

All HRBOX's are equipped with at least 1 Midi I/O port. You can have up to 4 Midi ports. In your ShowFlow script, you can create and declare as many Midi output variables as you want. Available Midi commands that you can declare are:

- Midi Notes
- Midi Program Changes
- Midi Continuous Controllers

By using external Midi devices, you can easily expand the capabilities of your ShowFlow system. A typical example is to use a Midi sampler to play sound effects, or use our DO-16 as a remote digital output device that responds to Midi Note On/Off information. This allows you to run a single Midi cable over to the actual area where you want to control things, and then run short cables directly to the output drivers of the DO-16. These outputs then become ShowFlow variables that you can utilize just like built in Outputs.

To create or declare a Midi command as an output variable in ShowFlow, follow this syntax:

*[direction] [type] [Midi Channel] [number] [name]* and optionally *[= x]*

*direction*                    output  
*type*                            note, program, or controller

*Midi Channel* Channel 1-16 are normal Midi channels, but since this is a computer, we start with “0” instead of “1”. This makes our available channel numbers 0-15 . Additionally, if you have more than 1 Midi port on your system, add 16 to this number to access the second port, 32 to access the third port, and 48 to access the fourth port.

*number* This is the note or controller program number that will be generated when the output is set. Since Program changes have only a single variable assigned to them, there is no number if you are creating a program change output variable.

*name* The name of the variable

=x Sets a default value for the output. “x” is any number between 0 and 127.

Example:

```

output note 0 15 Seat1           // declares a Midi Note number 15 on channel 1 called
                                   // Seat1
output program 15 Preset       // declares a Program change variable on Midi channel 16
                                   // called Preset
output controller 0 7 Volume =64 // declares a Midi Controller number 7 on channel 1
                                   // called Volume with a default value of 64

```

## Input Variables

Input variables are the complement of output variables. They operate in the same manner except that they go in the opposite direction. Input variables will contain the value assigned by the last reception of their associated MIDI message, or their associated digital input pin. Input variables can be initialized, but they may be reassigned at any time by external events. Follow the examples above for specifics.

There are 4 types of input variables as follows:

1. Digital
2. Analog
3. Midi
4. Timecode.

The digital and Midi input variables operate exactly like their output counterparts. The Midi Timecode input allows you to receive incoming SMPTE timecode through the Midi input as Midi Time Code as explained below. This is handy to test against and make “if,then” statements with.

## Digital Inputs

A digital input refers to a physical input card or “Optomux” SSR input that is part of the system. These inputs are defined in the MMATION.CFG file that is configured with your system. A digital input is either On or Off. This can also be referred to as “1” or “0” or as “one” or “zero” within your script. The basic definition for digital inputs is as follows:

*input digital <number> <name>*

Example:

```

input digital 0 In_1           // declares the first digital input point as “In_1”
if (In_1 == 1) StartShow       // looks at In_1 and if on, jumps to the StartShow
                                   routine

```

## Analog Inputs

If your system is equipped with analog inputs, you can declare them in your ShowFlow script and monitor them at will. Just like digital inputs, these inputs also generate Midi data. Analog inputs have 12 bit resolution so they will have a value between 0 and 4095.

Example:

```
input analog 0 Ramp_Position //declare the first analog input as Ramp_Position
```

## Midi Inputs

All HRBOX's are equipped with at least 1 Midi I/O port. You can have up to 4 Midi ports. In your ShowFlow script, you can create and declare as many Midi input variables as you want. Available Midi commands that you can declare are:

- Midi Notes
- Midi Program Changes
- Midi Continuous Controllers

By using external Midi devices, you can easily expand the capabilities of your ShowFlow system. These inputs then become ShowFlow variables that you can utilize just like built in Inputs.

To create or declare a Midi command as an input variable in ShowFlow , follow this syntax:

```
[direction] [type] [Midi Channel] [number] [name]
```

*direction*                   input

*type*                         note, program, or controller

*Midi Channel*           Channel 1-16 are normal Midi channels, but since this is a computer, we start with "0" instead of "1". This makes our available channel numbers 0-15 . Additionally, if you have more than 1 Midi port on your system, add 16 to this number to access the second port, 32 to access the third port, and 48 to access the fourth port.

*number*                   This is the note or controller program number that will be monitored. Since Program changes have only a single variable assigned to them, so there is no number if you are creating a program change input variable.

*name*                      The name of the variable

Example:

```
input note 0 15 Belt1 // monitors Midi Note number 15 on channel 1 called
// Seat1
input program 15 Preset // monitors Program change variable on Midi channel 16
// called Preset
input controller 0 7 Volume =64 // monitors Midi Controller number 7 on channel 1
// called Volume with a default value of 64
```

## User Variables

User variables do not have an input or output. They are merely memory "placeholders" used to store numbers for testing or doing mathmatic operations on. You can declare as many user variables as you

need. The syntax is very easy. User variables can have simple integers or can have time in the standard SMPTE format of Hrs:Min:Sec:Frames

```
user MyVariable = 3           // declares a user variable called MyVariable with a default value of 3
const user Noon = 12:00:00:00 // declares a constant called Noon that is set to 12 hours
```

### **Group Variables**

A group variable allows you treat up to 24 individual variables as a single variable. This is very handy for setting or clearing a lot of variables that typically turn on/off together. You must declare the variables included within the group **before** you declare the group. Syntax is as follows;

```
group <name> variable1, variable2,.....variable”n”
```

Example:

```
group AllLights Light1, Light2, Light3 // groups 3 lights into a variable called AllLights
```

### **Time Code**

Timecode variables are a special case of the user variable that contain the value of the last complete timecode message received on the specified MIDI port. Timecode variables can be trapped, tested, etc. in the same manner as any other variable. When working with timecode variables, values can be specified *in* timecode, i.e., hours, minutes, seconds, and frames. 1 hour, 2 minutes, 3 seconds, and 4 frames is expressed as 01:02:03:04. For example, *set(port0time, 00:01:02:03)* and *inc(someTimecodeVar, 00:00:02:00)* are both legal ShowFlow expressions. Syntax:

```
timecode <midi port #> <frame_rate > <NAME> <default value in hh:mm:ss:ff>
```

Example:

```
timecode 0 30 port0time           // declares a timecode variable to monitor
incoming SMPTE code              // (MTC) on the first Midi port
```

*Note: In the current implementation the frame rate is always 30 fps, regardless of the value specified in the variable declaration.*

### **Devices**

ShowFlow allows you to control many different devices directly. A device is loosely classified as an internal or external component that has the ability to operate independently of your ShowFlow script instructions. This allows you to send simple commands to these devices to initiate their internal functions and/or playback. If a device is fully supported in ShowFlow, the interpretation of commands and any handshaking that takes place is handled automatically. Each device has its own specific commands, but we try to make the control of each device as similar as possible. Certain commands such as “stop”, “play”, and “seek”, are common to all devices that can support them. We are constantly adding new devices to ShowFlow, so keep in touch with us for updates.

You must declare your devices just like your variables before you can control them. Currently, the following devices are supported:

- Pioneer Laser Disk Players
- Moog Electric Motion Bases
- Pioneer Film Projector Controllers (not the same company as the Pioneer above)
- Internal Type “0” Midi Sequencers (single multi channel midi file)
- Internal Type “1” Midi Sequencer (Multi track Midi File)

A final device is what we call a “generic serial device”. Although not as elaborate a support structure as the other devices, this generic device allows you to declare and control just about any serial controlled device (RS232, 422, 485) even if we don not have full support for it built into ShowFlow. This usually works very well for just about any device you may want to control. If you have a particular device you want supported, please call us to discuss adding it to our list.

### ***Pioneer Laser Disk Players***

These Laser Disk Players or LDP’s are very popular in themed attractions. They are reliable, inexpensive, and provide high quality video and audio playback. ShowFlow fully supports these LDP’s. Once declared, you can play, search, pause, still frame, and derive a SMPTE sync signal from any pro level Pioneer LDP connected to a serial port on the HRBOX. This last feature is very powerful as it allows you to use the LDP as master synchronization device even if you have no SMPTE timecode recorded on the disk. By polling the device continuously, we can ascertain which frame number it is on and convert it to a SMPTE equivalent to drive ShowFlow internal Midi sequencers.

The syntax to declare a Pioneer LDP is as follows:

**laserdisk PIONEER [serial port] [name]**

The serial port refers to ports A-J as defined in your MMATION.CFG file and supported by your hardware. ShowFlow can currently have up to 10 independent serial ports per system. MediaMation must configure your system to support the appropriate number of ports. We ship HRBOX with 2 serial ports standard.

The following commands are available for Pioneer LDP’s

- **play (LDPname)** Causes the player to begin playback
- **stop (LDPname)** Causes the player to stop playback
- **seek (LDPname,xxxxx)** Searches the player to a particular frame number or SMPTE time
- **cmd (LDPname,statuson)** Causes ShowFlow to begin monitoring the LDP’s status
- **cmd (LDPname,statusoff)** Causes ShowFlow to stop monitoring the LDP’s status
- **sync (LDPname)** Causes ShowFlow to poll the LDP and derive a SMPTE timecode from the current frame numbers and use this for the sequencer playback.
- **cmd (LDPname,”xx”)** Allows the ShowFlow script to issue any other command not listed above to the LDP by simply entering the ASCII characters as described in the Pioneer manual.

### ***Moog Electric Motion Base***

The Moog electric motion base is an excellent, and very popular, motion base for simulator theaters around the world. They come in 2 basic options: with ride storage (moogrs), or without ride storage (moog). As a supported device, all the handshaking and other functions are taken care of automatically and/or simplified into easy to use commands. Each Moog base has to have a unique “unit Number” that is assigned to the base via its internal computer. As many as 32 bases can be daisy chained to a single serial port on the HRBOX. To declare a base, the syntax is as follows:

**motionbase moogrs [serial port A-J] [pod ID#] [name]**

The control of a Moog base without ride storage is more specialized. Please contact MediaMation for information on the specific for that task. The following commands are available for a MoogRS motion base:

- **stop (MoogBase1) // (MoogBase1 is a sample name, not a fixed command)**
- **play (MoogBase1)**
- **cmd (MoogBase1 , load)**
- **cmd (MoogBase1 , engage)**
- **cmd (MoogBase1 , devstop)**
- **cmd (MoogBase1 , engage)**
- **cmd (MoogBase1 , run )**
- **cmd (MoogBase1 , file, xxxx)**
- **cmd (MoogBase1 , reset)**
- **cmd (MoogBase1 , inhibit )**
- **cmd (MoogBase1 , disable )**
- **cmd (MoogBase1 , estop )**
- **cmd (MoogBase1 , status )**
- **cmd (MoogBase1 , park )**
- **cmd (MoogBase1 , query)**

Please see the Moog manual for a detailed description of what each of these commands controls. All periodic polling of each Moog base is automatic within ShowFlow.

### ***Pioneer Film Projector Controller***

The Pioneer electronic film projector controller is a popular controller for large format film projectors. It is equipped with a RS232 communication port and ShowFlow supports this device. The syntax to declare a projector is as follows:

**projector [serial port A-J] [name]**

The Pioneer Electronic Film Projector controller has a small set of ASCII commands that can be issued depending upon the projector and configuration purchased. To simplify integration, ShowFlow can issue any of these commands with the following syntax to the projector.

**cmd (ProjectorName, ASCII, "FP")**

The characters inside the quotes are transmitted to the projector with the proper termination characters. Please refer to the Pioneer manual for the various commands available. Responses are read by ShowFlow and displayed in the DVar for the projector.

Additionally, you can play, stop, and turn the polling of the projectors current status on/off with the following:

**cmd (ProjectorName, statuson)**

**cmd (ProjectorName, statusoff)**

**play (ProjectorName)**

**stop (ProjectorName)**

You must be careful with this projector as many of the responses are identical regardless of the "question" asked from your script. That is why you can turn status on or off. Sometimes you need to

turn it off before you ask a question of the projector to prevent a response from the status poll being confused with the response to the query you have asked.

### **Single Track Midi Sequencers**

ShowFlow allows you to declare and play back as many midi files as you want to simultaneously. These files are typically used to store complex motion profiles and sequential show data. As a Midi sequencer is one of the best and most powerful way to program shows, this feature is invaluable. Simply save your Midi sequence as a Standard Midi File (SMF) type "0" and you can then replay the file anytime you wish from ShowFlow. You can play multiple files simultaneously or asynchronously with the playback of one file not affecting the others currently playing. You can also play from the internal clock, or the SMPTE timecode coming in as MTC through the Midi In jack, or from the sync signal derived directly from a Pioneer LDP as described above. Remember, all of the digital and analog outputs in your HRBOX are available to these Midi files as they playback within ShowFlow. This is a powerful feature allowing you access to these in both a pre-programmed show and as part of your logic script. The Midi sequence is the normal way to play back show data within ShowFlow.

To declare a type "0" Midi sequencer, use the following syntax:

**sequencer singletrack [name]**

There are 2 things you must do to play back a Midi file from your sequencer. The first is to make sure that the SMF you have saved on your disk has a .MID suffix attached to it. Secondly, you must "load" the file into your sequencer as follows:

**load (Sequencer1, show.mid)**

The other 3 commands are Play, Stop, and Sync.

**play (Sequencer1)**

**stop (Sequencer1)**

**sync ([sync type])**

The 3 available sync types are:

- internal                Plays the sequenc from ShowFlow's internal clock
- 7. MTC                    Plays the sequence locked to incoming Midi Time Code
- 8. [LDP name]            Plays the sequence locked to the SMPTE code derived from an LDP

### **Sequencer MultiTrack**

The Multitrack sequencer is used if you want to record Midi data into ShowFlow. This is useful for certain interactive applications. It operates identically to the singletrack sequencer except that it has the additional "record" command. The sequencer is a 16 track sequencer with one track assigned to each of the 16 available Midi Channels. Additionally, using the "load" command without telling it what .MID file to load will empty the contents of the multitrack sequencer. The syntax for declaring and recording is as follows:

**sequencer multitrack [name]        // Creates a multitrack sequencer**

**record ([sequencer name], [Midi Channel])**

The Midi channel is between 1-16. Data on tracks that are not currently in record will play back.

Recording on a track erases the existing information on that track.

## Generic Serial Devices

The generic serial device gives you a very versatile method of controlling most any serial device regardless of whether or not it is a supported device in ShowFlow. Certain devices are too obscure or simple to justify programming hooks into the ShowFlow program, and this generic device handles them. It has been used for things such as Video CD players and Visa/MasterCard terminals in automated interactive kiosks.

The main function of a generic serial device is to send commands to the device, and listen to its responses. The generic serial device gives you this ability. To declare a generic device, use the following syntax:

**generic SERIAL [serial port] [name]**

There are four commands for the generic serial device:

- ASCII                Send an ASCII string to the device
- 3. BINARY            Send a Binary string to the device
- 4. LISTEN            Listen for a particular ASCII String from the device
- 5. UPDATE            Place the incoming numeric response into the devices DVar

Here are the examples:

```
cmd (GenDevice,ASCII,"PL")     // sends the ASCII "PL" to a device named GenDevice
cmd (GenDevice,BINARY,13)     // sends a Binary 13 (CR) to GenDevice. Most
                               // serial devices need a CR or LF (carriage return /
                               // line feed) after each command. Since you
                               // cannot type these things into your script, you can send
                               // them as a BINARY command.

cmd (GenDevice, LISTEN, "Any Message")
                               // Tells ShowFlow to listen for the string "Any Message"
                               // from the GenDevice. When it hears that particular
                               // message, ShowFlow will automatically increment the
                               // generic devices DVar by 1. This allows you to know
                               // when that response has occurred.

cmd (GenDevice,UPDATE)        // transfers all the numeric characters received next into the
                               // generic devices DVar until a non numeric character is
                               // recognized, such as a letter or CR.. This is typically used
                               // in conjunction with LISTEN to know exactly when to
                               // update the DVar.
```

## DVar's

A DVar is short for Digital Variable. Whenever you define a device or machine in your initial declarations, a DVar for that device is automatically created. This DVar is displayed in the System Variables screen and will have the name "your\_nameDVar". You can set, monitor, and test this DVar to find out the status of the particular device assigned to it. This DVar changes based upon the status of that device. For example, if you declare a Midi Sequencer in your script, a DVar for that sequencer will be created and will tell you if the sequencer is idle (value of 0), playing (value of 1), or waiting for external synchronization signals (value =3). Below are listed the Dvar values for the various devices described above:

### **Pioneer Laser Disk Players**

Park = 1  
Play = 4  
Still = 5  
Pause = 6  
Search = 7

### **Moog Electric Motion Base**

The status response and hence the moogrsDVar describes two things about the moog; settled or not and base state. The moogrsDVar is the addition of these two pieces of information. The settled value adds decimal 16 to the number, thus any value greater than 16 indicates the base is not settled. Refer to the MOOG manual for further details.

For example:

moogrsDVar = 2 means the moogrs is settled and is in standby state.

moogrsDVar = 18 means the moogrs is not settled and is in standby state.

Settled values:

MOOGRS\_SETTLED 0x00 = 0 decimal

MOOGRS\_NOT\_SETTLED 0x10 = 16 decimal

#### **Other Base State Values:**

MOOGRS_POWERUP	0x00
MOOGRS_IDLE	0x01
MOOGRS_STANDBY	0x02
MOOGRS_ENGAGED	0x03
MOOGRS_RUN	0x04
MOOGRS_PARKING	0x05
MOOGRS_DISABLED	0x06
MOOGRS_INHIBITED	0x07
MOOGRS_FAULT2	0x08
MOOGRS_FAULT3	0x09 = 9 decimal
MOOGRS_NORESPONSE	0x0A = 10 decimal value seen in variable window

### **Sequencer DVars**

There are 4 different states that your sequencer (both single and multi track versions) can be in. The DVar indicates that state. The status is as follows:

Idle = 0  
Playing = 1  
Recording = 2  
Waiting for external sync = 3

### **Generic Devices**

A generic device by definition cannot have any fixed DVar values. The DVar will be determined by your script.

## NODE Statements

The remainder of the program text file consists of one or more Node statements. The balance of this overview will be dedicated to the explanation of the nodes, the core programming units of ShowFlow.

Node definitions begin with the keyword **NODE** followed by a unique node name. This identifier can either be a number or a more intuitive label for the node, such as 'Run\_Show\_Sequence'. The node name can only contain one word or number. No spaces are allowed. Each node needs a unique name, number or combination. No two nodes can have the same name or else the program will not be able to differentiate them and will not compile properly.

Each node name is followed by the actual function, or command, to be executed. The function is followed by the keyword **NEXT** and identifier of the next node to be executed. A complete node specification takes the following form:

```
node < node name>
func (argument1, argument2)
next < another node name>
```

The *next* command allows you to tell the script exactly which node to execute next. Typically this is the node immediately after the current node, but many times, you may need to jump to another place in the program once a node has executed. It is important to understand that jumping, or branching, to another node is NOT the same as calling a subroutine. A subroutine is designed to execute all of its nodes, and then return to the point in the program where you were before you went to the subroutine.

Branching is also supported with the **IF()** function, which evaluates an expression and changes program flow based on the result. Control transfers to the specified branch node when the expression is true, otherwise control passes to the 'next' node.

The only required node name is for the node where you want the script to begin executing. This node **MUST** be named **BEGIN** in capital letters. Example:

```
node BEGIN
null ()
next The_Next_Node
```

## SUBROUTINES

A subroutine is a portion of your script that acts like a small independent section of the entire script. It is an excellent way to “compartmentalize” your script. It is highly recommended that you break your entire script into a series of subroutines. This allows you break each function of your script into smaller, easily managed, routines. Additionally, it is wise to make a single subroutine for any repetitive functions. That way, you only have to write the function once and can call it from various parts of your script as needed.

Subroutines are supported with the **CALL()** function, or the **TRAP()** function which stores the current program location on an execution stack and branches to the specified node. Subroutines are terminated when a node with a “next -1” is executed, at which time execution will resume at the node following the original **CALL()** function or the last node before the **TRAP()** was triggered. All subroutines **MUST** end with a “next -1” for the last line of the last node in that subroutine.

A well designed program will typically contain mostly subroutines. This makes tracing problems and making changes very easy. For example, the actual flow of a program will typically look something like this:

```

node BEGIN // Start of program execution
call (Initialize_Outputs) // Calls a subroutine to set the initial values of your outputs
next TrapButtons

node TrapButtons
call (TrapAll) // Calls a subroutine to trap your operator interface buttons
next PrintStatus

node PrintStatus
call (StatusRunning) // Calls a subroutine to print messages on the screen to
next Loop // inform the operator of the current status

node Loop
null () // The program simply sits here and loops around until a
next Loop // trap is triggered and runs a particular subroutine.

```

This could easily be the main program loop for a very complicated program. Each the subroutines will probably be made up of even more subroutines. The TRAP() function (described in detail below) allows you to monitor I/O's without dedicating any processor time to the monitoring function. This simplification of the program makes troubleshooting and changes very easy.

## TRAPS

One of the most powerful features of ShowFlow is the TRAP() function. a TRAP monitors the state of a variable and jumps to a particular node in your script when that variable changes or changes to a specific value or range. You can have hundreds of traps running simultaneously without any effect on the speed of the program or processing time. You can trap any variable defined in your script. This includes, inputs, outputs, LED's, buttons, hotspots, user variables, Midi I/O's, timecode, and DVars. You can trap when a variable is a certain value such as

```

trap (Digital_In_1 == 1) GoHere // when Digital_In_1 is on or "1", jump to the subroutine
// starting at the node "GoHere"
or
trap (User_1 >200) GoHere // when User_1 is greater than "200", jump to the
// subroutine starting at the node "GoHere"

```

You can also trap a variable for any change by trapping it to NOT EQUAL itself. For example:

```

trap (Digital_In_1 != Digital_In_1) GoHere // when Digital_In_1 changes its value, jump
// to the subroutine starting at the node
// "GoHere"

```

Traps allow you to emulate all the "ladder Logic" functions typically associated with PLC's and more. It allows the ability to monitor all kinds of variables even as your program is progressing. This is

essential for safety monitoring, EStops, asynchronous input processing, and user interfaces. Of course, there are hundreds of other uses for TRAP() that you will discover as you program in ShowFlow. An important new addition to ShowFlow is the ability to run multiple traps on the same variable. This way, you can jump to a number of different subroutines based on the value of a single variable. This is perfect for monitoring user, Midi or Analog I/O's and processing the results based on the number received. Many times, the use of multiple traps on the same variable is much faster and easier than running a series of IF statements on that same variable to determine the proper response.

As a necessity, when a trap is triggered, all other traps are suspended until a RESUME() command is run. However, even if you are processing a trap and other traps are triggered before you tell ShowFlow to resume traps, the states of all the other traps are remembered and will trigger as soon as a RESUME() command is initiated if they have changed to a trapped state. Typically, you will put a RESUME() command as the first node in a subroutine called by a TRAP(). This insures that all other traps will still operate while you are processing that particular trap. Many times, however, you may want to suspend all other traps until you are ready to activate them again. This allows you to prioitize your traps and subroutines. Simply put a RESUME() as the last node in your subroutine and that subroutine will take prioity over all other traps until it is complete.

**Remember, ALL subroutines called by a trap should have a RESUME() in them somewhere, or else all your traps will be suspended until a RESUME() is encountered elsewhere. It is highly recommended that you include the RESUME() command in every subroutine called by a TRAP().**

Of course, as with all things in life, the need to do a particular operation also requires the need to do the opposite sometimes. ShowFlow accomplishes this with the UNTRAP() command. UNTRAP() removes a trap from a variable. You can UNTRAP() a paritcular trap on a variable, all traps on a variable, or all currently set traps in the whole script. For example:

First, we set a couple of traps;

**trap (User\_1==5) GoHere**                      *Will run the subroutine called "GoHere" when User\_1 changes to 5.*

**trap (User\_1==6) GoThere**                      *Will run the subroutine called "GoThere" when User\_1 changes to 6.*

Then, we can selectively remove traps;

**untrap (User\_1 ,GoHere)**                      *Untraps only the traps set for the variable User\_1 that are set to call the subroutine "GoHere". All other traps on that variable are still active.*

**untrap (User\_1)**                                      *Untraps ALL traps on the variable User\_1.*

**untrap (ALL)**                                      *Untraps all currently running traps within your entire script.*

The use of traps within ShowFlow is a very powerful feature. The speed with which they are executed, and the fact that they do not take up any processing time while they are being monitored makes them the preferable over a typical IF command used in ShowFlow or most other programming languages. Once you set the trap, the program simply moves on to the node directed by the *next* command in that node.

This makes it possible to quickly set hundreds of traps within a couple of milliseconds and then continue on with the normal execution of the program.

## **NODE PARAMETERS**

Finally we are at the actual functions of the program. ShowFlow's ease of programming stems from the fact that almost anything you may need to do can be done with these simple functions or *objects*. It is a very "high level", *object oriented* language. What this means is that the function each node does is described with these simple commands, regardless of the amount of specialized processing that takes place behind the scenes to actually accomplish the task.

For example, lets look at a node function such as BLINK\_LED that causes an on screen LED to oscillate between its original color, and another color every 1/2 second until it is reset. The ShowFlow function node is simple:

```
node BlinkMe
blink_led (OnScreenLED#1, 25)    // Blinks the LED with color #25
next NextNode
```

Pretty simple! Now, if we were to really analyze all the steps needed to accomplish this inside the computer, it may flow something like this:

- Find the pixels on the screen described as OnScreenLED#1
- 3. Store their current color in buffer x, and store palette color #25 in buffer y.
- 4. Set a counter to 500
- 5. Wait 1 millisecond
- 6. Decrement the counter.
- 7. Check to see if this blink function has been canceled by a "clear" or "set" function called somewhere else in this script. If so, change the color of each pixel within the portion of the screen described as OnScreenLED#1 to color palette number previously stored in buffer x and stop running this function.
- 8. Does the counter equal 0 yet? If not return to step 4.
- 9. If counter equals 0, change the color of each pixel within the portion of the screen described as OnScreenLED#1 to the color palette number stored in the buffer x.
- 10. Repeat steps 3-8, but alternately change the palette color in step 8 between the values stored in buffers x and y each time through this loop.

Even this long, drawn out description of the internal workings of the function are *greatly* simplified, and does not take into account any of the interfacing to the rest of the program, which continues to execute even though this LED is blinking. Step 6 alone is actually about 6 different functions. Nor are any of the specifics, such as where in the computers memory we store the values or set up the counter, determined in this simple description. Add to this that most high to mid level programming languages have very specific syntax and format rules, not the straight forward English descriptions used above, and you can start to get an idea of the tremendous amount of "behind the scenes" work that ShowFlow takes care of for you automatically.

Below is a list of the most common parameters required and used in this documentation. Following that is a complete list of all the node functions, their required parameters, and explanations of what they do. Again, the ShowFlow compiler is case-sensitive, but all keywords are capitalized for clarity.

<b>COLOR</b>	A palette color number from 0 to 255
<b>LEDNAME</b>	An LED identifier, as defined in the .GUI file
<b>NODENAME</b>	A node identifier
<b>MSGNAME</b>	A Message Box identifier, as defined in the .GUI file
<b>STRING</b>	A sentence, word, or phrase enclosed in double quotes (")
<b>VARNAME</b>	The name of a ShowFlow variable
<b>BOOLEAN</b>	1 or 0, YES or NO, TRUE or FALSE, ON or OFF, etc.

## **FUNCTIONS**

### **NULL()**

Null nodes are placeholders, and take no parameters. When ShowFlow encounters a null() node, no function is performed, and control passes to the specified next node.

Example:

```
node ThisNode
null ()
next TheNextNode
```

### **COLOR\_LED( LEDNAME, COLOR)**

Color\_led is used to set the color of an onscreen LED. If the LED was blinking before, it stops blinking and assumes the specified color.

Example:

```
node ThisNode
color_led (OnScreen_LED, 100)
next TheNextNode
```

### **BLINK\_LED( LEDNAME, COLOR )**

Blink\_led maintains the previous color of an onscreen LED, but blinks the LED between the previous color and the new, specified one. The blink period is fixed to alternate every half-second, and all LEDs use the same timer to determine when to blink. This means that all LEDs will blink simultaneously.

Example:

```
node ThisNode
blink_led (OnScreen_LED, 100)
next TheNextNode
```

### **COLOR\_BUTTON( BUTTONNAME, COLOR)**

Color\_button is used to set the color of an onscreen BUTTON. If the BUTTON was blinking before, it stops blinking and assumes the specified color.

Example:

```
node ThisNode
color_button (OnScreen_BUTTON, 100)
next TheNextNode
```

### **BLINK\_BUTTON( BUTTONNAME, COLOR )**

Blink\_button maintains the previous color of an onscreen BUTTON, but blinks the BUTTON between the previous color and the new, specified one. The blink period is fixed to alternate every half-second, and all BUTTONs use the same timer to determine when to blink. This means that all BUTTONs will blink simultaneously.

Example:

```
node ThisNode
blink_button (OnScreen_BUTTON, 100)
next TheNextNode
```

### **TRAP\_BUTTON ( BUTTONNAME) NODENAME**

Trap\_button is used for catching user input on the video monitor. When a trap\_button command is issued, the region on the screen occupied by the button becomes a 'hot spot' until it is either pressed, clicked, or deactivated (see UNTRAP\_BUTTON). Note that in order for the trap to work, the user must also RELEASE the mouse, or stop touching the screen, on the LED. If the user clicks on the button, but moves the mouse off of the button before releasing the mouse button, no trap event takes place.

When the trap event actually does happen, execution of the program is interrupted and control handed over to the node specified in NODENAME. Subsequent attempts to press the button will not do anything until the program explicitly asks the button to trap again.

Example:

```
node ThisNode
trap_button (OnScreen_Button) GoHere
next TheNextNode
```

### **UNTRAP\_BUTTON( BUTTONNAME )**

Untrap\_button cancels the pushbutton function of a BUTTON previously set to trap by TRAP\_BUTTON. An example would be a screen where you wish the user to press one of two buttons. When the user presses one button, that button automatically will stop being polled. You will need to explicitly tell ShowFlow if you want the other button to also stop being polled by use of untrap\_button. Optionally, if you use the keyword 'all' in place of the name of a particular BUTTON, ShowFlow will search through all of your BUTTONs and deactivate any that are still attempting to trap user input.

Example:

```
node ThisNode
untrap_button (OnScreen_Button)
next TheNextNode
```

### **MSG( MSGNAME, STRING ) and LOGMSG( MSGNAME, STRING )**

Msg and Logmsg both output the specified string to the appropriate message box, erasing any previous message in that space. Clicking on a message box will reveal the last five messages entered using MSG or LOGMSG.

LOGMSG differs from MSG only in that the STRING is also written to the log file, 'RECORD.TXT', preceded by the appropriate date and time stamp for later use.

Example:

```
node ThisNode
msg (StatusMessageBox, "Show is Running") // Prints text in the on screen message box
next TheNextNode
```

### **SPIT( MSGNAME, VARNAME ) and LOGSPIT( MSGNAME, VARNAME )**

Both spit and logspit perform the same function, with logspit additionally writing the message to the log file with the correct date and time stamp. Spit writes to a message box the name of the variable, followed by a colon, and then the current value of that variable i.e.

"Passengers: 15" or "Show\_Number: 3".

This command is especially useful for keeping track of things on screen and maintaining a record file on the disk that can be downloaded later. This file can be especially helpful for having a stored history of the programs progress for troubleshooting, or billing purposes.

Example:

```
node ThisNode
logspit (StatusMessageBox, "User_1") // Prints current value of User_1 to on screen
next TheNextNode // message box and saves to disk
```

### **WINDOW( WINDOWNAME )**

Window deactivates the current display window and brings up another window on the monitor, as defined in the .GUI file. No attempt is made to erase the previous window: it is simply overwritten on screen.

Example:

```
node ThisNode
window (Stop_ShowWIndow)
next TheNextNode
```

### **CALL( NODENAME )**

Call() interrupts the flow of the program begins execution of a subroutine which begins with the NODENAME specified. Execution at the new location continues until a 'next' node with a number of -1 is encountered, at which point execution resumes at the function after the call. This is used in a fashion similar to subroutines and subfunctions in other programming languages.

Example:

```
node ThisNode
call (RunTheShow) // Runs the subroutine "RunTheShow"
next TheNextNode
```

### **WAIT( NUMBER )**

Wait does nothing more than delay execution of the program by some amount of time before continuing on to the next function. The number given is in milliseconds (1/1000 sec). So a wait(3000) function would cause the program to wait 3 seconds before continuing. This is useful if you need to delay to give the operator time to respond, read a status message on screen, or give an external device time to respond to a command.

Example:

9/13/2000

```
node ThisNode
wait (1000)
next TheNextNode
```

### **SCHEDULE( NUMBER, NODENAME)**

Schedule is a tricky command to use effectively. Schedule works like a CALL function, but does not actually call the subroutine until a certain amount of time has passed. While waiting for this time to pass, ShowFlow continues on with the next node specified. Schedule effectively tells ShowFlow to interrupt the regular flow of the program and redirect execution to the specified subroutine given in NODENAME, but not to actually interrupt until a certain amount of time later, given in milliseconds. This is useful for the creation of periodic routines, which can end with the function to call itself again a second later, no matter what the program wants to be doing.

Schedule differs from WAIT in that the program continues on while waiting. Many times, it is itself the main function of a subroutine. This allows you the flexibility of only calling one subroutine and knowing that it will automatically complete it's operation a set amount of time later, without any significant interruption of your noraml program. Some uses for this function would be

- Trap a digital input. When the trap is triggered, turn on a digital output, schedule it to run another subroutine to turn itself off a few seconds later as required, and go back to the main loop while waiting.
- Trap and Blink an interactive button and schedule it to clear if noone pushes it within a certain amount of time. Meanwhile, jump back to the main program as you may have other asynchronous interactive buttons that you are dealing with and don't want to interrupt their function.

Example:

```
node ThisNode
schedule (2000, GoHere)    //In 2 seconds, run the subroutine "GoHere"
next TheNextNode
```

### **FLUSH( )**

Flush() takes no arguments. It simply cancels ALL events scheduled to take place in the future, as created by the SCHEDULE command. There is currently no method of flushing only a particular SCHEDULE command without affecting all current schedules taking place.

Example:

```
node ThisNode
flush()
next TheNextNode
```

### **EXIT( )**

Exit() takes no arguments. Exit forces termination of the ShowFlow program. This command should be avoided except during field testing and development, as it does not change any output before ending the program and quitting to DOS. There is also a good chance that it will simply freeze and crash your computer!

Example:

```
node ThisNode
```

```

exit ()
next Who_Cares_We_Are_Done_Now

```

## RESTART()

This function restarts the ShowFlow script at the first node (“node begin”). Variables are not affected. Restart() is intended to be used to recover from emergency stop conditions, and it is the only reliable way to return from subroutines other than the “next -1” syntax. It does not, however, clear the stack pointer, which is used by ShowFlow to keep track of what subroutines are currently being executed and where to return to when the subroutine is finished (next -1). Continuous use of this function could cause the stack to eventually “fill up” and overflow which causes a fatal crash of your computer! This command is typically used in conjunction with the BLOWTHESTACK command.

Example:

```

node ThisNode
restart ()
next Who_Cares_We_Are_Done_Now

```

## TRAP( VARNAME, BOOLEAN FUNCTION) NODENAME

This function is similar to TRAP\_LED, except that it can be applied to any variable, and it traps to a particular state, value, or range. ShowFlow will monitor the specified variable and branch to NODENAME when the variable changes to the state described in your boolean function. When a trap occurs, that particular variable value will not be trapped again until another TRAP() function is executed on that variable with the same boolean function. Also, when any trap occurs, ShowFlow stops checking ALL other traps until a 'RESUME' command is executed.

An important new addition to ShowFlow is the ability to run multiple traps on the same variable. This way, you can jump to a number of different subroutines based on the value of a single variable. This is perfect for monitoring user, Midi or Analog I/O's and processing the results based on the number received. Many times, the use of multiple traps on the same variable is much faster and easier than running a series of IF statements on that same variable to determine the proper response.

Example:

```

node ThisNode
trap (Digital_In_1 == ON) GoHere // when Digital_In_1 is on or "1", jump to the subroutine
next TheNextNode // starting at the node "GoHere"

node ThisNode
trap (User_1 >200) GoHere // when User_1 is greater than "200", jump to the
next TheNextNode // subroutine starting at the node "GoHere"

node ThisNode
trap (User_1 <100) GoThere // when User_1 is less than "100", jump to the
next TheNextNode // subroutine starting at the node "GoThere"

```

***Note that we are actually running 2 separate traps on the variable User\_1 above***

You can also trap a variable for ANY change by trapping it to NOT EQUAL itself.

For example:

```
node ThisNode
trap (Digital_In_1 != Digital_In_1) GoHere // when Digital_In_1 changes its value, jump
next TheNextNode // to the subroutine starting at the node "GoHere"
```

### UNTRAP( VARNAME )

Untrap forces ShowFlow to stop monitoring a variable for a change in state. See UNTRAP\_LED for an example situation. UNTRAP optionally takes the keyword 'all' to deactivate every trapped digital input. If a variable has multiple traps on different values associated with it, you can selectively untrap only those traps you want to.

For example:

```
node ThisNode
untrap (ALL) // Untraps ALL traps currently running on all variables
next TheNextNode
```

or

```
node ThisNode
untrap (User1) // Untraps all traps currently running on the variable User1
next TheNextNode
```

or

```
node ThisNode
untrap (User1, GoHere) // Untraps only the trap on User1 that is assigned to call the
next TheNextNode // subroutine "GoHere". All other traps are left untouched
```

### RESUME( )

Used in conjunction with the trap() command. Use RESUME at the beginning or end of a 'trap' subroutine. When a variable is trapped, checking of all other trapped variables is halted to allow full processing of the trapped event. The RESUME command resumes trap processing. It is required to run a RESUME after any trap has been triggered in order to re-activate any other traps, even if you have not untrapped them. Even though a RESUME has not been initiated, ShowFlow keeps track of all other current traps and will trigger any traps that may have been triggered before a RESUME was initiated. This insures that even though you are processing a particular trap subroutine, you will not "miss" any other traps that may have occurred during this processing. This allows you to prioritize your traps. Any trap subroutines that are low priority can have a RESUME as the first node in order to re-activate all other traps. High priority traps, however, can wait until the end of their subroutine to RESUME to guarantee that other traps will not interrupt their processing.

For example:

```
node ThisNode
resume () // Resumes all currently active traps
next TheNextNode
```

### **CLEAR( VARNAME )**

If the specified variable is not some form of digital input, clears the variable's value to zero. If it's a digital output, also set that output low. If it's a MIDI controller or note, send the appropriate MIDI message.

### **SET( VARNAME [EXPRESSION] )**

Set is used to assign a value to a variable. If no expression is given, the variable is set to 1. The expression can be a number, such as 5, or the name of another variable. If VARNAME refers to a digital input or MIDI node, the value is overridden to be 1/high/on, and the appropriate message is sent. If VARNAME is a midi controller, the appropriate MIDI message is sent.

### **INC( VARNAME [EXPRESSION] ) and DEC( VARNAME [EXPRESSION] )**

Inc and Dec increase and decrease a variable's value, respectively. If no expression is given, the variable is increased or decreased by 1. The expression can be a number, such as 5, or the name of another variable. If VARNAME refers to a digital input or MIDI node, INC is replaced by SET() and DEC is replaced by CLEAR(). If VARNAME is a midi controller, the appropriate MIDI message is sent.

### **BLINK\_VAR( VARNAME )**

Blink\_var can only be used on a digital output variable. Like BLINK\_LED, the blinking is synchronized to an internal timer that alternates every half-second. An output is blinked until a subsequent function SETs or CLEARs it.

### **LOAD( FILENAME )**

Load is an function to the HRBOX playback device. Load loads into ShowFlow a Standard MIDI file type 0 for playback at a later time. ShowFlow can intelligently interpret the clock cycle of the MIDI file and recalibrate it to its millisecond timer.

### **HOME( )**

The HOME command directs all connected servos to their home positions at a safe speed. The home position and safe speed is determined by the HRBOX.CFG configuration file. The command, once issued, will continue to attempt to bring the servos home until a command to move to some other position is sent to HRBOX.

### **SYNC(option)**

Sync tells HRBOX which clock timer to use for playback. If Sync is MTC, then playback is slaved to an external timing device such as a MIDI synchronizing timer. If Sync is INTERNAL, then playback will start immediately upon the play(sequence) command. If Sync is LASERDISK then the frame number given from the laserdisk player will be interpolated into time from zero and lock to the frame return numbers, Any playback will not take place unless the incoming time code lies between the start times and stop times predefined for the show as specified in the HRBOX.CFG configuration file. Otherwise, playback is slaved to the CPU's own millisecond clock.

### **MOUSE( BOOLEAN )**

Mouse tells HRBOX to display or hide the mouse pointer. If Mouse is on/true/yes/1, then the mouse pointer is displayed, otherwise it is hidden..

### **RAMP( BOOLEAN )**

Ramp tells HRBOX to activate or deactivate safety ramping. Safety ramping refers to severaly clipping the velocity at which a servo changes from one state to another, resulting in a slow, steady movement of all servos. Ramping works on all servos simultaneously, not on individual servos.

### **KINEMATICS( BOOLEAN )**

Kinematics tells HRBOX the nature of the MIDI playback file or incoming MIDI data. If Kinematics is off/clear/etc., then all MIDI data reflects directly what the output should be. If, however, the kinematics are on/set/etc., this means that all playback or incoming MIDI data are actually variables to be plugged into six simultaneous kinematics equations of six unknowns to determine how to position the servos. This kinematics filter is generally constructed by a third party and incorporated into the ShowFlow program to reflect the needs of a particular piece of hardware.

### **START\_SHOW( )**

Start\_show takes no parameters. This command starts playback of the last loaded MIDI file through HRBOX from the beginning. See the HRBOX specifications for more on show playback.

### **STOP\_SHOW( )**

Stop\_show takes no parameters. This command unconditionally halts playback of any playing MIDI file before it has ended normally.

### **MIDI\_OUT( MIDISTREAM )**

Midi\_out allows you to directly send specific midi bytes to any devices on your MIDI chain. Up to 16 bytes can be given in decimal, per MIDISTREAM, all delineated by commas, but there is no limit on the number of midi\_out commands you issue. This command is useful for sending System Exclusive commands to particular devices.

### **SERIAL\_OUT( PORT, STRING )**

Serial\_out allows you to transmit a series of ASCII characters from serial ports specified in the hardware configuration file. This command is useful for controlling devices that take serial input, such as laser disc players.

PORT is a case insensitive ASCII character 'a' or 'b' that selects the serial port for the transmission. Attempts to send bytes out an unsupported PORT will generate either a compile-time or run-time error. STRING can contain up to 16 printable characters. If STRING contains spaces or tabs it must be enclosed in double quotes. SERIAL\_OUT transmits a carriage return (ASCII 13) after transmitting STRING.

Examples:

```
serial_out(a, "MediaMation Rules")
```

```
serial_out(A, StartTheLaserDisc)
```

### **SLIDE ( BYTE1, BYTE2, VALUE1, VALUE2, TIME )**

Slide() is a useful command for the control of MIDI devices. Slide will generate a series of three-byte MIDI commands, sent out at 30-millisecond intervals, according to your parameters. The first two bytes sent are BYTE1 and BYTE2, but the third byte is a value smoothly interpolated from VALUE1 to VALUE2 over a short period of time. As always, TIME is a quantity given in tenths of seconds. For example,

```
slide(176, 0, 127, 0, 25)
```

is cryptic, but what it says to do is generate midi commands every 30 milliseconds for 2.5 seconds of the format 176, 0, <x> where <x> slides smoothly from 127 to 0 over the course of the 2.5 seconds. If the slide is so slow that a value didn't change during the past 30 milliseconds, no value is output.

### **Branch Nodes (IF (<expression>) NODE)**

Branching nodes give the programmer control of the control flow of the show profile. If *expression* is true, program flow will go to NODE. Otherwise, program flow will continue to the regular next node. The first part of a test expression is the name of a variable, usually followed by the test to be performed upon it. If *expression* contains only the variable name, the variable is tested for a nonzero condition, i.e. 1 or more.

The remainder of tests are comparisons of the variable to a value. The value can either be a number, or the name of a second variable can be given and the test will be the first variable's value against the second variable's value.

The allowed tests are:

```
== Value 1 is equal to Value 2
< Value 1 is less than Value 2
> Value 1 is greater than Value 2
!= Value 1 is not equal to Value 2
<= Value 1 is less than or equal to Value 2
>= Value 1 is greater than or equal to Value 2
& Value 1 and Value 2 are both nonzero
```

### **CHANNELIZE**

Channelize takes incoming Midi data an external Midi source, and converts the data from one Midi channel to another. This is handy if you want to have a single input device that can be easily assigned to different output devices, tracks in a MULTITRACK SEQUENCER object, or I/O's.

For instance, let's say you have a device generating data on Midi channel 1 such as a Midi keyboard. Let's also assume that your HRBox unit has digital outputs assigned to Midi Channel 1, 2, and 3.

Depending on how you channelize the incoming data from the Midi Sequence, you can selectively choose what outputs will respond to the keyboard.

Channelize takes 2 arguments: The incoming Midi Channel number, and the Midi Channel number you wish to re-assign it to. Arguments start at "0" which translates to Midi Channel "1" in the outside world. Remember, all data on that channel will remain channelized until you re-assign it back to its normal assignment. You can have several different channelize effects on different channels going on simultaneously.

## **SEQUENCES**

### **SEQUENCER SINGLETRACK NAME**

These are declared at the beginning for each MIDI file you would want to have loaded simultaneously into the playback buffer, each having a different name. Node usage of these is as followed.

*load(name, "show1.mid")*

NOTE: If you issue a load command with no argument, the current sequence will be unloaded and no new data will be loaded. This is similar to erasing the sequence.

*play(name)*

*stop(name)*

*sync (source) // sources are INTERNAL, MTC, or A Declared Pioneer Laser Disk*

### **SEQUENCER MULTITRACK NAME**

Multitrack sequencers are different from singletrack sequencers in that they can also *record* data in real time on different tracks. Each track is permanently assigned to a corresponding Midi Channel. All other commands are the same as for singletrack sequencers.

*record (seq0,1) // sequence name, track to record on*

## **DEVICES**

OBJECTS will give a DVar in the variables window showing the status return of the particular object.

### **MOTIONBASE MOOGRS NAME**

An object made for MOOG motionbases with RS485 control.

Node usage is as follows.

Global commands give commands to all base regardless of the base declared.

**PLAY(NAME)**

Starts motion of all bases. (Global)

**LOAD(NAME, FILE)**

Loads joystick.XXX profile into buffer for playback. You will only declare the 3 end numbers for the file such as "Joystick.600" would be just "600". (Global)

**STOP(NAME)**

Stops the motion of an individual base.

**CMD(NAME, ENGAGE)**

Prepares bases for movement, there must be a few seconds given after this command before giving the run command. (Global)

**CMD(NAME, DEVSTOP)**

Stops the motion of an individual base

**CMD(NAME, RUN)**

Starts motion of all bases. (Global)

**CMD(NAME, FILE, XXX)**

Loads joystick.XXX profile into buffer for playback. You will only declare the 3 end numbers for the file such as "Joystick.600" would be just "600". (Global)

**CMD(NAME, INHIBIT)**

Temporarily deactivates a base making it ignore all further commands except RESET or STAT until the RESET command is received.

**CMD(NAME, RESET)**

Brings a specific base out of inhibit, and brings it out of a fault 2 condition after it reaches home.

**CMD(NAME, DISABLE)**

Disables a specific base permanently and will no longer communicate until the base power is reset.

**CMD(NAME, ESTOP)**

Brings base down under battery power and puts it into a fault 3 state and will need to be re-powered.

**CMD(NAME, STATUS)**

Requests status communication from a peticular base. This signal is also regularly put out to each declared base as soon as the program is booted.

**CMD(NAME, PARK)**

Park will bring all engaged bases down to home position and disengage them. (Global)

## **VIDEO PROJECTOR NAME**

For Various serial controlled devices that can give a status response.

### **CMD(NAME, ASCII, "XXX")**

Sends the specified ASCII string to the named device. XXX is the ASCII characters that correspond to the serial to be sent

### **CMD(NAME, STATUSON)OFF)**

Turns off and on a specified status command to the device to be able to ask a specific question.

## **laserdisk PIONEER a Name**

A device made especially for Pioneer laser disks machines for location polling and status updateing

### **CMD(NAME, STATUSON)**

Starts the HRBox polling the laserdisk about its condition.

### **CMD(NAME, STATUSOFF)**

Stops the polling of the condition.

### **CMD(NAME , QUERY)**

Asks the laserdisk once obot its condition.

### **start(NAME)**

Starts playing the laserdisk.

### **stop(NAME)**

Stops the laserdisk.

### **seek(NAME, 00:00:00:00)**

Searches laserdisks to time in Hours : Minutes : Seconds : Frames.

## **Conclusion**

This concludes the explanation of the ShowFlow programming environment structure. For more information contact our technical department at (310) 320-0696.

ShowFlow<sup>a</sup> ©1995 MediaMation Incorporated, all rights reserved.

2461 West 205th. Street, Ste: B100, Torrance, Ca, 90501 Phone:(310) 320-0696 Fax: 0699