

Multifunction Show Control Showgram User Manual

A. Introduction

Showgram is a flexible scripting language. Once compiled and downloaded to the Multifunction Show Controller (MSC) via the host port, it allows the controller to operate standalone. A Showgram file consists of one or more units called sequence. Sequences can run simultaneously. One sequence can start or stop other sequences. For example, a sequence plays a video loop in response to a “play” button push and another sequence stops the video loop and restarts the video loop at a different frame location in response to a “change language” button push.

A sequence consists of one or more command statements. A command can be an action command, e.g., turn on a relay. It can also be a flow control command, e.g., wait for a button push and then jump to a labeled command. There are also 5 different kinds of variables that the user can define and manipulate, long, byte, bit, string and time. Arithmetic and logical operations can be applied to these variables. Their values can also be tested to provide conditional execution of commands. Variables can also be defined as local to a sequence or global to all sequences. Local variables can only be manipulated within the scope of a sequence. Global variables can be manipulated by all sequences and so provide interactions between sequences. For example, one sequence can be waiting for a bit variable’s state to be changed by another sequence. Relays, opto inputs and tally outputs are just global bit variables with predefined names.

Showgram has a full set of features to make scheduling of events easy. Each controller maintains a system clock and the day of the week. A time bracket can be specified for each sequence, within which the sequence will be allowed to run. Each command can also be time stamped. The command will be executed when the time in the time stamp has arrived or passed. The time stamp can be specified as a time relative to the beginning of the sequence or a system clock time. A command without a time stamp would be executed as soon as possible. A sequence can also be forced into an idle wait state with the *idle for* and *idle until* commands. The system clock can be accessed as a special time variable *now* and the current day of the week as a byte variable *dayofweek*. These variables not only allow execution of commands at a specific time and day of the week, but also calculation of elapsed time between events.

B. Showgram Script Files, Compiler

Showgram script files are plain text files. They can be edited with any text editors or word processors that are capable of saving files in plain text format. Names of all showgram script files must end with an extension of “.msc”. Each script file must be compiled into a binary form for downloading.

The compiler program msc.exe is a 32-bit program that runs inside a DOS window. The command syntax for msc.exe is

```
msc [-l] [-m] [-v18][-v20][-v22] myscript[.msc]
```

The optional argument “-l” creates a listing file showing the pseudo code that will be executed by the show controller. When observing the progress of execution of sequences on the front panel display, the line numbers shown are the pseudo code line number in the listing file. Therefore, the listing file is useful for debugging the script. The “-m” optional argument creates a map file showing the symbol table and memory usage at the show controller level. The “-v18”, “-v20” and “-v22” optional arguments enforce compatibility with earlier version hardware.

The compiler generates 4 files. If myscript.msc is the input file, the output files are

```
myscript.bin  -- download file
myscript.lst  -- listing file generated only when “-l” is specified.
myscript.map  -- map file generated only when “-m” is specified.
myscript.ppf  -- script files after pre-processing.
```

C. Showgram Language Reference

For ease of presentation in the follow sections, we denote keywords of the script language in *italic* form and user defined variables in Courier. Keywords are part of the language and must be used accordingly. All keywords are case insensitive. However, variable names and command labels are case sensitive. Variable names and command labels can consist of any number of characters from the alphabet 'a' to 'z', 'A' to 'Z', the digits 0 to 9 and '_'. The only restriction is that they may not begin with a digit nor '_'. The name of a sequence can be any length. However, only the first eight characters will be transmitted to the show controller. It is therefore advisable to pick sequence names unique to the first eight characters. All command labels must be left justified, i.e., start at the first column, and must be followed by a ':'. There is a predefined label *next* which always refers to the next statement and the actual label *next* must never be defined explicitly.

Extra spaces and tabs may be used anywhere in the script file to improve readability. The compiler automatically ignores them. Wherever space and tab are allowed, a linefeed is also allowed. For example

```
VarA = VarB * (VarC + VarD);
```

and

```
VarA = VarB *  
(VarC + VarD);
```

are two identical command statements. Comments can be added anywhere in the script file by preceding the comment with "/*".

Time is specified in SMPTE format: hh:mm:ss.ff, where hh is hours, mm is minutes, ss is seconds and ff is frames. Preceding zeros are not necessary, i.e., 1:2:3.4 is a valid time string. Time duration can be specified in SMTPE format or in frames as a long integer. For example, 5 seconds can be specified as 0:0:5.0 or 150L where L is used to distinguish it from a byte value. All time strings are converted to number of frames by the compiler and stored internally in both the compiler and the show controller as long integers.

1. Showgram Script File

A script file consists of zero or more global variable declarations followed by one or more sequences.

2. Sequences and Command Statements

There are two possible syntaxes for a sequence:

```
sequence mysequence
{
    // zero or more local variable declarations
    // one or more command statements
}
```

or

```
sequence mysequence : starttime , endtime
{
    // zero or more local variable declarations
    // one or more command statements
}
```

where *mysequence* is the name of the sequence, *starttime* and *endtime* are SMPTE time strings defining the bracket within which the sequence is allowed to run. The pair of braces define the scope of *mysequence*.

There are four general command forms:

```

                command;           // possible comment here
timestamp    command;           // possible comment here
label:       command;           // possible comment here
label: timestamp command;     // possible comment here
```

The label must start at the first column and followed by a ‘:’. The “timestamp” is in SMPTE time format or simply a SMPTE frame number, and the command must be terminated with a ‘;’.

There are five sequence specific commands controlling the behavior of the sequences. They can appear anywhere within the sequence.

a. *autostart*

```
sequence mysequence
{
    autostart;
    // other statements
}
```

The sequence *mysequence* will start automatically everytime the show controller is powered up or after a download.

b. *autorun*

```
sequence mysequence
{
    autorun;
    // other statements
}
```

The sequence *mysequence* will restart automatically after the last statement of the sequence has been executed.

c. *relative*

Time stamps of command statements are compared to the system clock by default. The *relative* command specifies that time stamps are relative to the start time of the sequence. If *autorun* is also specified, the start time is updated whenever the sequence is restarted.

An example of absolute time stamps:

```
sequence mysequence
{
  12:0:1.0 nop;          // runs at 12:00:01 pm
  12:30:0.0 nop;        // runs at 12:30:00 pm
}
```

An example of relative time stamps:

```
sequence mysequence
{
  relative;
  0:0:1.0 nop;          // runs 1 second after sequence started
  0:30:0.0 nop;        // runs 30 minutes after sequence started
}
```

d. *reset time* (version 1.2 and above)

When *relative* is specified for a sequence, the time reference is not restricted to the beginning of the sequence. A new time reference can be specified with the *reset reference* command. For example,

```
sequence mysequence
{
  relative;
  0:0:1.0 nop;          // command1;
  0:0:2.0 nop;          // command2;
  reset reference;
  0:0:2.0 nop;          // command3;
  0:0:4.0 nop;          // command4;
}
```

where *command1* will be executed 1 second after sequence start time and *command3* will be executed 2 seconds after the *reset reference* command is executed.

e. *reference = expression* (version 1.9 and above)

When *relative* is specified for a sequence, the time reference is not restricted to the beginning of the sequence. A new time reference can be specified with the *reference* command. For example,

```
sequence mysequence
{
  relative;
  0:0:1.0 nop;          // command1;
  0:0:2.0 nop;          // command2;
  reference = 90L;      // set time reference to 0:0:3.0;
  0:0:6.0 nop;          // command3;
  0:0:7.0 nop;          // command4;
}
```

```
}
```

where `command1` will be executed 1 second after sequence start time and `command3` will be executed 2 seconds after the `reference =90L` statement. This feature is useful for synchronizing events with video. For example, a video clip is started at frame 1000 and two events are required to happen when the video reaches frame number 2000 and 3000.

```
sequence videosequence
{
    relative;
    // start video at from 1000 here
    ...
    // immediately after the video is started
    reference = 1000L;    // set time reference to 1000 frames
    2000 nop;            // execute event1 here
                        // time stamp is in SMPTE frames
    3000 nop;            // execute event2 here
                        // time stamp is in SMPTE frames
}
```

Sequences can also be manually started and stopped by other sequences. Three commands are available:

```
start mysequence;    // start mysequence from the beginning
stop mysequence;     // stop mysequence after the current statement
pause mysequence;    // same as stop
resume mysequence;   // resume mysequence at the next statement
```

3. Variable Declarations

Variable declarations are of the general form:

```
vartype varname;
```

e.g.,

```
bit bitvar;  
time timevar;
```

When the declaration is specified at the beginning of the script file, outside the scope of any sequence, the variables are global and can be manipulated by all subsequently defined sequences. There are five variable types, *long*, *byte*, *bit*, *string* and *time*.

long – 32-bit integers. *long* constants are numbers with ‘1’ or ‘L’ appended.

byte – 8-bit integers. *byte* constants are numbers without any suffix.

bit – 1-bit logical (boolean) variables. *bit* constants can be specified as 1 or 0, *true* or *false*, *open* (= 1) or *closed* (= 0) and *on* (= 1) or *off* (= 0).

string – ASCII strings of a maximum length of 255 bytes. Constant strings are ASCII characters enclosed in two double-quotes (“”), or in hexadecimal format. For example, “ABC” and 0x414243 define the same string. Hexadecimal format is interpreted two characters at a time except when the length is an odd number. When the length is odd, a zero is automatically inserted after the hexadecimal designator “0x”. For example, 0x12345 would be interpreted as 0x012345. The double-quote (“”) character must be specified in hexadecimal format. So the string A”B must be defined as a concatenation of three strings

```
“A” + 0x22 + “B”
```

time – time strings. Time strings are specified in SMPTE format. The compiler automatically converts all SMPTE strings to long integers representing the totally number of SMPTE frames. Therefore, time and long variables are synonymous.

There are predefined variables that are mapped to the show controller hardware. The bit variables *relay1* to *relay6* refer to the six relays. The read-only bit variables *opto1* to *opto12* refer to the 12 opto-inputs. The bit variables *tally1* to *tally12* refer to the 12 tally outputs. The read-only variable *now* stores the current system clock time and *entrytime* stores the sequence entry time. The read-only variable *dayofweek* stores the current day of the week. The read-only variable *elapsed* stores the number of SMPTE frames elapsed since the start time of the current sequence. When the sequence is in *relative* time mode, the elapsed time is relative to the time reference, which can be affected by the *reference* statements.

The show controller automatically maintains two transition state variables for every bit variable defined. In addition to the 1 or 0 (*true* or *false*, *open* or *closed*, *on* or *off*) states, a bit variable can also be tested for *up* and *down* transitions. An *up* transition corresponds to a change of state from 0 to 1 (*false* to *true*, *closed* to *open*, *off* to *on*) and a *down* transition a change from 1 to 0 (*true* to *false*, *open* to *closed*, *on* to *off*). Whenever a transition occurs, the corresponding transition state variable is set to true. Testing for a transition

resets the corresponding transition state variable to false. Therefore, it is impossible to tell how many transitions have occurred since the last test.

A keypad of size up to 8x8 can be defined using the opto inputs and the tally outputs. Keypad scanning is enabled with the command statement

keyscan M,N

where M and N are numbers between 1 and 8. M denotes the number of tally outputs to be used and N denotes the opto inputs to be scanned. Once keypad scanning is enabled, the state of a key, including the *up* and *down* transition states, at the cross point (m,n) can be tested using the predefined variable *skey(m,n)*.

4. Variable Assignments and Expressions

There are two forms of variable assignment, the simple assignment and the arithmetic assignment. The simple assignment can be of the following forms:

```
variable = expression;  
string-variable = string-expression;
```

where “variable” can be a bit, byte, long or time variable, including all the predefined variables. The script compiler will warn about incompatibility of the type of the variable and the result of the expression. An expression can be a simple expression or a compound expression. A simple expression can be

- i. a timestring
- ii. a long integer
- iii. a byte integer
- iv. *now*
- v. *dayofweek*
- vi. *entrytime*
- vii. *true*
- viii. *false*
- ix. *on*
- x. *off*
- xi. *open*
- xii. *closed*
- xiii. a bit variable
- xiv. one of *tally1* to *tally12*
True if tally is on, false if tally is off.
- xv. one of *opto1* to *opto12*
True if opto is open and false if opto is closed.
- xvi. one of *relay1* to *relay6*
True if relay coil power is on and false if power is off.
- xvii. *skey(m,n)*
True if the key at the cross point (m,n) is open and false if closed.
- xviii. *random(n)*
Returns a random integer between 0 and n.
- xix. *elapsed*
Elapsed time in SMPTE frames since the beginning of the sequence or time elapsed since the time reference. Both day of the week and current time from mid-night are used to calculate the elapsed time. Therefore, Elapsed time cannot be longer than 1 week.
- xx. *elapsed(expression)*
Evaluates the given expression and calculates the elapsed time using the result of the expression as the starting time. Only the current time from mid-night is used to calculate the elapsed time. Therefore, the elapsed time cannot be longer than 2592000 SMPTE frames (1 day). For example,

```
sequence mysequence  
{  
    long oldTime;
```

```

long timeGap;
oldTime = now;           // remember old time
....                    // some statements
timeGap = now - oldTime; // gets negative result if we crossed
                        // mid-night
timeGap = elapsed(oldTime); // takes care of mid-night crossing
}

```

A compound expression is the result of applying arithmetic or logical operations on expressions.

- i. expression1 + expression2
- ii. expression1 - expression2
- iii. expression1 * expression2
- iv. expression1 / expression2
integer division of expression1 by expression2
- v. expression1 % expression2
expression1 modulo expression2
- vi. expression1 & expression2
bitwise and of expression1 and expression2
- vii. expression1 | expression2
bitwise or of expression1 and expression2
- viii. expression1 ^ expression2
bitwise exclusive or of expression1 and expression2
- ix. ~expression1
bitwise negation of expression
- x. expression1 > expression2
- xii. expression1 >= expression2
- xiii. expression1 < expression2
- xiv. expression1 <= expression2
- xv. expression1 == expression2
- xvi. expression1 != expression2
- xvii. (expression)

Use the parentheses to force the order of evaluations of the compound expressions. When parentheses are not used, the precedence of the operators from high to low is

```

~
* / %
+ -
> >= < <=
== !=
&
|
^

```

For example:

```

relay1 = on;           // energize the relay coil
tally2 = on;          // turn on tally2

```

```
bitvar = relay2 | opto3;           // logical-or of relay2 and opto3
timevar = 0:0:30.0;              // set timevar to 30 seconds
timevar = timevar + 0:0:20.0;    // add 20 seconds to timevar;
bitvar = (bytevar == 5) / opto2; // if bytevar is 5 or opto2 is open
```

The “string-expression” is either a constant string or concatenation of two other “string-expressions”. For example:

```
strvar = “abcde”;                // set strvar to “abcde”
strvar = strvar + 0x414243;      // append “ABC” to strvar
```

The arithmetic assignment statements combine an arithmetic operation with the assignment operation. The possible forms are:

```
Variable += expression;         // variable = variable + expression;
Variable -= expression;         // variable = variable - expression;
Variable *= expression;         // variable = variable * expression;
Variable /= expression;         // variable = variable / expression;
Variable &= expression;         // variable = variable & expression;
Variable |= expression;         // variable = variable | expression;
Variable ^= expression;         // variable = variable ^ expression;
Variable %= expression;         // variable = variable % expression;
```

The arithmetic assignment takes less time to execute compared to the expanded statement in the comment section on the right.

5. Opto, Relay, Tally and Keypad Scanning Related Commands

The show controller maintains the up and down state variables for each bit variable defined, including the predefined hardware bit variables. These state variables can be manually cleared in the script.

```
clear relay up;           // clear the up state variable
clear relay down;       // clear the down state variable
clear tally up;         // clear the up state variable
clear tally down;       // clear the down state variable
clear opto up;          // clear the up state variable
clear opto down;        // clear the down state variable
clear skey(m,n) up;     // clear the up state variable
clear skey(m,n) down;   // clear the down state variable
clear bit-variable up; // clear the up state variable
clear bit-variable down; // clear the down state variable
```

where “relay” is one of *relay1* to *relay6*, “tally” one of *tally1* to *tally12*, “opto” one of *opto1* to *opto12*.

The first 8 tallies, *tally1* to *tally8*, can also be switched to PWM mode. The base PWM frequency can be changed from the script. The duty cycle can also be changed individually. The commands are:

```
tally type = pwm;           // set to PWM mode
tally type = simple;       // set it back to tally mode
tally dutycycle = cycle-in-percent; // set the duty cycle to the given percent
pwmfreq = frequency;     // set the PWM base frequency
```

where “tally” is one of *tally1* to *tally8* and frequency is in Hertz. The valid frequency range is between 120Hz and 50000Hz.

Keypad scanning is achieved by enabling a row of keys with a tally output and then read the key states via the opto inputs. By using multiple tally outputs and multiple opto inputs, a keypad of size up to 8x8 can be scanned. By default, keypad scanning is turned off. The command to turn on keypad scanning of size MxN is

```
keyscan M,N;
```

To allow for different kinds of keypad construction and wiring, the tally output state used to enable a row of keys can be selected with the following two commands

```
keyscan high;
keyscan low;
```

Once keypad scanning is enabled, a key at a cross point (m,n) can be accessed and set via the predefined bit variable *skey(m,n)*.

6. Unconditional Jump Statement

There are two kinds of jump statement, unconditional and conditional. There is only one unconditional jump statement

```
jump jumplabel;
```

The next command statement to be executed is the one labeled as jumplabel. jumplabel must be defined within the scope of the current sequence. For example

```
sequence mysequence  
{  
loop:  nop;           // statement1  
       nop;           // statement2  
       nop;           // statement3  
       jump loop;  
}
```

7. Conditional Jump Statement

There are three categories of conditional jump statement:

a. if condition then jump label

If the specified condition is true then jump to the specified label. The available command statements are

```
if bit-variable up then jump jumplabel;  
if bit-variable down then jump jumplabel;  
if relay up then jump jumplabel;  
if relay down then jump jumplabel;  
if tally up then jump jumplabel;  
if tally down then jump jumplabel;  
if opto up then jump jumplabel;  
if opto down then jump jumplabel;  
if skey(m,n) up then jump jumplabel;  
if skey(m,n) down then jump jumplabel;  
if port error then jump jumplabel;  
if day then jump jumplabel;  
if not day then jump jumplabel;  
if expression then jump jumplabel;  
if not expression then jump jumplabel;  
if port-expression then jump jumplabel;
```

where *bit-variable* is a user defined bit variable, “relay” can be one of *relay1* to *relay6*, “tally” one of *tally1* to *tally12*, “opto” one of *opto1* to *opto12* and “port” is one of the 7 serial ports, *port1* to *port7*. “day” is one of *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday* and *Sunday*, and is compared against the current day of the week. There are many possible forms of “expression”. Only “expression” that evaluates to true or false are allowed in conditional jump statements.

A “port-expression” tests the receiver buffer of a specified serial port. The receiver buffer is not modified by the tests.

- i. port == string-expression

Compare the content of the receiver buffer of the given port with the specified string expression. The length of the result of the string expression determines how many characters will be tested. For example:

```
if port1 == "abc" then jump loop;  
if port1 == "cde" + strvar then jump loop;
```

- ii. port == number
- iii. port < number
- iv. port <= number
- v. port > number
- vi. port >= number
- vii. port != number

Compare the number of received characters in the specified port with the given number. For examples,

```
if port1 == 5 then jump loop;  
if port2 >= 5 then jump loop;
```

b. wait for condition then jump label

Wait for the condition to be true then jump to the labeled command. The available form of commands are

```
wait for bit_variable high then jump jumplabel;  
wait for bit_variable low then jump jumplabel;  
wait for bit_variable up then jump jumplabel;  
wait for bit_variable down then jump jumplabel;  
wait for relay on then jump jumplabel;  
wait for relay off then jump jumplabel;  
wait for relay up then jump jumplabel;  
wait for relay down then jump jumplabel;  
wait for tally on then jump jumplabel;  
wait for tally off then jump jumplabel;  
wait for tally up then jump jumplabel;  
wait for tally down then jump jumplabel;  
wait for opto open then jump jumplabel;  
wait for opto closed then jump jumplabel;  
wait for opto up then jump jumplabel;  
wait for opto down then jump jumplabel;  
wait for skey(m,n) open then jump jumplabel;  
wait for skey(m,n) closed then jump jumplabel;  
wait for skey(m,n) up then jump jumplabel;  
wait for skey(m,n) down then jump jumplabel;  
wait for relay on;  
wait for relay off;  
wait for relay up;  
wait for relay down;  
wait for tally on;  
wait for tally off;  
wait for tally up;  
wait for tally down;  
wait for opto open;
```

```

wait for opto closed;
wait for opto up;
wait for opto down;
wait for skey(m,n) open;
wait for skey(m,n) closed;
wait for skey(m,n) up;
wait for skey(m,n) down;

```

These are blocking commands in the sense that the sequence will be stuck at these statements until the test conditions become true. The short forms are equivalent to the corresponding long forms with `jumplabel` set to the predefined label `next`, meaning proceeding to the next statement if the condition is tested true. The short form for bit variables is not supported.

c. if condition in timevar then jump label

Jump to the specified labeled command whenever the condition becomes true within the period of time specified in “timevar”. Go to the next command statement if the condition stays false throughout the specified period.

```

if bit_variable high in timevar then jump jumplabel;
if bit_variable low in timevar then jump jumplabel;
if bit_variable up in timevar then jump jumplabel;
if bit_variable down in timevar then jump jumplabel;
if relay on in timevar then jump jumplabel;
if relay off in timevar then jump jumplabel;
if relay up in timevar then jump jumplabel;
if relay down in timevar then jump jumplabel;
if tally on in timevar then jump jumplabel;
if tally off in timevar then jump jumplabel;
if tally up in timevar then jump jumplabel;
if tally down in timevar then jump jumplabel;
if opto open in timevar then jump jumplabel;
if opto closed in timevar then jump jumplabel;
if opto up in timevar then jump jumplabel;
if opto down in timevar then jump jumplabel;
if skey(m,n) open in timevar then jump jumplabel;
if skey(m,n) closed in timevar then jump jumplabel;
if skey(m,n) up in timevar then jump jumplabel;
if skey(m,n) down in timevar then jump jumplabel;

```

For example:

```

sequence blink
{
    time timevar;
    // we blink tally2 on for half a second and off half a second
    // until we detect a button push at opto3 input
    timevar = 15L;    // set timevar to 1/2 of a second
loop:

```

```
// wait for 1/2 second unless opto3 is pushed. If so, jump.  
if opto3 down in timevar then jump pushed;  
tally2 = ~tally2; // invert the state of tally2  
jump loop;  
pushed:  
    tally2 = false; // turn off the tally light  
}
```

8. Serial Port Commands

There are eight serial ports available from the show controller. One of the ports is the host port and is reserved for interactive communication with a computer. The computer connected to the host port is responsible for downloading the compiled script to the show controller. The remaining seven ports can both be controlled via the host port and via the scripts. Script commands only apply to the seven non-host ports.

a. Port Definitions

There are currently three different built-in port protocols, plain ASCII, Pioneer laser disk, and external variable (version 1.6 and above). The port protocol is specified with

```
port type = ascii;  
port type = ld;  
port type = nbytes;
```

where “port” is one of *port1* to *port7*. For examples,

```
port1 type = ascii;  
port5 type = ld;
```

There is no built-in processing of data sent to and received from an ASCII port. The script is fully responsible for manipulating the data. Pioneer laser disk control protocol is built into the show controller. The laser disk control script commands are described in the next section.

The parameter “nbytes” can be one of *int1*, *int2*, *int3* and *int4*. The *intN* protocol is a custom protocol that allows exchange of up to 16 integer values between the show controller and an external serial device. For the *intN* protocol $N \times 7$ -bit of data will be exchanged. The show controller maintains 16 predefined long variables *extvar(m,n)* for each *intN* port, where *m* is the port number, 1 to 7, and *n* specifies which integer variable, 1 to 16. A basic *intN* packet has the following form

```
varaddr first-byte ... Nth-byte checksum
```

where “varaddr” has a value between 0x80 to 0x8F. The second nibble specifies which integer variable. Each of the following *N* bytes defines 7 bits of the final value. The MSB of these bytes are zero. The last byte is the sum of all the preceding bytes with the MSB masked. As far as the show script is concerned, the port communication is all hidden. Assigning a value to an external variable *extvar(m,n)* automatically triggers the show controller to send a packet to the external device at that port. Similarly, the *extvar(m,n)* is continuously updated by new packets received from the external device.

b. Serial Port Parameters

The generic port communication parameters can be set with the following script commands.

```
port baudrate = number;  
port parity = parity;  
port wsize = wordsize;
```

```
port stopbit = stopbits;
```

The allowable baudrates are 300, 1200, 2400, 4800, 9600, 19200, 31250, 38400, 57600, 125000, 256000. The allowable parity is “none”, “high”, “low”, “even” and “odd”. The allowable word sizes are “5BIT”, “6BIT”, “7BIT” and “8BIT”. The allowable stop bits are “ONE”, “ONE&HALF” and “TWO”.

b. Port Transmit and Receive

Data can be transmitted to a port irrespective to whichever protocol it is configured to:

```
send string-variable to port;  
send string-expression to port;
```

The show controller automatically consumes data received by Pioneer laser disk ports and ports running the *intN* protocol. For an ASCII port, the data received can be tested as described in section 7 above.

c. Port Management

Each port can be reset with the reset command:

```
reset port;
```

Resetting the port resets the serial communication hardware and clears both the receive and transmit buffers. However, the configuration of the port stays the same. The receive buffer can also be emptied via the script:

```
empty number-of-bytes from port;  
empty port;
```

The first form empties the specified number of bytes from the given “port”. The second port removes all the received bytes.

Port receive and transmit can be put on hold for the purpose of synchronizing control of multiple ports. For example, multiple Pioneer laser disk ports can be put on hold, a video play command is then sent to each port, the ports are then released so that all the ports receive the play command within the same video frame. The syntax of the commands are:

```
hold portlist;  
release portlist;
```

where “portlist” is a comma separated list of ports, *port1* to *port7*. For example

```
hold port2,port3,port5; // Hold ports 2, 3 and 5  
ldplay port2; // issue the play command to port 2  
ldplay port3; // issue the play command to port 3  
ldplay port5; // issue the play command to port 5  
release port2,port3,port5; // send the play commands out
```

9. Video Laser Disk Control Commands

Currently only Pioneer laser disk protocol is implemented. There are two categories of script commands, synchronous and asynchronous. An asynchronous command sends the command to the laser disk and returns the control back to the script. A synchronous command would not return control until the response from the laser disk has been received. The available synchronous commands are

```
ldstart port;           // spin up the laser disk
ldstop port;           // stop and spin down the laser disk
ldpause port;         // pause the player and blank the screen
ldstill port;         // freeze the player at the current frame
ldplay port;          // start playing
ldplay port to expression; // play until the frame specified by the expression
ldplay port to string-expression; // play to the location specified by the string
ldsearch port to expression; // search to the frame specified by the expression
ldsearch port to string-expression; // search to the location specified by the string
```

The string-expression form is used to address non-CAV and compact discs. The corresponding synchronous commands are

```
wait for ldstart port;
wait for ldstop port;
wait for ldpause port;
wait for ldstill port;
wait for ldplay port;
wait for ldplay port to expression;
wait for ldplay port to string-expression;
wait for ldsearch port to expression;
wait for ldsearch port to string-expression;
```

For examples

```
wait for ldsearch port1 to 12345L; // search to frame 12345
ldplay port1; // start playing
idle for 300L; // idle and wait for 10 seconds
wait for ldstop port1; // stop the player
```

There are also 7 predefined variables *vframe1*, *vframe2*, ..., and *vframe7*. When the player N is in play, stilled or paused mode, *vframeN* is the current video frame location. Otherwise, *vframeN* will be zero.

10. Show Controller Display

Progress of execution of sequences can be monitored from the front panel of the show controller. Sequences can also send specific strings to be displayed on the user page of the front panel. The available display commands are

```
display string-expression to line1;      // display to line1  
display string-expression to line2;      // display to line2  
display variable to line1;                // display long or byte variables  
display variable to line2;                // display long or byte variables
```

All commands start displaying at column one and fill the remaining of the line with space. The following forms are only available in versions 1.6 and above. It allows display to start at a specified “position” of the given line and it would not blank the remaining of the line.

```
display string-expression to line1 at position;  
display string-expression to line2 at position;  
display variable to line1;  
display variable to line2;
```

Device serial port receive buffer can also be displayed on the front panel. The commands are:

```
display port to line;  
display n-characters from port to line;  
display port to line at position;  
display n-characters from port to line at position;
```

where “port” is one of *port1* to *port7*, “line” is *line1* or *line2*, “n-characters” is the number of characters from the receive buffer and “position” is between 1 and 20. The commands with “position” specified only available in versions 1.6 and above. Displaying the receiver buffer leaves the buffer untouched.

11. Time Commands

When time stamps are not used, script commands are available for explicit event scheduling.

```
idle for expression;           // wait for a given period  
idle until expression;       // wait until the specified time  
idle until nextday;         // wait until next day  
framesync;                   // wait until the next video frame
```

Here expression can be an explicit time in SMPTE format, a long integer constant, a long variable, or an arithmetic expression that results in a long integer. For examples,

```
idle for 10L;                 // wait for 10 frames;  
idle for 0:30:0.0;           // wait for 30 minutes;  
idle until 12:30:0.0;        // wait until 12:30pm
```

12. Host Messages (version 1.6 and above)

Host messages can be sent back to the host computer via the host port. The messages can serve as an error log to be saved by the host computer. Internally, the show controller maintains a 256-byte message buffer. The host must actively retrieve message. If there is no host computer connected to the host, messages sent to the message buffer may overflow the buffer. Once overflow occurred, new host messages would be discarded until the buffer is emptied. The available host message commands are

```
send string-variable to host;  
send string-expression to host;  
empty host messages; // actively empty the message buffer
```

13. Miscellaneous Commands

a. Null Command

```
nop; // just a place holder
```