Integrated Media Server Programming Manual

V5.0 2. August 2011

Granite Precision Inc.
2305 Alpen Ct. Ste. 201
Pine Mountain Club
CA 93222-5476

The Integrated Media Server is dedicated to the memory of Walt Disney Imagineer H.F. Crane. Without his interest, enthusiasm, and evangelism in the early days, this product would never have gotten off the ground.

The positive attitude H.F. showed from the first moment he was diagnosed with Stage IV cancer was amazing. Yet at the same time, it matched how he approached everything in life.

This document was written with Sun Microsystems OpenOffice.org 1.1.2

Revision History

v1.0	15. August 2004	Initial Rev
v1.1	9. October 2004	Small changes to video section
v2.0	4. February 2005	Audio revisions
v3.0	15. May 2005	SMPTE revisions

Overview

The Integrated Media Server Platform ™ can come in many different physical hardware arrangements. Some of these arrangements are not reprogrammable. For the rest of the arrangements, programming is done mainly via our Braintrust ™ software.

Using Braintrust, you will create a show, save it, upload it to the Integrated Media Server (IMS) and watch it run.

Using Braintrust

Braintrust can be run on the IMS, or it can be run on any other Windows 2000/XP computer on an Ethernet TCP/IP network attached to the IMS. We are working on a Windows 7 version currently, but it will work in Windows 7 XP compatibility mode. Most of its functions are offline functions, which means that you do not need to be connected to the IMS. Some of its functions are online functions, which do require you to be connected. To be "connected" means to be both physically wired to the IMS, and to be logically connected inside the software. See the next section on how to connect.

Connecting is only mandatory *at the time of* synchronizing the show.

Connecting to the Integrated Media Server

To connect to the IMS, make sure it is up and running. Run the Braintrust software. If the correct IP address is not already set, select Tools | IMS Network Settings. Simply put the IP address of the IMS here. If Braintrust is running on the IMS server itself, you may enter the loopback IP address – 127.0.0.1. To officially connect, select either Tools | Connect to IMS, or click the "Connect" toolbar button. If you do not receive an error message, and the status bar displays "Connected", then you are, in fact, connected. You will also see a Connect message in the Log Window. You may then disconnect by exiting the program, selecting Tools | Disconnect from IMS, or clicking the "Disconnect" toolbar button.

Checking the Syntax of Your Show

Before uploading your show, you may check its syntax. This operation ensures that the tasks and commands that you have created are valid and understandable by the IMS. This step is not required, as the syntax will automatically be checked when you attempt to synchronize this show with an IMS, if it has not been checked before. The syntax check functionality simply gives you an opportunity to debug your script when your PC is not connected to an IMS.

To check the syntax, select Tools | Check Syntax… or click the "Check Syntax" toolbar button. Any errors should appear in the Output Window. If there are no errors, a message

will appear within the Output Window stating that "no errors were found in the syntax". If there are errors, you may read the errors and find the problem on your own, or you may click on the error to be taken directly to the offending task and/or command.

Synchronizing Your Show with the IMS

Once connected to an IMS, the show may be transferred from your PC by choosing Tools | Synchronize… or by clicking the "Synchronize" toolbar button. If the syntax has been checked and was found to be valid since the last time changes were made to the show, the show will be instantly transferred to the IMS and it will begin execution. If the syntax has not been checked recently, it is automatically checked. If there are errors, the synchronization process is halted and the programmer may click on the errors within the Output Window to address them. If no errors are found in the syntax, the show is transferred automatically to the IMS, and it begins execution.

Programming Concepts

Programming is done with Braintrust.

Scripting

In Braintrust, you create real-time tasks to accomplish goals, and individual commands within the task to fulfill those goals.

Resources

The Integrated Media Server is open-ended when it comes to resources. That means essentially two things:

1) The IMS can have an infinite amount of *types* of things in it
2) The IMS can have almost an infinite (for all intents and purposes) *number* of those things.

The IMS is made to be extremely scalable.

Notes:

1) Don't worry if you don't know what each of these resources is at the moment. They'll be discussed individually later.
2) The list below states the resources the IMS supports and the maximum number of those resources. However, you could not have the maximum number of all of those resources in one IMS, but if all you put in one IMS was one type of resource, then the maximum would apply!
3) We'll also list the name of the resource in Braintrust that you'll be using. We call that the Braintrust Identifier.
4) Some resources have inputs and outputs that are physically grouped together on one port, such as a serial port, which is one port, but has both an incoming data stream and an outgoing data stream. In some cases, as with the serial port, we still distinguish the input differently than the output for simplicity while programming.
5) In some cases with externally interfaced resources, the maximum number is not tied to how many can be physically put inside the IMS, such is as listed below, and, therefore, the maximum is truly our ultimate internal maximum of 4,294,967,295.
6) Built-in Resource types are included with every IMS.

Built-In Resource Types	Braintrust Identifier	Maximum Number
TASKS	task	4,294,967,295
VARIABLES	var	4,294,967,295

Optional Resource Types	Braintrust Identifier	Maximum Number
DIGITAL INPUTS	diginput	1024
DIGITAL OUTPUTS	digoutput	1024
ANALOGUE INPUTS	anainput	128
ANALOGUE OUTPUTS	anaoutput	64
SERIAL PORTS	serport	128
ETHERNET PORTS	ethport	32
SMPTE LTC INTERFACES	smpte	16
MIDI INPUTS	midiinput	16 (256 channels)
MIDI OUTPUTS	midioutput	16 (256 channels)
DMX INPUTS	dmxinput	32 (16,384 channels)
DMX OUPUTS	dmxoutput	32 (16,384 channels)
DISPLAY	display	32
SYNAPSE CONSOLE SCREENS	screen	16,384
BUTTONS	button	1024
VIDEO DECODER SOURCES	vidsource	16
VIDEO DECODER OUTPUTS	vidoutput	16
AUDIO CHANNEL SOURCES	audsource	96
AUDIO CHANNEL OUTPUTS	audoutput	96

Resources are listed by their type, and index (number). The first one is always 1, and the last one is the largest number of that resource type you have in the system. Tasks and Variables are internal resources that can number up to 4,294,967,295. Here are some examples:

var:1
var:2
var:8
var:10
var:19
var:92
var:100
var:1000
var:10000
var:100000
var:4294967295

External resources depend on the physical quantity installed in your IMS. For example, if you have one 8 port rs-232 serial card installed, you can access:

serport:1- serport:8

there is NO serport:9

However, if you install *two* 8 port rs-232 serial cards, then the amount doubles:

serport:1 – serport:16

Some commands let you set a *range* of resources:

```
00:00:00:01    On digoutput:1-3
```

This allows you to avoid typing the same thing in multiple times:

```
00:00:00:01    On digoutput:1
00:00:00:01    On digoutput:2
00:00:00:01    On digoutput:3
```

Resource Objects

Sometimes it's useful to utilize a portion or a property of a resource. Most resources have some sort of property that can be changed or viewed. This is denoted with the dot "." between the resource and its object.

```
00:00:00:01    Set var:1 serport:1.inbuffercount
```

Sometimes there is more than one resource property with the same name, and you have to list an index (just like resource indices)

```
00:00:00:01    Write display:1.row:1 "displayed on row 1"
```

Sometimes you have to work with more than one resource property at a time. To do this, you use a *range*, just like with resource indices.

```
00:00:00:01    Delete serport:1.inbuffer:2-6
```

The more specialized the operation, the more you drill down into the resource. We call properties of resources objects. The resource may have more than one object, and that object may have its own properties (object).

Tasks

Tasks are short programs that manipulate resources. Tasks are like recipes, and commands are like the individual steps in a recipe. You could program the simplest show in the world with just one task with one command in it. You don't have to have more, but it makes life a lot easier when you can group things into tasks, especially if you're going to be doing that "task" more than once. The other neat thing about tasks is that they can run simultaneously, synchronously, or asynchronously, so very complex things can be accomplished.

General Task Usage

Tasks are your way to manipulate resources. Start a task whenever you need to make changes in the system. Normally tasks should be sectioned off to logical steps in your overall program. For example, using the recipe metaphor, if you're cooking several things for dinner, you might want to group together the details of cooking each item together:

- Dinner -
Cook Entree
Cook Side Dishes
Cook Dessert

You might want to further subdivide your cooking into even smaller tasks:

- Cook Entree -
  Prepare Foods for this entree
  Cook Foods for this entree

These smaller tasks could be divided further into subtasks, or enacted through commands (instructions) right there in those tasks. It's up to you, and how complex your show is:

- Prepare Foods for this entree -
  Preheat Oven
  Chop Onions
  Chop Celery
  Crumble Bread
  Mix in Hamburger
  Mix in Sausage
  Mix in Egg

- Cook Foods for this entree -
  Put Meatloaf Mixture in Oven
  Cook for 45 minutes on low temperature

Task Time

Each running task has its own clock which runs independently of all the other tasks clocks that are running. Tasks that are running are dormant and do not have their clocks running. There is no central clock in the IMS system. The chronological time and date are available to the system, but they do not fundamentally affect the running systems' time-keeping mechanisms. A **Stopped**(not running task) has its clock reset to 00:00:00:00. When a task is started from being stopped (regardless of the method of starting it), its clock begins to run. The clock advances one frame at a time. When the running clock reaches the framerate set in the IMS, it advances by 1 second, sets the frame to 0, and continues. This is similar to how normal clocks work (for example, the time after 00:59 is 01:00). In an IMS with a clock set to run at 30 frames per second (fps), the following example is an accurate measure of time:

```
00:00:00:28 Write display:1.row:1 "hello world"
00:00:00:29 Write display:2.row:1 "hello world2"
00:00:01:00 Write display:3.row:1 "hello world3"
```

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

MainTask
02
03
04
05
06
07
08
09

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 00:00:00:28 Write display:1.row:1 “hello world”
30 00:00:00:29 Write display:2.row:1 “hello world2”
31 00:00:01:00 Write display:3.row:1 “hello world3”

The running clock looks at the times listed in the task, and when the running clock's time matches a time listed in the task, the command at that time is executed. Many commands may reside on the same time, and the running clock will execute all of them on the same frame (provided they are internal or *intrinsic* commands. External or *extrinsic* command times are explained momentarily). The running clock never goes backward. Unfortunately, as much as we'd all like to make Time go backward, it can't. This is a crucial concept with the IMS. If the running clock encounters a time that is less than the current time, it will execute it immediately, as if it were listed as the current time.

```
00:00:00:01 Write display:1.row:1 “hello world”
00:00:00:05 Write display:2.row:1 “hello world2”
00:00:00:01 Write display:3.row:1 “hello world3”
```

This is how the previous example would behave:

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

MainTask
02 00:00:00:01 Write display:1.row:1 “hello world”
03 00:00:00:02
04 00:00:00:03
05 00:00:00:04
06 00:00:00:05 Write display:2.row:1 “hello world2”
06 00:00:00:01 Write display:3.row:1 “hello world3”

See that the third command is executed on the same frame at the second command?

The running clock works the same when a Goto command interrupts timeflow:

```
00:00:00:01 Write display:1.row:1 “hello world”
00:00:00:05 If var:1 = 1 Goto NextStep
00:00:00:10 Write display:2.row:1 “hello world2”
00:00:00:01 End
00:00:00:15 Label NextStep
00:00:00:03 Write display:3.row:1 “hello world3”
```

This is how the previous example would behave if var:1 = 0

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

MainTask
02 00:00:00:01 Write display:1.row:1 “hello world”
03 00:00:00:02
04 00:00:00:03
05 00:00:00:04
06 00:00:00:05 if var:1 = 1 Goto NextStep (var:1 = 0, so continue)
07
08
09
10
11 00:00:00:10 Write display:2.row:1 “hello world2”
11 00:00:00:01 End
MainTask Stopped

This is how the previous example would behave if var:1 = 1

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

MainTask
02 00:00:00:01 Write display:1.row:1 “hello world”
03 00:00:00:02
04 00:00:00:03
05 00:00:00:04
06 00:00:00:05 if var:1 = 1 Goto NextStep (var:1 = 1, so jump to NextStep
07
08
09
10
11
12
13
14
15
16 00:00:00:15 Label NextStep
16 00:00:00:03 Write display:3.row:1 “hello world3”
MainTask Stopped

No matter what happens, the running clock of a task will not go backward or back-up for any reason, even for a task with a goto command that jumps backward in the command list.

```
00:00:00:01 Label Loop
00:00:00:01 Write serport:1 “hello world”
00:00:00:03 If var:1 <> 1 Goto Loop
```

This is how the previous example would behave if var:1 = 1 on the fourth time through the loop.

Absolute Time (in frames)

01		MainTask is started (from somewhere else)
	MainTask	
02	00:00:00:01	Label Loop
02	00:00:00:01	Write serport:1 “hello world”
03		
04	00:00:00:03	If var:1 <> 1 Goto Loop (var:1 is 0 so loop)
05	00:00:00:01	Label Loop
05	00:00:00:01	Write serport:1 “hello world”
06		
07	00:00:00:03	If var:1 <> 1 Goto Loop (var:1 is 0 so loop)
08	00:00:00:01	Label Loop
08	00:00:00:01	Write serport:1 “hello world”
09		
10	00:00:00:03	If var:1 <> 1 Goto Loop (var:1 is 0 so loop)
11	00:00:00:01	Label Loop
11	00:00:00:01	Write serport:1 “hello world”
12		
13	00:00:00:03	If var:1 <> 1 Goto Loop (var:1 is 1 so continue)
		MainTask Stopped

Extrinsic commands are those that take longer than one frame to execute. Often these commands deal with external resources, but not always. Commands that are extrinsic are listed as such.

An extrinsic command causes the task to pause until the command completes. If the command does not complete due to an error or if the command times out, the command is aborted, and the task is stopped immediately.

The following task uses the extrinsic command **Queue**, and the time after the command occurs varies with how long the extrinsic command took to complete:

```
00:00:00:01 Queue vidsource:1 “c:\media\mympegfile.mpg”
00:00:00:05 Start vidsource:1
```

It may be that the extrinsic command takes only a few frames to complete, or it may take several minutes, depending on the command.

Absolute Time (in frames)

01	MainTask is started (from somewhere else)	
	MainTask	
02	00:00:00:01	Queue vidsource:1 “c:\media\mympegfile.mpg” MainTask Paused waiting for completion of Queue command
03		
04		
05		
06		
07		
08		Queue command completes here. MainTask Re-Started from here.
09		
10		
11		
12	00:00:00:05	Start vidsource:1

Note that

`00:00:00:05 Start vidsource:1`

did not occur 5 frames after the start of the task as would normally occur with intrinsic commands, but rather, 5 frames after the start of the task *PLUS* the length of time it took the extrinsic command to complete.

Task Commands

The following commands apply to tasks:

`Start`
`Stop`
`Pause`
`Call`
`Label`
`If`

Start

Start <task> <params>

Starts a task directly from another task. Now, both tasks are running.

– MainTask -
`00:00:00:10 start SubTask`

When this new task is started, the original task continues execution.

– MainTask -
`00:00:00:10 start SubTask`
`00:00:00:15 Write serport:1 “hello”`

– SubTask -

```
00:00:00:05 Write serport:2 “hello”
```

For this example, the word “hello” will be sent out both serial port 1 and serial port 2 at the EXACT same time.

The task started will start execution on the following frame, regardless of what task you start, or how many.

See the below timelisting for an exact view:

Absolute Time (in frames)

01	MainTask is started (from somewhere else)			
	MainTask		**SubTask**	
02	00:00:00:01			
03	00:00:00:02			
04	00:00:00:03			
05	00:00:00:04			
06	00:00:00:05			
07	00:00:00:06			
08	00:00:00:07			
09	00:00:00:08			
10	00:00:00:09			
11	00:00:00:10	SubTask Started, and MainTask continues		
12	00:00:00:11		00:00:00:01	
13	00:00:00:12		00:00:00:02	
14	00:00:00:13		00:00:00:03	
15	00:00:00:14		00:00:00:04	
16	00:00:00:15	Data “hello” sent out serport:1 MainTask Stopped	00:00:00:05	Data “hello” sent out serport:2 SubTask Stopped

The start command is special. You do not actually need to use the word 'Start' ! As long as there is a task with the name you have listed present, the IMS will understand you want to start it. This is useful for times in which you want to make your tasks seem like commands themselves:

Instead of:
```
00:00:00:10 Start SubTask
```

You only need:
```
00:00:00:10 SubTask
```

When you **Start** (or **Call**) a task, you can pass parameters. This allows you to provide information to a task without having to put data in a variable first. The data is accessed via the **param** resource. The param resource acts just like a variable resource, but is specific to the task *at the time* of being called. The **param** resources are replaced with whatever resource is put after the task to **Start**, in order.

- Add -

```
00:00:00:01    SetVar var:1 param:1
00:00:00:01    AddVar var:1 param:2

00:00:00:01    Start Add 9 3
```

The result of starting this task is that var:1 will contain **12**.

```
00:00:00:01    Set var:2 “Tyler Durden”
00:00:00:01    Start Add “And that's how I met “ var:2
```

The result of starting this task is that var:1 will contain “And that's how I met Tyler Durden”

Stop

Stop <task>

Stopping a task is as easy as starting it. When you stop a task, it will cease execution on the following frame, regardless of what task you stop, or how many. Once you stop it, you can start it again, and it will start from the beginning.

```
00:00:00:10 Stop SubTask
```

Pause

Pause <task>

Pausing a task causes it to stop execution like **Stop** does, but unlike **Stop**, when you **Start** it again, it starts from the command that you paused it on. When you pause a task, it will cease execution on the following frame, regardless of which task you pause, or how many.

Call

Call <task> <params>

Calling a task is exactly the same as starting a task, except the original task pauses until the subtask completes.

- MainTask -

```
00:00:00:10 call SubTask
00:00:00:15 Write serport:1 “hello”
```

- SubTask -

```
00:00:00:05 Write serport:2 “hello”
```

In this example, “hello” is sent out serial port 2, and 5 frames later “hello” is sent out serial port 1. See the below time layout for an exact view:

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

	MainTask	**SubTask**
02	00:00:00:01	
03	00:00:00:02	
04	00:00:00:03	

05	00:00:00:04			
06	00:00:00:05			
07	00:00:00:06			
08	00:00:00:07			
09	00:00:00:08			
10	00:00:00:09			
11	00:00:00:10	SubTask Started, and MainTask Paused		
12			00:00:00:01	
13			00:00:00:02	
14			00:00:00:03	
15			00:00:00:04	
16			00:00:00:05	Data “hello” sent out serport:2 SubTask Stopped MainTask Re-Started
17	00:00:00:11			
18	00:00:00:12			
19	00:00:00:13			
20	00:00:00:14			
21	00:00:00:15	Data “hello” sent out serport:1 MainTask Stopped		

If you call a task on the last command of the current task, the call command is effectively turned into a start command.

Trigger

Trigger<expression> <task> <params>

Triggers are used to start tasks at arbitrary times, when some event occurs, rather than at a pre-determined time. The task will begin when the expression listed is true. In order for the trigger to happen a second or subsequent time, the expression must go false, then true again for another trigger to happen. This behavior is called being *edge-triggered*.

```
00:00:00:01 Trigger button:1 = ON MyTask
```

In the above example, the task MyTask will start when button:1 becomes ON. It is possible and sometimes common to have multiple triggers for one task:

```
00:00:00:01 Trigger button:1 = ON MyTask
00:00:00:01 Trigger button:3 = OFF MyTask
00:00:00:01 Trigger var:3 > 10 MyTask
00:00:00:01 Trigger dmxinput:[var:391] <> anaoutput:21 MyTask2
00:00:00:01 Trigger vidsource:1.status = VIDEOERROR MyTask2
```

All of the above triggers will start the specified task when their expression is found to be true. You can also start multiple tasks with the same expression:

```
00:00:00:01 Trigger serport:19.inbuffercount > 3 MyTask
00:00:00:01 Trigger serport:19.inbuffercount > 3 MyTask2
```

although it would be easier and clearer to instead have one trigger start MyTask and have MyTask start MyTask2 as its first command.

Here is an in-depth analysis of an expression. An expression compares two different things together. You can compare a resource with another resource, or a resource with a literal value. There are no restrictions to the resources that you can compare. The comparison matches the value of each side at the moment in question. With these values the comparison is made. Here are some various resources and literals being compared to see if they are equal:

var:1 = var:2
var:1 = 21
diginput:1 = 1
diginput:1 = var:1
dmxoutput:921 = dmxinput:129
screen:1.mousedown = anaoutput:13
var:1 = 123.456
var:1 = "hello world"
vidsource:12.status = "VIDEOSTOPPED"

The comparison can be equal, not equal, less than, greater than, less than or equal to, greater than or not equal to. Here are some examples of comparison with different operators for the same two resources. You may use either the BASIC language symbols for comparison, or the C language symbols, and you may mix-and-match. These first examples use the BASIC language symbols:

```
var:1 = 3       // equal
var:1 <> 3      // not equal
var:1 < 3       // less than
var:1 > 3       // greater than
var:1 <= 3      // less than or equal to
var:1 >= 3      // greater than or equal to
```

Here are the same examples using the C language symbols:

```
var:1 == 3      // equal
var:1 != 3      // not equal
var:1 < 3       // less than
var:1 > 3       // greater than
var:1 <= 3      // less than or equal to
var:1 >= 3      // greater than or equal to
```

RemoveTriggers

RemoveTriggers <task>

or

RemoveTriggers ALL

Use RemoveTriggers to "turn off" triggers for the specified task.

```
00:00:00:01 RemoveTriggers MyTask
```

When you want all tasks to have their triggers removed, use:

```
00:00:00:01 RemoveTriggers All
```

Note that if you have a Task named "All" and then issue the command

```
00:00:00:01 RemoveTriggers All
```

RemoveTriggers will not realize you specified a task, and will remove all triggers from all tasks.

End

End

The End command stops execution of the current task immediately.

```
00:00:00:01 Write display:1.row:1 "hello world"
00:00:00:03 End
00:00:00:05 Write display:1.row:2 "goodbye cruel world"
```

The above would execute in this manner:

Absolute Time (in frames)
01 MainTask is started (from somewhere else)

MainTask
02 00:00:00:01 Write display:1.row:1 "hello world"
03
04 00:00:00:03 End
MainTask stopped

Label

Label <labelname>

The label has two different purposes. It is most commonly used when you want to mark a placeholder in your task to "jump" to from an IF command or a GOTO command. When you jump to a label, you jump over any commands in between the current execution point, and the label, and begin executing commands from the label down.

```
00:00:00:01 Write serport:1 "hello"
00:00:00:01 goto DoThis
00:00:00:01 Write serport:2 "hello" (never happens!)
00:00:00:01 Write serport:3 "hello" (never happens!)
00:00:00:01 Label DoThis
00:00:00:01 Write serport:4 "hello"
```

The second purpose is simply to have a highlighted comment between commands, and a label will do that nicely for you.

Goto

Goto <label>

It's hard to explain a Label without a Goto, and it's hard to explain a Goto without a label, so here's the other half of what you need to know. A goto statement tells the IMS to stop executing the current set of commands, and instead jump to a new place in the task. Time continues to run *forward*.

```
- MainTask -
00:00:00:01 set var:7 3 // init var 7 to 3 (loop 3 times)
00:00:00:01 label loop // loop starting here
00:00:00:01 if var:7 = 0 end // if var 7 has reached 0 get out
00:00:00:01 on digoutput:1 // turn on lite 1 (start flash)
00:00:00:05 off digoutput:1 // turn off lite 1 (stop flash)
00:00:00:05 subvar var:7 1 // subtract 1 from var 7
00:00:00:05 goto loop // loop here!
```

Here **Goto** is used to generate a loop within a task. Note that this is a jump *backward* in the command list. Whenever you jump backwards, the label is jumped to on the *next* frame. This is to prevent infinite recursion on one frame.

Absolute Time (in frames)

01 MainTask is started (from somewhere else)

MainTask

02 00:00:00:01 set var:7 3
02 00:00:00:01 label loop
02 00:00:00:01 if var:7 = 0 end (var:7 is 3, so continue)
02 00:00:00:01 on digoutput:1
03
04
05
06 00:00:00:05 off digoutput:1
06 00:00:00:05 subvar var:7 1 (var:7 is now 2)
06 00:00:00:05 goto loop
07 00:00:00:01 label loop
07 00:00:00:01 if var:7 = 0 end (var:7 is 2, so continue)
07 00:00:00:01 on digoutput:1
08
09
10
11 00:00:00:05 off digoutput:1
11 00:00:00:05 subvar var:7 1 (var:7 is now 1)
11 00:00:00:05 goto loop
12 00:00:00:01 label loop
12 00:00:00:01 if var:7 = 0 end (var:7 is 1, so continue)
12 00:00:00:01 on digoutput:1
13
14
15
16 00:00:00:05 off digoutput:1
16 00:00:00:05 subvar var:7 1 (var:7 is now 0)
16 00:00:00:05 goto loop
17 00:00:00:01 label loop

17 00:00:00:01 if var:7 = 0 end (var:7 is 0, so end task immediately)
MainTask Stopped

The example above loops 3 times of 5 frames each, exactly as was asked. In Absolute frames, MainTask started one frame later from wherever it was called, and the loop (in this example) actually starts the fourth loop (one additional frame) before checking to see that it should terminate. Therefore 3 x 5 = 15 frames plus the two additional frames is 17.

If

If <expression> <command>

The **IF** command, like **Trigger**, evaluates an expression, and, if the expression is true, executes the command listed. There are 3 types of possible commands: **Goto**, **Start**, and **End**. Each of these commands behaves exactly as if it were given without the **If** part of the statement.

```
00:00:00:01 Label loop
00:00:00:01 If diginput:191 = OFF Start MyTask2
00:00:00:01 If dmxoutput:1001 = 45 End
00:00:00:01 If var:2 <= var:3 Goto loop
```

Note that an **IF** looks at the *current* state of the resource in question. This behavior is known as being *level-triggered*. Although there is no "triggering" going on, this term is used to denote when a resources' current state is used directly.

The All-Important AUTOBEGIN task

To start, you only need one task. It is called AUTOBEGIN. You MUST have this task, If you do not, no tasks will ever be run, and your show will not happen. AUTOBEGIN is like the **autoexec.bat** file for those of you who remember running DOS on a PC. The AUTOBEGIN task executes when you first upload a show, or when the IMS is restarted. To "make" a task AUTOBEGIN, simply name it that. In the IMS, only quoted strings are case-sensitive, so you may name it AutoBegin, AUTOBEGIN, autobegin, AuToBeGiN, or any other combination you like. It does not have to be the first task in the show, although most programmers commonly make it the first task.

The AUTOBEGIN task is the best place to "setup" your show. As you learn more about Integrated Media Server programming in the pages to follow, you'll learn more and more commands that can be put in AUTOBEGIN.

Variable Resources (var)

Once you understand how tasks work and interact, you can start sectioning off your programming according to your goals. However, it's hard to program your resources without keeping track of what you're doing. The state of your show and various data elements can be stored in variables. Variables can actually store 3 different kinds of data: integer numbers, floating point numbers and strings. You can have as many variables as you need, up to 4,294,967,295 in fact. However, if you really used that many variables (over four billion), you would eat up 16 Gigabytes of memory for the variables alone, so in practice, you can't really use that many unless you purchase your IMS with a special memory card to increase your memory to more than 16 Gigabytes. It's certainly safe to say that for almost any application imaginable, 10,000 variables should be enough. But it's nice to know you can use more if you really do need more – they're waiting for you to use!

General Usage

To create a variable, simply start using it by setting it to some value. It's important to set your variable to some value first, before you use the variable for some purpose, because otherwise the script won't know what type of variable it's supposed to be, and your later comparison won't be of much use if the value you are comparing is not meaningful. Generally, you can set a variable to a fixed literal value, the value of another variable, or to the value of another resource. The Set command is the lifeblood of Braintrust. It will allow you to set almost anything to almost anything. Here we'll show you a few examples of setting a variable to a value:

Setting var:1 to a literal integer number

```
00:00:00:01 Set var:1 19 // literal integer number
```

Setting var:1 to a literal floating-point number

```
00:00:00:01 Set var:1 123.456
```

Setting var:1 to a literal string of characters

```
00:00:00:01 Set var:1 “hello world”
```

Setting var:1 to both the *type* and *value* of var:2. Var:2 could be an integer number, a floating-point number, or a string. Regardless of what type of variable var:1 used to be, it's now the same type as var:2.

```
00:00:00:01 Set var:1 var:2
```

Setting var:1 to both the *type* (integer) and the *value* of the resource (0 or 1) since a digital input can only be either 0 or 1 (on or off).

```
00:00:00:01 Set var:1 diginput:3
```

Setting var:1 to both the *type* (integer) and the *value* of the resource (0-255)

```
00:00:00:01 Set var:1 dmxoutput:92
```

Setting var:1 to both the *type* (string) and the *value* of the resource (any string of characters)

```
00:00:00:01 Set var:1 vidsource:1.statustext
```

Integer Variables

Integer variables store numbers between 0 and 4,294,967,295. Integer variables (and resources for that matter) can also be treated as *Boolean* variables. That is to say, you can use an integer variable to store a “true” or “false”. The variable still stores a 0 or a 1, ~~but effectively you can forget about that~~. Braintrust has several reserved values which you can use. This table lists the Reserved Values, which can be used with both variables *and* resources in general.

Reserved Value Name **Value** **Usage Examples**

OFF	0	Trigger diginput:3 = OFF Set var:2 OFF
FALSE	0	If var:69 = FALSE Set anaoutput:1 FALSE
ON	1	Set digoutput:92 ON Set dmxoutput:1 ON If anainput:19 = ON
TRUE	1	Set var:3 TRUE Trigger var:101 = TRUE If var:15 = TRUE Set digoutput:1023 TRUE

These are very common ways of using Reserved Values.

Integer values are great for storing intermediate values:

```
00:00:00:01 Set var:1 dmxinput:1 // first dmx input channel
00:00:00:01 Set dmxoutput:1 var:1 // first dmx output channel
```

You don't have to use intermediate variables, but sometimes it's useful:

```
00:00:00:01 Set dmxoutput:1 dmxinput:1
```

Usually you'll have one or more places in your show where a variable takes on a value. In this case, we're contriving the values, as long as they're unique, and we're just checking later to see what it was set to...

```
00:00:00:01 Set var:1 1 // run first show
```

somewhere else...

```
00:00:00:01 Set var:1 2 // run second show
```

then you'll have a place in your show where you check to see what the value of the variable is:

```
00:00:00:01 If var:1 == 1 Start FirstShow
00:00:00:01 If var:1 == 2 Start SecondShow
```

Sometimes you want to use a variable to store a calculation:

```
00:00:00:01 Set var:1 anainput:6 // get value of old position
```

Then later you check again...

```
00:00:00:01 Set var:2 anainput:6 // get value of new position
```

Then you want to do a calculation:

```
00:00:00:01 Subvar var:1 var:2 // subtract var:2 from var:1
```

Then test the value for whatever reason:

```
00:00:00:01 If var:1 < 2 Goto Error // pneumatics out of range
```

Floating-Point Variables

Floating-point variables are always used for calculations. They could, for example, be used for reading the value of a sensor that returns floating point numbers,

```
00:00:00:01 Call GetSensorData // puts sensor value in var:1
00:00:00:01 FormatVar var:2 "The sensor reads: %4.6f" var:1
00:00:00:01 Write display:1.row:1 var:2 // put on LCD
```

or for calculating prices and taxes for a cash register / receipt printer:

```
00:00:00:01    Set var:2 var:1 // keep var:1, use temp var:2
00:00:00:01    Set var:3 0.825 // Los Angeles Tax
00:00:00:01    MultiplyVar var:2 var:3 // get sales tax
00:00:00:01    AddVar var:1 var:2 // total for customer
00:00:00:01    FormatVar var:4 "Your total is $%4.2" var:1
00:00:00:01    Write serport:1 var:4 // print to receipt
```

String Variables

String variables are almost always used for reading by humans, whether in a log, printout, on the screen, or elsewhere. They can also be used for sending messages to serial or ethernet devices, and for those devices that require string commands be sent to them.

Here is an example of sending the error message from a video card to a Display LCD

```
00:00:00:01 Write display:1.row:2 vidsource:1.errortext
```

In the section explaining Floating-point variables we used two examples of string variables. In the first example, the string variable was sent to a display LCD. In the second example, the string variable was sent out a serial port connected to a thermal receipt printer.

Variable Commands

Some commands that are used for all resources apply to variables as well. Although these commands are explained elsewhere, a brief syntax description will be listed here:

Set <resourceto> <resourcefrom>
Write <serport> <resource1> .. <resourceN>
Display <display> [row] [col] <resource>
If (expressionterm equate expressionterm) <command>
Trigger (expressionterm equate expressionterm) <task> <params>

Some commands are specifically used with variables:

AddVar <varto> <varfromorliteral>
SubtractVar <varto> <varfromorliteral>
MultiplyVar <varto> <varfromorliteral>
DivideVar <varto> <varfromorliteral>
ModVar <varto> <varfromorliteral>

OrVar <varto> <varfromorliteral>
AndVar <varto> <farfromorliteral>
XorVar <varto> <varfromorliteral>
FormatVar <varto> <formatstring> <resource1>..<resourceN>

AddVar

AddVar adds the second value to the first variable, and leaves the result in the first variable.

SubtractVar

SubtractVar subtracts the second value from the first variable, and leaves the result in the first variable.

MultiplyVar

MultiplyVar multiplies the first variable by the second value, and leaves the result in the first variable.

DivideVar

DivideVar divides the first variable by the second value, and leaves the result in the first variable.

ModVar

ModVar divides the first variable by the second value, and leaves the REMAINDER in the first variable.

OrVar

OrVar bit-wise Ors the first variable by the second value, and leaves the result in the first variable.

AndVar

AndVar bit-wiseANDs the first variable with the second value, and leaves the result in the first variable.

XorVar

XorVar bit-wise XORs the first variable with the second value, and leaves the result in the first variable.

FormatVar

FormatVar is the most complicated command in Braintrust. Pages and pages could be written about it. Suffice it to say that FormatVar works identically to the very famous and ubiquitous to programmers C language **printf** command. If you're reading this right now and you're in a pickle to make **FormatVar** do something very complex or fancy, your best bet is to look up **printf** on the Internet or in any beginning C language book.

However, we won't leave you hanging completely...

FormatVar's syntax is:

FormatVar <varto> <formatstring> <resource1> .. <resourceN>

<varto> is the variable you wish the resulting formatted string to be stored in.
<formatstring> is the string template you want to fit your variables into.

In this simple example, <formatstring> is put into <varto> unmodified:

```
00:00:00:01 FormatVar var:1 “The month is January”
```

But FormatVar is best at making a nice string when you don't know all the data. This example uses printf's % symbol to replace part of the string with another variable. The letter after the % indicates what format of variable it is. In this case, it's a string. You then have to list, in order, what variable you want to replace it with:

```
00:00:00:01 Set var:2 “January”
00:00:00:01 Set var:3 “February”
00:00:00:01 FormatVar var:1 “The month is %s” var:2
```

Which would give you the result: *The month is January*

You can do this with multiple variables and replacements:

```
00:00:00:01 Set var:2 “Monday”
00:00:00:01 Set var:3 15
00:00:00:01 Set var:4 “March”
00:00:00:01 FormatVar var:1 “Date: %s %d 2004” var:4 var:3
```

Which would give you the result: *Date: March 15, 2004*

If you want to use the % symbol in your string text, use it twice to clue in FormatVar that that's what you want to do.

```
00:00:00:01 Set var2 50
00:00:00:01 FormatVar var:3 “The discount is %d%% off” var:2
```

Which would give you the result: *The discount is 50% off*

Here are various format types:

Type	**Description**
%d	integer number
%f	floating-point number
%h	hexidecimal number
%s	string

Sometimes it's useful to specify the precision of the variable. That is, the number of digits to display for an integer number or floating-point number

%2.2h
%4.2f

The number to the left of the period indicates how many total digits to display, and the number to the right of the period indicates how many digits should be displayed to the right of a decimal point in a floating point number. Here are some examples:

Type and Precision	Value	Result Formatted
%2.2h	31	1e
%2.2H	31	1E
%d	31	31
%1d	31	3
%2d	31	31
%3d	31	031
%f	31	31
%2.2f	31.123	31.12
%6.3f	31.123	031.123
%6.1f	31.123	00031.1
%s	“hello”	hello
%s	“goodbye”	goodbye

MPEG Standard-Def and Hi-Def Video Resources (video)

The Mpeg Decoder video resource is the resource we are best known for. The outputs are broadcast quality, the bits-per-second rate is unmatched, and the audio is clean. But what really sets us apart is our intense accuracy. The IMS platform in general provides a 60th of a second accuracy, which will perfectly match to each *field* (half of a video frame) when video resources are added. The audio coming out of the decoders will be *sample* accurate. This means that if the encoded audio is 48 kHz, or 48,000 samples per second, each decoder, even when synchronized with other decoders, will output exactly the same number of samples of audio, at exactly the same time/rate per field. We can loop video with audio with mpeg files as small as 5 frames with perfect playback accuracy. We can loop video files without audio down to 1 (one, yes you read that right) frame. We can jump from playing any file to playing any other file seamlessly, even if it's in the middle of the file. We can jump from a looping file to any other file seamlessly as well. We can pause on any frame of any file flawlessly. Of course other IMS subsystems can come into play. We can, for example, jump from looping a 3 frame mpeg file on two synchronized standard-definition decoders to playing another mpeg file seamlessly, while sending a serial message to a video switcher in the vertical interval causing the video switcher to switch outputs on the standard-def decoders before the first field of the new begins playing, while causing one high-definition decoders to start playback of a hi-def mpeg file 2 frames after the new standard definition file loop seamless switches, while setting an analogue output attached to the limb of an animated character to a value one *field* after that, ensuring perfect audio lip synch with the audio, while sending an ethernet message to the Parkwide Attraction Monitoring System that the various subsystems in that attraction are performing flawlessly for the 400th day in a row...

Whew!

There are attractions out there right now that utilize this sophistication *daily*.

Working with Disney attractions for so many years caused us to have a clear appreciation for the importance of accuracy. That accuracy is not easy, and it came at a huge development price. But the results are astounding, and worth it. Every subsystem benefits from being

forced to be as accurate as the video subsystem. Everything, and I do mean everything, can be made to be very precise.

MPEG Decoder resources, whether high-definition or standard-definition, are represented by the IMS video resource called "video." Files are loaded into the decoder, then started, then stopped. The basic method for getting MPEG files into the decoder is with the **Queue** command. Video playback is started via the **Start** command, and stopped with the **Stop** command.

Here is the most basic show video example imaginable: Playback of one file from start to finish:

```
00:00:00:01    Start vidsource:1 "myfile.mpg"
```

See how easy that was? One line of code. Need it explained?

Here is the second most basic show video example imaginable: Looping one file.

```
00:00:00:01    Start vidsource:1 "myfile.mpg" loop
```

Here's an example of playing two files back to back on three decoders and looping the second set of files. Remember that the video and audio will be in perfect synch:

```
00:00:00:01    Queue vidsource:1-3 "myfile.mpg"
00:00:00:01    Start vidsource:1-3
00:00:00:01    Queue vidsource:1-3 "myfile2.mpg" loop
```

You want something fancier? How about looping three different 10 second, 3 frame MPEG files on three different decoders, then, after they loop exactly 4 times, seamlessly transition to playing back another file synchronously, and on top of that, add in another decoder to match in perfect synch:

```
00:00:00:01    Queue vidsource:1 "myfile1.mpg" loop
00:00:00:01    Queue vidsource:2 "myfile2.mpg" loop
00:00:00:01    Queue vidsource:3 "myfile3.mpg" loop
00:00:00:01    Queue vidsource:4 "myfile4.mpg"
00:00:00:10    Start vidsource:1-3
00:00:35:00    Queue vidsource:1-3 "myfile4.mpg"
00:00:40:22    Start vidsource:4
```

You want something fancier *still*? What if you don't know which of two different files you want to play on decoder 2 until one frame before all four decoders are supposed to start playing together? Let's say you also want to fade to black on decoder 1 slowly over the entire 40 seconds 12 frames, so that it's exactly 50% black on the exact frame that all four decoders start playing the last file. Let's say then that you want decoder 4 to keep playing this 15 second file (loop it) while the other 3 decoders stop, and 21 hours 5 minutes, 4 seconds later (exactly to the frame) you want to stop decoder 4, which you know, even after 5060 times of looping, the video will be on the file's 119th frame of vidsource:

```
00:00:00:01    Queue vidsource:1 "myfile1.mpg" loop
00:00:00:01    Queue vidsource:2 "myfile2.mpg" loop
```

```
00:00:00:01    Open vidsource:2 “newfile1.mpg”
00:00:00:01    Open vidsource:2 “newfile2.mpg”
00:00:00:01    Queue vidsource:3 “myfile3.mpg” loop
00:00:00:01    Queue vidsource:4 “myfile4.mpg” loop
00:00:00:10    Start vidsource:1-3
00:00:00:10    Ramp vidoutput:1.videolevel 50% 00:00:40:12
00:00:35:00    Queue vidsource:1 “myfile4.mpg”
00:00:35:00    Queue vidsource:3 “myfile4.mpg”
00:00:40:21    Label It's the frame before- figure it out now!
00:00:40:21    If var:1 == 1 Goto UseFile1
00:00:40:21    If var:2 == 2 Goto UseFile2
00:00:40:21    End
00:00:40:21    Label UseFile1
00:00:40:21    Queue vidsource:2 “newfile1.mpg”
00:00:40:21    End
00:00:40:21    Label UseFile2
00:00:40:21    Queue vidsource:2 “newfile2.mpg”
00:00:40:21    End
00:00:40:22    Start vidsource:4
21:05:44:22    Stop vidsource:4
```

Ok, so you get the idea. Let's get back to basics. To play video files, you need to setup the output format of the video card if the default is not desired, you need to open the files you intend to play, Queue them when the time comes, and then Start and Stop them.

Video File Locations

All mpeg video files should be put in the directory \ims\shows\default\media

To access these files, you simply use the filename, like this:

Start vidsource:1 “myfile.mpg”

However, you may also put files in subdirectories of \ims\shows\default\media. For example, you may choose to breakup files between *preshow* and *mainshow*.

C:\ims\shows\default\media\preshow\queuearea1.mpg
C:\ims\shows\default\media\preshow\queuearea2.mpg
C:\ims\shows\default\media\preshow\preshow.mpg
C:\ims\shows\default\media\mainshow\left.mpg
C:\ims\shows\default\media\mainshow\centre.mpg
C:\ims\shows\default\media\mainshow\right.mpg

So you would play these files in Braintrust like this:

```
Start vidsource:1 “preshow\queuearea1.mpg”
Start vidsource:1 “preshow\queuearea2.mpg”
Start vidsource:1 “preshow\preshow.mpg”
Start vidsource:1 “mainshow\left.mpg”
Start vidsource:1 “mainshow\centre.mpg”
Start vidsource:1 “mainshow\right.mpg”
```

Note that you do NOT reference the file path prior to the subdirectory.

Now let's explain each of the commands in detail, then give some real-world simple examples of how video would be used.

Video Commands

Open

Open <video> <file> [options]

Opens files into the decoders, waiting to be queued. You can **Open** as many files as you'd like. Open loads the beginning part of the specified file into memory (see the **memory** option for how to load the entire file into memory). There is no importance in the order in which files are opened. They are simply loaded into a different slot. Opening a file reduces the time a queue takes later, and once a file is opened, it stays opened until specifically closed. Opening allows you to specify special parameters that will affect playback later. See the **memory** option and other options below.

```
00:00:00:01    Open vidsource:1 "file1.mpg"
00:00:00:01    Open vidsource:1 "file2.mpg"
00:00:00:01    Open vidsource:2-5 "file3.mpg"
```

If you want the opened file to be played back from memory, use the **memory** option. This will load the entire file into memory at the time of the **Open**, so it's somewhat slower to use this option. The normal default behavior is for **Open** to load only the first portion of the file into memory, and then load the rest of the file as needed into memory off the hard disc as playback ensues. The default way is fast to **Open**, but hard disc bandwidth will be diminished for each decoder that plays back from hard disc. The three factors that determine acceptable bandwidth for hard disc playback are

1) The speed of the hard disc/RAID drives configuration selected for the IMS
2) The number of decoders playing back from hard disc simultaneously
3) The bitrate of the MPEG files

Use the **memory** option when you can.

So to summarize, you should call Open anytime you know that you're going to be calling Queue when it has to be frame accurate, and you want to make sure Queue can be as fast as possible.

Queue

Queue <video> <file> [options]

Queue opens and loads MPEG video files into the decoders, ready for playback. **Queue** actually does two things. It calls **Open** to open the file if it is not already open from a previous **Open** or **Queue** command, and it tells the decoder to get ready to play it. The **Open** part of the job takes a while, but the second part of the command takes less than a frame. If the file is *already* opened via the **Open** command (see the **Open** command) then the entire length of the **Queue** operation is one frame.

If the decoder is stopped when you issue the Queue command, the file specified will be loaded into the decoder so that it will play when you issue a Start command. If there was a previous file queued (that wasn't played) when you issue the Queue command, the new Queued file will replace the old one.

```
00:00:00:01    Queue vidsource:1 “file1.mpg”
00:00:00:01    Queue vidsource:1 “file2.mpg” // replaces
file1.mpg
00:00:00:01    Start vidsource:1 // file2.mpg is played
```

If you want to loop a file add the **loop** option to the **Queue** commands.

```
00:00:00:01    Queue vidsource:1 “file5.mpg” loop
```

If you don't know what file you're going to play next until just before the end of the current file, use **Queue** *after* starting the decoder. Just make sure you issue a new Queue command during the currently playing file. As long as you continue to issue a new Queue command during the currently playing file, video will continue to play through each file without stopping, *ad infinitum*.

```
00:00:00:01    Queue vidsource:1 “file1.mpg”
00:00:00:01    Start vidsource:1
00:02:00:00    Queue vidsource:1 “file2.mpg”
```

If you **Queue** a file once a decoder has started playing, and then **Queue** another file, the second **Queue** replaces the already queued file. This allows you to change your mind on the fly.

```
00:00:00:01    Queue vidsource:1 “file1.mpg”
00:00:00:01    Start vidsource:1
00:02:00:00    Queue vidsource:1 “file2.mpg”
00:02:00:00    Queue vidsource:1 “file3.mpg” // replaces
file2.mpg
```

file3.mpg will play seamlessly after file1.mpg, and file2.mpg is not involved.

If you changed your mind, and don't want to play any file after file1.mpg, use the **Queue** command with the “clear” option

```
00:00:00:01    Queue vidsource:1 “file1.mpg”
00:00:00:01    Start vidsource:1
00:00:05:00    Queue vidsource:1 “file2.mpg”
00:02:00:00    Queue vidsource:1 clear
```

Note that Queue Clear does not stop the decoder from playing, only takes away the next file from the queue to play next.

You can also issue a Queue vidsource:1 unloop

```
00:00:00:01    Start vidsource:1 “file1.mpg” loop
00:10:00:00    Queue vidsource:1 unloop
```

This will cause the looping to stop, but not the decoder. In other words, the decoder will finish playing the existing file and stop.

If you want the queued file to be played back from memory, use the **memory** option. This will load the entire file into memory at the time of **Queue**, so it's somewhat slower to use this option. The normal default behavior is for **Queue** to load only the first portion of the file into memory, and then load the rest of the file as needed into memory off the hard disc as playback ensues. The default way is fast to **Queue**, but hard disc bandwidth will be diminished for each decoder that plays back from hard disc. The three factors that determine acceptable bandwidth for hard disc playback are

4) The speed of the hard disc/RAID drives configuration selected for the IMS
5) The number of decoders playing back from hard disc simultaneously
6) The bitrate of the MPEG files

Use the **memory** option when you can afford to.

Note that Queue actually is calling Open to do the open, regardless of whether it's in memory or not.

```
00:00:00:01    Queue vidsource:1 “file1.mpg” memory
```

If you need to jump *out* of a playing file to another file, even if it's a looping file or a playlist of looping files, use Start again. The default behavior for **Queue** is to queue the file to seamlessly play at the end of the currently playing file. Calling Start when you're already playing causes playback to jump to the specified file *immediately*.

Here is an example of jumping out of a looping file immediately to a new file. In this example, we will have two tasks. The first task loops a playlist of queued “queue line area” files, and the second task is started by a “stop show” button being pressed by a ride operator. In this example, the first loop of files is being played out of memory, and the second loop is being played off the hard disc.

```
- AutoBegin-
00:00:00:01    Trigger ( diginput:101 == ON ) StopShowTask
00:00:00:01    Start vidsource:1 “queueareafile3.mpg” memory
loop

- StopShowTask -
00:00:00:01    Start vidsource:1 “stopshowfile.mpg”
```

Start

Start <video> [options]

We’ve been using the **Start** command in previous examples while demonstrating the **Queue** command. **Start** begins playback on the specified decoders.

```
00:00:00:01    Start vidsource:1
```

Multiple decoders may be started at the same time. Playback will always be synchronous.

```
00:00:00:01    Start vidsource:1-3
```

Playback actually starts on the next frame. Any decoders started on the same frame will play on the next frame, and playback synchronously, even if they're from different **Start** commands:

```
00:00:00:01    Start vidsource:1-3
00:00:00:01    Start vidsource:5
00:00:00:01    Start vidsource:7-9
```

You can specify a file to play, and if you're stopped, the decoder will Queue the file before playing it.

```
00:00:00:01    Start vidsource:1-3 "myfile.mpg"
```

If you want to pass the memory option to Queue (and to Open) just add the word "memory" after the file to queue and then start.

```
00:00:00:01    Start vidsource:1-3 "myfile.mpg" memory
```

Stop

Stop <video>

Stop stops the specified decoders from playing back. Playback will stop on the next frame. The syntax and rules are the same as for the **Start** command.

```
00:00:00:01    Stop vidsource:1-9
```

After Stopping playback with the **Stop** command, the queued file, as it existed at the time of the **Stop** command, will be started again. This is most useful when a playlist is created with the **Queue** command before the **Start** command is issued. If the decoders are stopped, the playlist can be begin again simply by issuing the **Start** command again.

Pause

Pause <video>

Pause pauses (freezes) the specified decoders. Playback will pause on the next frame. The syntax and rules are the same for the Start command.

```
00:00:00:01    Pause vidsource:1-3
```

To unpause, issue a **Start** command, and the decoders will resume where they left off in the file they were on. The queue for the decoders stays intact as well.

Playlists

As you already know, you can playback more than one file seamlessly by queueing a file, starting the decoder, and continuing to queue files repeatedly. Alternatively, you can create a playlist of queued files *if you know ahead of time what files you want to queue in order*.

You can have as many playlists as desired, and use any created playlist as if you were playing one file. This applies to audio playback as well, which we will discuss later.

```
00:00:00:01    Queue playlist:1 VIDEOFILE “file1.mpg”
00:00:00:01    Queue playlist:1 VIDEOFILE “file2.mpg” memory
00:00:00:01    Queue playlist:1 VIDEOFILE “file3.mpg” memory
```

(Note that in this arbitrary example, the first file is not queued to memory. It does not matter whether the files you queue in the playlist are opened to memory or opened to hard disc).

Later you use this playlist when queuing to a decoder:

```
00:00:00:01    Queue video:3 playlist:1
```

All the files in the playlist will play in order, as if you had queued them manually.

You can loop playlists as if it were just one long file as well.

```
00:00:00:01    Queue video:9 playlist:6 loop
```

You can change an entry (file to play) in the playlist, after it has been created by issuing the following command (example):

```
00:00:00:01    Set playlist:4.entry:16 “file96.mpg”
```

You can delete an entire playlist by issuing:

```
00:00:00:01    Delete playlist:7
```

or you can delete just one entry in a playlist:

```
00:00:00:01    Delete playlist:2.entry:8
```

Video Objects

Video resources have several object properties. Some of the objects are settings that can be changed programatically, others are set by the IMS system for user interrogation, and still others are both.

Mixing/Routing

Currently, there is no on-board mixing/routing for video. This means there is a fixed, one-to-one correspondence between the vidsource and the vidoutput. For example, the corresponding output of vidsource:13 is vidoutput:13. Although you cannot substitute vidsources for vidoutputs, the index referring to either will always be the same, as with the example for 13, above.

Output Level and Fading

You can set the level of the output of either video or audio immediately using the Set command, or over time using the **Ramp** command (for Fading). For video, the level represents how much of the picture is showing, and how much is black. For audio, the level represents the gain or volume of the audio.

Let's set the audio level to 75% for 10 seconds, then set it back to 100%.

```
00:00:00:01    Set vidoutput:1.audiolevel 75%
00:00:10:00    Set vidoutput:1.audiolevel 100%
```

Let's set the video level to 60% picture, 40% black and leave it there.

```
00:00:00:01    set vidoutput:1.videolevel 60%
```

You can also **Fade** the video or audio over time using the **Ramp** command. This example fades to black over a 3 second period.

```
00:00:00:01    Ramp vidoutput:1.videolevel 0% 00:03:00:00
```
This example fades audio to nothing over a 1 second period.

```
00:00:00:01    Ramp vidoutput:1.audiolevel 0% 00:02:00:00
```

This example fades video down then back up again between files:

```
00:00:00:01    Queue vidoutput:1 “firstfile.mpg”
00:00:02:00    Start vidoutput:1
00:01:02:00    Ramp vidoutput:1.videolevel 0% 00:02:00:00
00:01:04:00    Queue vidoutput:1 “secondfile.mpg”
00:01:04:02    Ramp vidoutput:1.videolevel 100% 00:02:00:00
```

Captioning

Set this object to true if Closed Captioning is desired on the line 21 of the video. Make sure your mpeg file has Closed Captioning Data for the file in question.

```
00:00:00:01    Set vidoutput:1.captioning true
00:00:00:01    Set vidoutput:1.captioning on
00:00:00:01    Set vidoutput:1.captioning 1
00:00:00:01    Set vidoutput:1.captioning off
```

Of course, as always, you can do silly things if you want to. You never know what unique show situations will arise which may require these rare options:

```
00:00:00:01    Set vidoutput:1.captioning diginput:3
```

Format

To initialize the video card output format, use the **Format** object property. The default output format for standard-definition decoders is **NTSC**. The default output format for high-definition decoders is **1080I_2997_RGBHV**.

```
00:00:00:01    Set vidoutput:1.format PAL
00:00:00:01    Set vidoutput:1.format 1080I_2997_YPBPR
```

Here is a list of all of the allowed video formats for the regular standard definition decoders:

NTSC
PAL

Here is a list of all of the allowed video formats for the standard definition decoders with the RGB option.

NTSC
PAL
480I_RGB
480I_RGBHV
480I_YPBPR
576I_RGB
576I_RGBHV
576I_YPBPR

Here is a list of all of the allowed video formats for high definition decoders:

NTSC
PAL
1080I_2997_RGB
1080I_2997_RGBHV
1080I_2997_YPBPR
1080I_3000_RGB
1080I_3000_RGBHV
1080I_3000_YPBPR
720P_5994_RGB
720P_5994_RGBHV
720P_5994_YPBPR
720P_5994_YPBPR
720P_6000_RGB
720P_6000_RGBHV
720P_6000_YPBPR

Status and StatusText

These are read-only objects that inform the user of the video resource playing status. As read-only files, they cannot be set by the user, but instead are set by the video subsystem as the status changes.

The **Status** object is an integer number which can easily be triggered or compared to our pre-defined values.

```
00:00:00:01    Set var:1 vidsource:1.status
00:00:00:01    Trigger vidsource:1.status = STATUSSTARTED
Playing
00:00:00:01    If vidsource:1.status = STATUSSTARTED Goto
Playing
```

Valid values for the **Status** Object Property are:

STATUSSTARTED
STATUSSTOPPED
STATUSPAUSED

The **StatusText** objects is a string value used mainly for logging or for displaying to a human due to some condition. This object's textual value will always be set at the same time as the **Status** object. They have the same content, but Status is used to make decisions with and **StatusText** is used to display to humans (or Klingons that speak English).

```
00:00:00:01    Set var:1 vidsource:1.statustext
00:00:00:01    Write serport:1 vidsource:1.statustext
00:00:00:01    Write display:1.row:2 vidsource:1.statustext
```

Error and Error Text

These are objects which inform the user what went wrong with the corresponding video resource. They are not changed by the system until there is a new error. If you are finished using the information when an error occurs, you may reset to some non-error condition, or you may leave the error message there until the next error occurs.

The **Error** property is an integer number which can easily be triggered or tested in some way, the same way as **Status**.

```
00:00:00:01    Set var:1 vidsource:1.error
00:00:00:01    Trigger vidsource:1.error != NOERROR
ErrorVideo1
00:00:00:01    If vidsource:1.error = ERRORFILEMISSING Goto
NoFile
```

Valid values for the **Error** object are:

ERRORNONE
ERRORMISSINGFILE
ERRORNOTQUEUED
ERRORQUEUING
ERROROPENING
ERRORCLOSING
ERRORINITIALIZING
ERRORSTOPPING
ERRORLOOPING
ERRORMUTING
ERRORCUEING
ERRORSETTINGOUTPUTFORMAT
ERRORSETTINGGENLOCK
ERRORSETTINGCAPTIONING

The **ErrorText** object is a string value used mainly for logging or display, the same way that **StatusText** is used.

```
00:00:00:01    Set var:1 vidsource:1.errortext
00:00:00:01    Write serport:1 vidsource:1.errortext
00:00:00:01    Write display:1.row:2 vidsource:1.errortext
```

Common Examples of Video Playback

Suppose you want to loop a file all day long on a standard definition decoder. The NTSC output format is fine, and it is the default, so you don't need to set anything. As a small file, it will play many times so you decide you want it to play out of memory. You use task Daymode to start this looping file up, and task Nightmode to stop it.

- Daymode -
```
00:00:00:01    Start vidsource:1 "loopingfile.mpg" memory loop
```

- Nightmode -
```
00:00:00:01    Stop vidsource:1
```

You want to playback a file and reload it for a ride track trigger. This time, you need to playback files in PAL. In this first example, you know the length of the TinkerbellFlitters.mpg file, so you just queue it after it plays:

- AutoBegin -
```
00:00:00:01    Set vidoutput:1.format PAL
00:00:00:01    Queue vidsource:1 "TinkerbellFlitters.mpg"
00:00:00:01    Trigger diginput:3 = OFF GoTink
```

- GoTink -
```
00:00:00:01    Start vidsource:1
00:00:17:15    Queue vidsource:1 "TinkerbellFlitters.mpg"
```

In this second example, the length of TinkerbellFlitters.mpg is not known, or it may change after the media is updated, or you just want the IMS to handle reloading it for you. So, you use a trigger of the Status object compared to STATUSSTOPPED (tinkerbellflitters.mpg ends) to run task LoadTink, which queues the file to run again)

- AutoBegin -
```
00:00:00:01    Set vidoutput:1.format PAL
00:00:00:01    Queue vidsource:1 "TinkerbellFlitters.mpg"
00:00:00:01    Trigger diginput:3 = OFF GoTink
00:00:00:01    Trigger vidsource:1.status = STATUSSTOPPED
LoadTink
```

- GoTink -
```
00:00:00:01    Start vidsource:1
```

- LoadTink -
```
00:00:00:01    Queue vidsource:1 "TinkerbellFlitters.mpg"
```

In this example, you want to loop a hi-definition decoder file on a hi-definition output card set to 1080I_2997_RGBHV for a preshow. When it's time to go into the theatre, you want it to seamlessly jump immediately to another file while fading down during the transition.

- AutoBegin -
```
00:00:00:01    Set vidoutput:1.format 1080I_2997_RGBHV
00:00:00:01    Queue vidsource:1 "PreshowLoop.mpg" memory loop
00:00:00:01    Trigger diginput:19 = ON StartShow
```

- Daymode -

```
00:00:00:01    On var:1 // now in daymode
00:00:00:01    Start vidsource:1 // begin looping file
```

- StartShow -

```
00:00:00:01    If var:1 = OFF End // Must be in daymode
00:00:00:01    Label Fade to 0% in 2 seconds
00:00:00:01    Ramp vidoutput:1.videolevel 0% 00:00:02:00
00:00:00:01    Label Jump to new video
00:00:02:01    Start vidsource:1 "NowEnteringTheatre.mpg"
00:00:02:03    Set vidoutput:1.videolevel 100% // back to
picture
```

Audio Resources (audsource, audoutput)

The Integrated Media Server allows a large number of audio channels to be played back sample-accurately, started and stopped on a frame basis, in the same manner as video. Each audio card can have its own sample rate and resolution (unfortunately you cannot have a different sample rate and resolution for each channel, but fortunately that's rarely a concern). You can mix up to four streams of audio out each audio output at different volume levels.

The IMS can play files up to 192kHz, 32 bit files.

Audio File Locations

Audio files are put in the same place as video or other files on the IMS.

C:\ims\shows\default\media

See the video section for details on where files are placed.

Audio Commands

Audio Resources work *exactly* the same as video sources, except for the object properties. It's so similar, there's no need to repeat it here. Opens, Queues, Starts, Stops, and Playlists are all the same except for the .wav extension on the audio files instead of .mpg. See the Video Resource section for details. We'll give an example here just so you can get a feel for it, and then move on to the audio objects, where the differences lie.

This is an example of setting up an audio channel (output), and looping a file all day.

- AutoBegin -

```
00:00:00:01    Set audoutput:1.resolution 32 // 32 bit
00:00:00:01    Set audoutput:1.samplerate 96000 // 96 kHz
00:00:00:01    Start audsource:1 "myfile.wav" memory loop
```

Audio Objects

Audio resources have several object properties. Some of the objects are settings that can be changed programatically, others are set by the IMS system for user interrogation, and still others are both.

Objects that apply to how the audio is played are objects of **audoutput**. **Audsources** are channels to playback files from. **Audoutputs** are actual audio outputs that **audsources** play into. None, one, two, three, or four audsources may play into one audoutput. By default, each audoutput has exactly one matching audsource that plays into it.

audoutput:1 receives audio from audsource:1
audoutput:2 receives audio from audsource:2
...
audoutput:19 receives audio from audsource:19

etc.

as you will see below, this is equivalent to these mixing commands:

```
00:00:00:01    set audoutput:1.auxbuss:1 audsource:1
00:00:00:01    set audoutput:2.auxbuss:1 audsource:2
00:00:00:01    set audoutput:19.auxbuss:1 audsource:19
```

Mixing/Routing

If you want to change the default mixer setup, you can. There are four auxbuss crosspoints that allow you to flow up to four audsources into each audoutput. Note that you can only mix audsources from the same physical card as is the audoutput. You cannot mix between cards.

To set the auxbuss of an audoutput to a particular audsource, you pick the output you want to set (1..n), the auxbuss of that output you want to set (1-4), and the audsource you want to set (1..n).

```
00:00:00:01    Set audoutput:1.auxbuss:1 audsource:2
```

Here is an example of mixing two playback channels into the first output, and two other playback channels into the second output.

```
00:00:00:01    Set audoutput:1.auxbuss:1 audsource:1
00:00:00:01    Set audoutput:1.auxbuss:2 audsource:2
00:00:00:01    Set audoutput:2.auxbuss:1 audsource:3
00:00:00:01    Set audoutput:2.auxbuss:2 audsource:4
```

To disconnect an audsource from an output, you can either use:

```
00:00:00:01    Clear audoutput:1.auxbuss:1
```

or you can specifically set the auxbuss off:

```
00:00:00:01    set audoutput:1.auxbuss:1 OFF
```

Fading

You can set the level of the audio output immediately using the Set commands. The level represents the gain or volume of the audio.

Let's set the audio level to 75% for 10 seconds, then set it back to 100%.

```
00:00:00:01    Set audoutput:1.level 75%
00:00:10:00    Set audoutput:1.audiolevel 100%
```

You can also set the level of individual auxbusses (1-4).

```
00:00:00:01    Set audoutput:1.auxbusslevel:2 75%
```

Sample Rate and Resolution

You need to set the sample rate and resolution if your files to playback are not 16 bit 48kHz files. Note that each physical card must be set to the same sample rate and resolution, and setting any output channel on that card will change the sample rate and resolution for ALL the output channels on the card.

Valid values for the **SampleRate** object are:

44100
48000
96000
192000

Valid values for the **Resolution** Object Property are:

16
20 // not supported yet. Upres files to 32 bits for the exact same performance
24 // not supported yet. Upres files to 32 bits for the exact same performance
32

```
00:00:00:01    Set audoutput:1.samplerate    96000
00:00:00:01    Set audoutput:1.resolution    32
```

Status and StatusText

These are read-only objects that inform the user of the video resource playing status. As read-only files, they cannot be set by the user, but instead are set by the video subsystem as the status changes.

The **Status** object is an integer number which can easily be triggered or compared to our pre-defined values.

```
00:00:00:01    Set var:1 audsource:1.status
00:00:00:01    Trigger audsource:1.status = STATUSSTARTED
Playing
00:00:00:01    If audsource:1.status = STATUSSTARTED Goto
Playing
```

Valid values for the **Status** object are:

STATUSSTARTED
STATUSSTOPPED
STATUSPAUSED

The **StatusText** objects is a string value used mainly for logging or for displaying to a human due to some condition. This object's textual value will always be set at the same time as the **Status** object. They have the same content, but Status is used to make decisions with and **StatusText** is used to display to humans.

```
00:00:00:01    Set var:1 vidsource:1.statustext
00:00:00:01    Write serport:1 audsource:1.statustext
00:00:00:01    Write display:1.row:2 audsource:1.statustext
```

Error and Error Text

These are object properties that inform the user what last went wrong with the corresponding audio resource. They are not changed by the system until there is a new error. If you are finished using the information when an error occurs, you may reset to some non-error condition, or you may leave the error message there until the next error occurs.

The **Error** property is a number that can easily be triggered or tested in some way.

```
00:00:00:01    Set var:1 audsource:1.error
00:00:00:01    Trigger audsource:1.error != NOERROR ErrorAud1
00:00:00:01    If audsource:1.error = ERRORFILEMISSING Goto
NoFile
```

Valid values for the **Error** Object Property are:

ERRORNONE
ERRORMISSINGFILE
ERRORNOTQUEUED
ERRORQUEUING
ERROROPENING
ERRORCLOSING
ERRORINITIALIZING
ERRORSTOPPING
ERRORLOOPING
ERRORMUTING
ERRORSETTINGSAMPLERATE
ERRORSETTINGRESOLUTION

The **ErrorText** property is a string variable used mainly for logging or for displaying to a human.

```
00:00:00:01    Set var:1 audsource:1.errortext
00:00:00:01    Write serport:1 audsource:1.errortext
00:00:00:01    Write display:1.row:2 audsource:1.errortext
```

Digital Input Resources (diginput)

We support several configurations of digital input cards. This allows a great deal of flexibility, suited to the needs of the user. We support dry contact, wet contact, isolated, non-isolated, high-power, TLL, and just about every other form of digital I/O circuitry.

Regardless of the electrical configuration, when programming it's just a digital input.

Diginput Commands

If and Trigger

If and **Trigger** commands were explained at the beginning of the manual, but here is a closer analysis with diginput resources. The valid state of diginput resources is either ON or OFF, logically either 1 or 0. The **If** command is *level-triggered*, and looks at the state of the diginput resource specified at the moment of the **If**. The **Trigger** command, on the other hand, is *edge-triggered*, and only starts the task if there is a transition between the opposite state and the desired state. If a diginput resource is OFF, and you want to trigger a task when it goes ON, the task will be started when the diginput resource goes from OFF to ON, and then does not trigger again until it goes from OFF to ON again.

Here are some examples of using **If** and **Trigger** with diginput resources:

```
00:00:00:01    If diginput:1 = ON Goto SensorEnabled
00:00:00:01    If diginput:3 = OFF End
00:00:00:01    If diginput:2 = digoutput:9 Start SensorMatch
00:00:00:01    Trigger diginput:1 = ON
00:00:00:01    Trigger diginput:592 = FALSE
00:00:00:01    Trigger diginput:96 = 1
00:00:00:01    Trigger var:3 = diginput:4
```

Digoutput Commands

If and Trigger

If and **Trigger** commands were explained at the beginning of the manual, but here is a closer analysis with digoutput resources. The valid state of digoutput resources is either ON or OFF, logically either 1 or 0. The **If** command is *level-triggered*, and looks at the state of the digoutput resource specified at the moment of the **If**. The **Trigger** command, on the other hand, is *edge-triggered*, and only starts the task if there is a transition between the opposite state and the desired state. If a digoutput resource is OFF, and you want to trigger a task when it goes ON, the task will be started when the digoutput resource goes from OFF to ON, and then does not trigger again until it goes from OFF to ON again.

You may ask yourself, why would I want to test a digital output? Isn't it my script that turned it on or off to begin with? The answer is yes of course, but you may feel it is appropriate to create self-contained tasks that test output conditions, or otherwise to make sure in one part of your show that the other part is working correctly, without a bunch of additional logic. It's just a great way to confirm things are working properly.

Here are some examples of using **If** and **Trigger** with digoutput resources:

```
00:00:00:01    If digoutput:1 = ON Goto PumpEnabled
00:00:00:01    If digoutput:3 = OFF End
00:00:00:01    If digoutput:2 = diginput:9 Start SensorMatch
00:00:00:01    Trigger digoutput:1 = ON
00:00:00:01    Trigger digoutput:592 = FALSE
```

```
00:00:00:01    Trigger digoutput:96 = 1
00:00:00:01    Trigger var:3 = digoutput:4
```

On

On <digoutput>

The **On** command turns a digoutput to the ON, TRUE, or 1 position, logically and electrically.

```
00:00:00:01    On digoutput:1
00:00:00:01    On digoutput:2-51,58,101-234
```

Off

Off <digoutput>

The **Off** command turns a digoutput to the OFF, FALSE, or 0 position, logically and electrically.

```
00:00:00:01    Off digoutput:1
00:00:00:01    Off digoutput:2,7-49
```

Set

Set <digoutput> <value>

The **Set** command turns a digoutput to a particular value. If the value is non-zero, it is equivalent to the **On** command. If the value is zero, it is equivalent to the **Off** command.

```
00:00:00:01    Set digoutput:1 ON
00:00:00:01    Set digoutput:16-21 OFF
00:00:00:01    Set digoutput:31 var:6
00:00:00:01    Set digoutput:92-95,97 dmxinput:13
```

Queue

Queue <digoutput> <animationfile.wav>

The **Queue** command loads the specified animation wave file into memory and assigns it to the specified digital output. Unlike audio and video files that you may choose to place entirely in memory or play off the hard disk, animation wav files are so small in comparison that they are always loaded into memory in their entirety.

Note: Animation files can be recorded by the IMS, or can be created by the user. The only requirements are that the data is recorded in mono (one channel), the samples per second rate is 60, and for digital animation files, values less than 50% are OFF and values greater than 50% are ON.

```
00:00:00:01    Queue digoutput:1 “diganimation.wav”
00:00:00:01    Queue digouput:9-22 “duplicatedata.wav”
00:00:00:01    Queue digoutput:1 “fountainshowchannel1.wav”
00:00:00:01    Queue digoutput:32 “fountainshowchannel32.wav”
```

Start

Start <digoutput>

The **Start** command begins playing back animation for the specified digital output resource from the current location in the animation file. If the animation file has just been queued, playback will start from the beginning of the file. If the animation file has previously been started and was paused, playback will resume from the pause point.

```
00:00:00:01    Start digoutput:1
00:00:00:01    Start digoutput:19-30
00:00:00:01    Start digoutput:5,12-18,21,141-156
```

Pause

Pause <digoutput>

The **Pause** command pauses animation playback for the specified digital output if it is currently playing back. A subsequent **Start** command will resume playback from where it paused.

```
00:00:00:01    Pause digoutput:1
00:00:00:01    Pause digoutput:38-251
00:00:00:01    Pause digoutput:9,13
```

Stop

Stop <digoutput>

The **Stop** command stops animation playback for the specified digital output if it is currently playing back. **Stop** causes the animation file to reset to the beginning.

```
00:00:00:01    Stop digoutput:1
00:00:00:01    Stop digoutput:56-59,61
00:00:00:01    Stop digoutput:3
```

Serial Port Resources (serport)

Serial ports allow serial communication between the IMS and another device. There are various types of serial ports which have different speed limitations, electrical characteristics, and sometimes other settings. The IMS supports a great deal of these options, and make their use as standardized as possible. Serial ports have incoming and outgoing data streams or *buffers*. This is where the data resides that is useful to the programmer. Almost all of the serial commands refer to an object property of the serial port.

NOTE: It is important to understand a few underlying concepts of the IMS in regards to serial ports. The first is that the general way to utilize a serial port is to configure its physical characteristics, and then to put data in its outgoing buffer or to look for data coming in its ingoing buffer. You may be interested only in telling a device what to do, with no feedback, or you may be waiting to be told to do something from another device without giving it feedback. In these cases, you can work with the outgoing or incoming buffer without worrying about the other. In all other cases, you'll be working with both. However, usually one section of code deals with outgoing messages, and one section of code deals with

incoming messages. Rarely, if ever, is it necessary to manipulate both buffers in the same section of code. We'll look at working with each buffer separately, as you undoubtedly will. Regardless of which buffer you're dealing with, the serial port needs to first be configured properly. This is normally done once in AutoBegin, or in a serial initialization routine called from AutoBegin.

The following example configures the serial port, and sends out several serial messages without expecting a response from the device (even if it sends one):

```
- AutoBegin -
00:00:00:01    Start ConfigSerialPorts
00:00:00:01    Start MainShow

- ConfigSerialPorts -
00:00:00:01    Set serport:1.baudrate 9600
00:00:00:01    Set serport:1.databits 8
00:00:00:01    Set serport:1 stopbits 1
00:00:00:01    Set serport:1 parity NONE

- MainShow -
00:00:00:01    Search serport:1 1000
00:00:00:01    Play serport:1

- Search -
00:00:00:01    Write param:1 “SE” param:2 0x0D

- Play -
00:00:00:01    Write param:1 “PL” 0x0D
```

If we want to make sure we got a response from the device, it takes a lot more additional code. Fortunately for serial ports, it is common to *import* sections of already created tasks, which can be considered a “protocol” file.

The following example expounds upon the prior one: the Search and Play tasks have been replaced, and additional tasks have been added.

```
- Search -
00:00:00:01    Write param:1 “SE” param:2 0x0D
00:00:00:01    Start GetResponseFromSerPort1

- Play -
00:00:00:01    Write param:1 “PL” 0x0D
00:00:00:01    Start GetResponseFromSerPort1

- GetResponseFromSerPort1 -
00:00:00:01    Delete serport:1.inbuffer
00:00:00:01    Set var:1 FALSE
00:00:00:01    Trigger serport:1.inbufferchanged SP1RXD
00:00:00:01    Label Wait 15 frames for a response
00:00:00:15    Label Should have gotten a response by now
00:00:00:15    RemoveTriggers SP1RXD
```

```
00:00:00:15    If ( var:1 = TRUE ) End
00:00:00:15    Label Error
00:00:00:15    Write display:1.row:1 “SP1: Invalid Response”

- S1RXD -
00:00:00:01    Label Trigger: serport:1.inbuffer has changed
00:00:00:01    Label make sure the inbuffer has min 2 bytes
00:00:00:01    If ( serport:1.inbuffercount < 2 ) End
00:00:00:01    Label make sure the first byte is correct
00:00:00:01    If ( serport:1.inbuffer:1 != “R” ) End
00:00:00:01    Label make sure the second byte is correct
00:00:00:01    If ( serport:1.inbuffer:2 != 0x0D End
00:00:00:01    Label if we got here we have a full response
00:00:00:01    Set var:1 TRUE
```


Write

Write <serport> <data1>..<dataN>

Transmitting data out a serial port is easier than receiving data. You don't have to worry about the outgoing buffer if you're only sending bytes out the port. **Write** will accomplish this:

```
00:00:00:01    Write serport:1 “send string out the port”
00:00:00:01    Write serport:9 15
00:00:00:01    Write serport:12-14 0x0D
00:00:00:01    Write serport:4 SERIALBREAK
00:00:00:01    Write serport:1 vidsource:2.statustext
00:00:00:01    Write serport:7 var:9
```

Multiple data sets can be sent out in one command. Here is an example of sending out a Pioneer laserdisc Search command to search to frame 100 (SE00100 CR LF)

```
00:00:00:01    Set var:1 100
00:00:00:01    FormatVar var:1 “%5.5d” var:2
00:00:00:01    Write serport:7 “SE” var:1 0x0D
```

One message can be divided over multiple Write commands as long as they are sent out at the same time:

```
00:00:01:01    Write serport:7 “SE”
00:00:00:01    Write serport:7 var:1
00:00:00:01    Write serport:7 0x0D
```

One special serial data type is SERIALBREAK. This special data type causes a Line Break to be put out on the serial port.

```
00:00:00:01    Write serport:5 SERIALBREAK
```

Delete

Delete <serport.inbuffer or serport.inbuffer.byte or serport:outbuffer or serport:outbuffer.byte>

The **Delete** command removes all or part of the specified buffer. For sending, Delete is usually used just to clear out the outbuffer, but for receiving, there are different methods which would cause you to delete the entire inbuffer, or one or more bytes out of it:

```
00:00:00:01    Delete serport:3.outbuffer
00:00:00:01    Delete serport:2.inbuffer
00:00:00:01    Delete serport17-19.inbuffer:2
00:00:00:01    Delete serport:5.inbuffer:2-5
00:00:00:01    Delete serport:5-9.inbuffer:17-41
```

Serport Objects

Serports have many objects. You need to configure the first set of properties, every time, unless you are happy with default 9600 baudrate, 8 databits, 1 stopbits, NONE parity settings.

Baudrate

The baudrate is the *speed* at which you need the data to send/receive at. You can set the baud rate to any value with our serial ports. If you're talking between two IMS units (for some reason) and want to make it difficult for someone to tap the line and listen to the messages? Set the baudrate on both to 10000 baud or 10001 baud, or some other unique value. The standard values are:

300
1200
2400
4800
9600
19200
38400
115200

```
00:00:00:01    Set serport:1.baudrate 9600
00:00:00:01    Set serport:3.baudrate 38400
00:00:00:01    Set serport:5-7.baudrate 115200
```

DataBits

The **databits** object is how many *bits* of data you want to **send** for each byte. You can either set it to **8** bits (0-255) or **7** bits (0-127).

```
00:00:00:01    Set serport:1.databits 8
00:00:00:01    Set serport:9.databits 7
```

StopBits

The **stopbits** object is how many bits you want to use to tell the receiving serial port that the data bits are done transmitting. You can set it to **1** or **2**. 1 is normal, but if you want to space out bytes a bit more, you can set it to 2 (if you know what you're doing).

```
00:00:00:01    Set serport:1.stopbits 1
00:00:00:01    Set serport:19 stobits 2
```

Parity

The **parity** object determines whether you want a parity bit at all, and if you do, whether it ~~is~~ tests for Odd or Even parity in the data bits, or if it's always 1, or always 0.

NONE: No parity bit at all
ODD: Tests data bits for odd parity
EVEN: Tests data bits for even parity
MARK: Force parity bit to a 0
SPACE: Force parity bit to a 1

```
00:00:00:01    Set serport:1.parity NONE
00:00:00:01    Set serport:7.parity EVEN
00:00:00:01    Set serport3-4.parity SPACE
```

Here are some common setups for serial port configurations. The speed (baudrate) can vary, but these settings normally go together:

```
00:00:00:01    Set serport:1.databits 8
00:00:00:01    Set serport:1.stopbits 1
00:00:00:01    Set serport:1.parity NONE

00:00:00:01    Set serport:1.databits 7
00:00:00:01    Set serport:1.stopbits 1
00:00:00:01    Set serport:1.parity EVEN

00:00:00:01    Set serport:1.databits 7
00:00:00:01    Set serport:1.stopbits 1
00:00:00:01    Set serport:1.parity ODD
```

Outbuffer

You can put bytes directly into the outgoing buffer. You do this with a **Set** command:

```
00:00:00:01    Set serport:1.outbuffer var:1
00:00:00:01
```

The **Write** command automatically assumes you want to put the data in the serial port buffer, but you can specify it manually. The following three tasks are equivalent:

- SendTest1 -
```
00:00:00:01    Delete serport:1.outbuffer
00:00:00:01    Write serport:1 “test”
```

- SendTest2 -
```
00:00:00:01    Delete serport:1.outbuffer
00:00:00:01    Set serport:1.outbuffer “test”
```

- SendTest3 -

```
00:00:00:01    Delete serport:1.outbuffer
00:00:00:01    Set serport:1.outbuffer:1 “t”
00:00:00:01    Set serport:1.outbuffer:2 “e”
00:00:00:01    Set serport:1.outbuffer:3 “s”
00:00:00:01    Set serport:1.outbuffer:4 “t”
```

OutBufferCount

The **outbuffercount** property tells you how many bytes are in the **outbuffer**. It's not as useful as **inbuffercount**, because usually you know how many bytes you sent out the port in your code, but it's there for completeness sake.

```
00:00:00:01    Set var:1 serport:1.outbuffercount
00:00:00:01    If ( serport:1.outbuffercount < 3 ) Goto Bad
00:00:00:01    Trigger serport:1.outbuffercount > 2048 BigMsg
```

OutBufferChanged

The **outbufferchanged** object tells you that the **outbuffer** has added or removed bytes. While this may also be determined by noticing a change through the outbuffercount value, the outbufferchangted property automatically informs the user when the outbuffer has changed for any reason.

```
00:00:00:01    Trigger serport:3.outbufferchanged MyTask
00:00:00:01    If serport:1.outbufferchanged = TRUE Goto Ok
```

InBuffer

The **inbuffer** object gives you access to bytes in the incoming receive buffer. In fact, this is the normal way to process serial commands. You can take bytes out with **Set**, Remove them with **Delete** , or test them with **If**.

```
00:00:00:01    Set var:1 serport:1.inbuffer
00:00:00:01    Set var:1 serport:1.inbuffer:3
00:00:00:01    Set var:1 serport.1.inbuffer:9-12

00:00:00:01    Delete serport:1.inbuffer
00:00:00:01    Delete serport:1.inbuffer:5
00:00:00:01    Delete serport:1.inbuffer:29-61

00:00:00:01    If ( serport:1.inbuffer = “test”) goto Ok
00:00:00:01    If ( serport:1.inbuffer:1-4 = “test”) goto Ok
00:00:00:01    If ( serport:1.inbuffer:5 = 0x0D ) goto Ok
```

InBufferCount

The **inbuffercount** object tells you how many bytes are in the inbuffer. If 3 bytes come in the serial port, inbuffercount will be 3. If you then delete one of the bytes with Delete, then inbuffercount will be 2. It's a great way of testing if enough data has come in to complete your message.

```
00:00:00:01    Set var:1 serport:1.inbuffercount
```

```
00:00:00:01    If ( serport:1.inbuffercount > 4 ) Goto Ok
00:00:00:01    Trigger serport:1.inbuffercount > 2 Got2Bytes
```

InBufferChanged

The in**bufferchanged** property tells you that the **inbuffer** has added or removed bytes. It's similar to in**buffercount** (because that changes whenever **inbufferchanged** changes), but it just lets you know anytime the **inbuffer** has changed for any reason.

```
00:00:00:01    Trigger serport:2.inbufferchanged SP2RXD
00:00:00:01    If serport:1.inbufferchanged = TRUE Goto Ok
```

Here is another example of using the properties of serport to look for an incoming message from another device. In this case, the example uses a 5 byte message, where the first bytes is the Start of Message byte, and is always 0xFF, and the last four bytes are data bytes to determine what to do:

```
- Setup -
00:00:00:01    Label Setup Serial Protocol
00:00:00:01    Set serport:1.baudrate 19200
00:00:00:01    Set serport:1.databits 8
00:00:00:01    Set serport:1.stopbits 1
00:00:00:01    Set serport:1.parity NONE
00:00:00:01    Label Setup Serial Interrupts
00:00:00:01    Trigger serport:1.inbufferchanged SP1RXD

- SP1RXD -
00:00:00:01    Label Add New Data to Serial Buffer
00:00:00:01    Label FF CMD ALC PAG BTN
00:00:00:01    Label LoopLookingForSOM
00:00:00:01    If ( serport:1.inbuffercount = 0 ) End
00:00:00:01    If ( serport:1.inbuffer:1 = 0xFF ) Goto GotSOM
00:00:00:01    Delete serport:1.inbuffer:1
00:00:00:01    Goto LoopLookingForSOM
00:00:00:01    Label GotSOM
00:00:00:01    If ( serport:1.inbuffercount >= 5 ) Start GoMsg

- GoMsg -
00:00:00:01    Label Command Byte
00:00:00:01    Set var:1 serport:1.inbuffer:2
00:00:00:01    Label Data Byte 1
00:00:00:01    Set var:2 serport:1.inbuffer:3
00:00:00:01    Label Data Byte 2
00:00:00:01    Set var:3 serport:1.inbuffer:4
00:00:00:01    Label Data Byte 3
00:00:00:01    Set var:4 serport:1.inbuffer:5
00:00:00:01    Label Get Rid of Message
00:00:00:01    Delete serport:1.inbuffer:1-5
00:00:00:01    If var:1 = 1 Start Command1
00:00:00:01    If var:1 = 2 Start Command2
```

Command1 and **Command2** etc. tasks are not shown...

Display Resources (display)

Display resources allow you to display information, most often text. An LCD display often comes with the IMS, and it is display:1. Displays are row and column indexed, although you don't have to specify the row and column just to display something.

Display Commands

Write

Write <display> <string or var>

The **Write** command writes to the display resource. You can specify the row and column, just the row, or neither. Ultimately a string is being written, so you can either specify a string to display, or use a string variable.

```
00:00:00:01    Write display:1 “hello world”
00:00:00:01    Write display:1.row:1 “on the first row”
00:00:00:01    Write display:2.row:1 5 “row 2, start col 5”

00:00:00:01    Set var:1 “Recycle Your Animals”
00:00:00:01    Write display:1.row:2 var:1

00:00:00:01    Set var:2 13
00:00:00:01    Set var:3 44
00:00:00:01    FormatVar var:9 “Row:%d Col:%d” var:2 var:3
00:00:00:01    Write display:6.row:13 44 var:9
```

Button Resources (button)

Button resources are just like digital inputs, except they are specifically designated as physical buttons that are local to the IMS itself. You can, of course, attach buttons to wires that are connected to digital inputs on the IMS, and those certainly work the same way, but these buttons are fixed on the machine.

Our LCD display that often comes with an IMS has 7 buttons on them. They are numbered in this order to their labeling:

Button Index	Label
1	Left Arrow
2	Right Arrow
3	Up Arrow
4	Down Arrow
5	Menu
6	F2
7	F1

Button Commands

If and Trigger

If and **Trigger** commands were explained at the beginning of the manual, but here is a closer analysis with button resources. The valid state of button resources is either ON or OFF, logically either 1 or 0. The **If** command is *level-triggered*, and looks at the state of the button resource specified at the moment of the **If**. The **Trigger** command, on the other hand, is *edge-triggered*, and only starts the task if there is a transition between the opposite state and the desired state. If a button resource is OFF, and you want to trigger a task when it goes ON, the task will be started when the button resource goes from OFF to ON, and then does not trigger again until it goes from OFF to ON again.

Note that since button resources are pressed by humans, they are normally pushed and let go, which means the **IF** command is not that useful. The **Trigger** command on the other hand, is very useful with button resources.

Here are some examples of using **If** and **Trigger** with button resources:

```
00:00:00:01    If button:3 = ON Goto StartShow
00:00:00:01    If button:2 = OFF End
00:00:00:01    If button:2 = digoutput:2 Start InterlockOk
00:00:00:01    Trigger button:1 = ON
00:00:00:01    Trigger button:69 = FALSE
00:00:00:01    Trigger button:96 = 1
00:00:00:01    Trigger var:3 = button:4
```

One particularly beneficial use of the IF command with a button is for debouncing / deglitching. Any electrical input signal can have noise and signal level changes right around transitions, but buttons are even more susceptible to human pushes. It's quite common for a person to accidentally push twice, or too not push hard enough. Again, keep in mind, that the example procedures below for debouncing / deglitching work well with diginput resources as well.

Here is an example of using a **Trigger** command on a button push, and then ensuring the button is still depressed a tenth of a second later using the If command. This prevents false positives.

- AutoBegin -
```
00:00:00:01    Trigger button:3 = ON PossiblyGotButtonPush
```

- PossiblyGotButtonPush -
```
00:00:00:01    Label Wait 3 frames, then check button again
00:00:00:04    If button:3 = OFF End
00:00:00:04    Label It's now safe to consider button pushed
00:00:00:04    Label Do something here
```

Here is a similar example, but this time we also want to make sure that the person let go of the button. It's best to wait a much longer time, such as a full second, in case they're a slow button pusher.

- AutoBegin -

```
00:00:00:01    Trigger button:3 = ON PossiblyGotButtonPush
```

- PossiblyGotButtonPush -

```
00:00:00:01    Label Wait 3 frames, then check button again
00:00:00:04    If button:3 = OFF End
00:00:01:04    If button:3 = ON End
00:00:01:04    Label It's now safe to consider button pushed
00:00:00:04    Label Do something here
```

SMPTE Resources (smpte)

Multiple streams of SMPTE LTC can be utilized simultaneously, although it is most common to have only one stream.

SMPTE Reading Commands

Reading SMPTE is quite easy. It's the default setting, and there's really nothing to change. Just start triggering off the SMPTE time.

- AutoBegin -

```
00:00:00:01    Trigger smpte:1 = 03:00:00:00 MyShow
```

Although you should never need to change modes with a smpte stream, you can still do it. If you needed to switch back to reading from generating, you can do this:

```
00:00:00:01    Set smpte:1.mode read
```

SMPTE Generating Commands

There are 3 things you need to set up if you're going to generate SMPTE LTC. The first is to take the smpte stream out of its default mode of reading by changing the mode. The second is to set the framerate to the rate of choice, and the third is to set the time to begin generating from.

Valid values for the **mode** object property are:

read
generate

Valid values for the **framerate** object property are:

23.976
24
25
29.97
30
30drop

Then you can Start and Stop the smpte stream:

```
00:00:00:01    Start smpte:1
00:00:00:01    Stop smpte:1
```

You can also supply a generate time when you Start smpte.

```
00:00:00:01    Start smpte:1 01:00:00:00
```

This is equivalent to

```
00:00:00:01    Set smpte:1.generatetime 01:00:00:00
00:00:00:01    Start smpte:1
```

Here is a common example of setting up smpte in a show and stopping it.

- AutoBegin -

```
00:00:00:01    Set smpte:1.mode generate
00:00:00:01    Set smpte:1.framerate 29.97
00:00:00:01    Label Generate with 10 frame preroll
00:00:00:01    Set smpte:1.generatetime 00:59:59:20
```

- StartShow -

```
00:00:00:01    start smpte:1
```

- StopShow -

```
00:00:00:01    stop smpte:1
```

You can change the time that you're generating from, in the middle of generating. In this example, smpte will jump from 01:02:00:00 to 03:15:00:00.

```
00:00:00:01    start smpte:1 01:00:00:00
00:02:00:00    set smpte:1.generatetime 03:15:00:00
00:02:30:00    stop smpte:1
```

STUFF TO FIX / GO OVER IN MANUAL (by me probably)
Get rid of stuff that doesn't work anymore

STUFF TO FIX / GO OVER IN MANUAL (by the Nancinator)
Replace all occurrences of Property with Object

STUFF TO ADD by the Nancinator with Jeff's help

- **Binary operations and comparisons**
- **TCP/IP ???**
- **Remote I/O**
- **Sign**
- **DMX**
- **DMX playback**
- **Synapse**
- **Var textual operations mid/right/left and conversions .number**
- **Hopelessly drifting, bathing in beautiful agony**
- **I am endlessly falling, lost in this wonderful misery**
- **In peaceful sedation I lay half awake**

- **And all of the panic inside starts to fade**
- **Hopelessly drifting, bathing in beautiful agony…**
- **Video objects for saturation, white balance etc.**
- **Task stuff - .running. .result, params**
- **Email stuff**
- **Hysteresis, lifo, fifo**
- **Audssource audio**
- **Log Stuff**
- **Artificial Intelligence/Neural Networking – Open the pod bay doors Hal**
- **Aliases!!!!!!!!!!!!!!**
- **System.hour system.dow system.sunset system.sunrise system.latitude system.longitude, other GPS functions**
- **Random**
- **File i/o**
- **Backup/restore variables**
- **Etc. etc.**