

---

# SuperHub

IntelliScript V1.00  
Scripting Language  
Programmers Manual

---

**INTRODUCTION**

What is IntelliScript?.....	3
Editing IntelliScript Files.....	3
The SuperHub mechanism.....	3

**INTELLIScript IN DETAIL****Basic File Layout**

Overview of file structure.....	5
• Library functions.....	5
• System functions.....	5
• Passing data between functions.....	5

**Basic IntelliScript Syntax Rules**

Anatomy of a function.....	7
• Name.....	7
• Parameters.....	7
• Description.....	8
• Comments.....	8
• Body of the function.....	9
• Line termination.....	9
• Declaring Functions.....	9
• Nesting Functions.....	10

**Data Manipulation**

• Introduction.....	10
• Variable types and rules.....	10
• Operators overview.....	11
• String Manipulation.....	11
• Array-like behaviour.....	11
• Concatenation.....	12
• Midstr.....	12
• Strlen.....	12
• Converting numbers to strings.....	12
• Decision making.....	
• Conditional tests.....	13
• Loops.....	13

**Special Functions**

• Initialisation.....	14
• Serial Comms.....	14
o Configuring ports.....	14
o Sending data.....	15
o Receiving data.....	15
• SuperHub LEDs.....	15
• Real-time clock/calendar.....	16
• Delay.....	16
• Device-specific functions.....	16

**Intelliplate-Specific Features**

• Rotary controls.....	17
o Configuring encoders.....	17
o Reading encoders.....	17
o Setting encoders.....	17
• Detecting Intelliplate switches.....	18
• Driving Intelliplate on-panel LED indicators.....	18
• GetNoofPanels function.....	18

**De-bugging tools**

• Overview.....	19
• Configuring the Master RS232 port for de-bug functions.....	19
• The DebugPrint embedded function.....	19

**Further reading.....20****APPENDIX**

Embedded functions and events.....Language Limits.....Keywords.....Common Errors.....ASCII character codes.....
---

**INDEX**

## INTRODUCTION

### What is IntelliScript?

IntelliScript is a unique scripting language created by Audace Ltd for programming the SuperHub A/V controller. When an IntelliScript file is loaded into the SuperHub, it is compiled internally, resulting in binary data, which tells the SuperHub's processor, principally:

- How to handle data coming into the serial ports and
- When to send messages out, and in what format.

### Editing IntelliScript Files

IntelliScript files are simply text files, which observe the rules of the language\*. They can, if required, be created, read and edited within a simple text editor such as Microsoft® Notepad. This means, for example, that you could create sophisticated system designs using a simple PDA device with a basic text editor. No high-powered processor is required to create SuperHub-based project files.

Alternatively, a dedicated scripting interface – IntelliScribe – is provided free of charge by Audace Ltd. This provides a degree of assistance and error checking to enable you to verify scripts prior to upload into your SuperHub unit. This interface runs under Microsoft Windows® 95, 98, ME, 2000 and NT.

Finally, a variety of third-party general-purpose script editing tools (e.g. Visual Slick Edit) is also available which may be used for this purpose. However, these do not provide the dedicated tools included within the IntelliScribe application.

### The SuperHub Mechanism

An IntelliScript file consists of a collection of blocks of code, called *functions*. Each function has a particular role to play in determining how the SuperHub operates, like cogs in a machine. So one function might process incoming data from a signal processor, another function might deal with button pushes on a remote panel and so on.

Just as the cogs in a machine need to interconnect for the whole machine to operate, functions 'interconnect' by passing data back and forth to each other using function *calls*. So one function can invoke another function in this way. Functions also have the ability to receive and send data to/from the outside world, as we shall see later.

To illustrate how this works, a simple application of the SuperHub might be a system where a message is sent out of one of the serial ports when a remote button is pressed. To achieve this, you would need to create:

- A function, whose role is to recognise when a remote button has been pressed and to generate a message inside the SuperHub to indicate that this has happened. The function then passes this message to:
- An 'output' function which formats the message and transmits it to one of the SuperHub serial ports so that it can be understood by a particular piece of third party equipment.

In this instance, the first function is said to 'call' the second function. In other words, it passes data to the second function for an operation to be performed. The result is that pressing a switch somewhere in the building changes a setting within a piece of controllable equipment, with the whole process being managed by the SuperHub.

Most functions perform one or more of the following actions:

- Read, test and process data coming into the SuperHub serial ports

- Generate messages and present them to third-party devices via the SuperHub remote ports
- Perform mathematical operations on data
- Store information for future use – e.g. error logs, initialisation data, port settings, variable values and so on

All functions, regardless of their role within the system, observe the same rules, syntax and basic structure, greatly reducing the learning curve for this language.

*\*IntelliScript is based closely on the 'C' programming language but differs in several respects, in order to more closely suit the application and to simplify the process of creating functional scripts.*

---

## INTELLIScript IN DETAIL

### BASIC FILE LAYOUT

IntelliScript files consist of two broad groups of functions – library functions and system functions. Your script does not have to be organised in this way but it will certainly help you to troubleshoot and plan your system design if you observe this discipline. It will also enable us to provide quicker and more accurate troubleshooting support if required.

### Library Functions

Library functions are general-purpose blocks of code, which are typically re-used across several projects and are not specific to one particular project or 'site'. For example, you might be controlling a switching unit by a manufacturer called Acme DSP. In order to access facilities within this device, you will need a series of library functions – one for each facility you are using with the device (e.g one to fire presets, one to interrogate the device and so on). It is entirely possible that several projects would use the same set of library functions if one or more of the same devices are to be controlled.

Library functions may be provided by Audace Ltd, borrowed from other projects or created from scratch with reference to the control protocol specification for the device you wish to control.

### System Functions

These blocks of code are specific to a particular site or project. They might, for example, include definitions of Intelliplate panels (which, in turn, will call Library Functions), definitions of calendared events and so on. In general, this section of your script file will differ from one project to another.

System functions can be notionally sub-divided further into the following types of functions:

- Panel Definitions (if used with Intelliplate panels)
- Room Mode definitions
- Event Definitions

### Passing Data Between Functions

Information is passed between functions in the form of **variables**. Variables are data elements, which have a *name* and a *value*. The name is fixed but the value of the variable may be changed at any time. So, for example, you might have a text variable with a name `message`. At one point in time, the value of this variable might be "Hello World" and at another time, the value may change to "Goodbye Cruel World". Functions have the ability to read, set and perform tests and mathematical operations on variables, as we shall see later.

Variables are either numerical or text based – a characteristic known as the **type** of the variable. In order to be able to use a variable within your script, you first need to create it. This is known as *declaring* the variable. There are two ways to declare variables, the most simple of which is shown below:

```
type <variable name>;
```

So the type of the variable is followed by the name that you will use to refer to it. So for our text variable, we would declare it as follows:

```
string message;
```

This variable is then available to functions within the script to be set, tested and used within mathematical operations.

The other method for declaring variables is within 'function titles', which will be explained in detail in the next section.

*Note that, whichever method of declaration you use, variables must only be declared once or errors will occur.*

As will be seen within the next section, a value can then be assigned to a declared variable using the simple assignment operator e.g.

```
message = "Hello Bunty";
```

However, this must be done within the body of a function (see next section).

An example of where this would be used is in setting the power-up conditions for the system.

Another characteristic of variables that is of interest to programmers is its **scope**. This refers to the section of code within which this variable can be employed. Generally within programming languages, variables are either global or local. Global variables can be used anywhere within a programme and local variables only exist within the function that contains their declaration. In IntelliScript, all variables are global. This means that any time you change the value of a variable, this new value will apply to any subsequent use of the variable, *by any function*, until it is changed again.

---

## BASIC INTELLIScript SYNTAX RULES

### Anatomy of a function

Here is an example function, which we will refer to during the course of this section:

```
BiampInit()  
// desc: Initialise system for communicating with Biamp  
{  
    SetBaud(38400);  
    SetEndOfMsg(13);    // carriage return  
    SetMsgTimeout(2000); // 2000ms  
  
    portNo=1; // serial port number that biamp will reside on  
    unitNo=1; // Biamp unit number  
    dspbNo=1; // Biamp DSP block number  
    inputID=2;  
    outputID=3;  
}
```

This is a library function which initialises a SuperHub to communicate with a Biamp Systems Inc DSP device. You will find that all manufacturers of controllable equipment provide a specification for the control port of their device and you will need to create an initialisation function similar to this for each port on your SuperHub that you intend to use.

### Function Name

The very first thing that is required when creating a function is a unique **name**. This enables the function to be identified and 'called' (i.e. used) by other functions within your script. In the case above, the name of the function is `BiampInit`. The name of the function should be chosen carefully so as to allow you to quickly identify it at a later date. The following rules apply to the function name:

- Must be unique within your script
- Must be no more than 32 characters long
- Should include no spaces or other punctuation
- The first character should not be numerical (i.e. should be a letter)

### Function Parameters

The name is *immediately* followed by '**parameters**', enclosed within parentheses '()'. This area can be left empty, as in the case above, but the parentheses should still be present. Parameters are variable names, enabling data values to be passed between functions. So, for example, you might have a general-purpose function, which enables a preset to be fired in a device. The actual preset that you want to fire will vary, depending on which button has been pressed. So you would create a function with a first line something like this:

```
FirePreset(float PresetNo) //Note that the type of the parameter should precede  
its name
```

To fire a preset from within your script, you would simply set the value of `PresetNo` and call the `FirePreset` function, as shown below:

```
FirePreset(3);
```

And the function would run with the value '3' being substituted wherever the `PresetNo` parameter was used.

In other words, the FirePreset function uses a variable called PresetNo, whose value is passed to it by another function, such as one that recognises which button has been activated.

*Note that when a variable is used within the title line of a function in this way, this also serves to declare the variable, so no other declaration of that variable should exist within your script.*

A function can have up to eight parameters. When creating a function with multiple parameters, simply separate each parameter with a comma. For example:

```
FirePreset(float PresetNo, float DeviceID, string RoomNo)
```

Similarly, multiple parameters can be passed between functions.

*The function call should pass the same amount of data in the same order (using the same comma separators) as in the function definition, otherwise errors will occur.*

### Function Description

The second line of a function is a **description** of the function, in the form of a **comment** (see below). Like all comments, this information is not actually used by the SuperHub compiler. However, it is useful for troubleshooting purposes at a later date and the IntelliScribe editing interface also makes use of this information. We strongly recommend including a brief description of the function here, if only for your own reference.

### Comments

Comments can be inserted at any point in your script using the `//` characters. Note that the SuperHub compiler will ignore all characters from the beginning of the `//` comment declaration to the end of that line. Here are some examples of how to include comments within your code (the comments have been colored red for clarity):

```
// desc: Initialise system for communicating with Biamp
SetEndOfMsg(13); // carriage return
SetMsgTimeout(2000);
```

Note that in the following example, an error would arise as the code `SetMsgTimeout(2000);` would be ignored and treated as a comment.

```
// desc: Initialise system for communicating with Biamp
SetEndOfMsg(13); // carriage return SetMsgTimeout(2000);
```

Other comments may follow the description but the next item of functional code is preceded by a `{` character. Note that for each `{` there should also be a `}` at some point in your code and that if these are 'nested' then the compiler will always treat a `}` as relating to the most recent occurrence of `{`. This is a universal coding convention and needs to be very carefully observed.

You will dramatically boost the clarity of your layout, and reduce the likelihood of errors, if you ensure that each time you use a `{` the indentation of subsequent lines is increased by one 'tab' and that each time you use a `}` the indentation of that line and subsequent lines is reduced by one 'tab' as shown in the example below\*.

```
{
  SerialSendStr(portNo,"SETL ");
  SerialSendNum(portNo,unitNo);
  if (io==0) {
    SerialSendStr(portNo," INPMUTE ");
    SerialSendNum(portNo,inputID);
  } else {
    SerialSendStr(portNo," OUTMUTE ");
    SerialSendNum(portNo,outputID);
  }
}
```

```

    }
    SerialSendStr(portNo, " ");
    SerialSenNum(chanNo);
    SerialSendStr(portNo, " ");
    if (state) {
        SerialSendStr(portNo, "1\n");
    } else {
        SerialSendStr(portNo, "0\n");
    }
    BiampWaitAffirm();
}

```

*\*Note that the 'Process' facility within the IntelliScribe editing tool will indent your code according to this rule, helping you to find bracketing errors more easily. IntelliScribe will also alert you to any discrepancy between the number of open brackets and the number of close brackets.*

*Note that functions cannot be created, though they may be called, within other functions. So when you place a new function within existing code, ensure that the number of '{' characters matches the number of '}' characters up to that point.*

Good practise, when creating functions, would be to immediately follow a '{' with a '}' and then to place the remaining code between these elements.

### Body of Function

The **body of the function** then follows, which or may not include comments (we recommend the liberal use of comments, as this will help you when testing and trouble-shooting your code). There may be any number of '{ }' pairs within the body of the function but the end of the function is always denoted by a concluding '}'.

### Line Termination

Note that each line of code *within* the function is terminated by a semicolon. This character should precede any 'in-line' comments as shown below:

```

unitNo=1;    // Biamp unit number
dspbNo=1;   // Biamp DSP block number

```

The following line would result in an error:

```

unitNo=1    // Biamp unit number
dspbNo=1;   // Biamp DSP block number

```

as the compiler will 'see' this as

```

unitNo=1dspbNo=1;

```

The only lines not requiring a semicolon are:

- The 'title line' of your function
- Lines terminating in a '{' or a '}'
- Conditional tests (see later)
- Comment lines

Note that it is only within the body of functions that setting, initialisation and mathematical/logical manipulation of variable values can take place. To initialise variables on power-up, do so within the `HubInit()` function.

Note that the only valid operation outside of functions is declaring variables (and, of course, making comments).

### Declaring Functions

Note that a function must be declared or defined before being used. If required, a dummy declaration, or 'prototype', of the function may be placed in the script before it is defined e.g.:

```
func1(float a, string b);
```

### **Nesting Functions**

Functions may be used within other functions. Nesting is permitted up to '32' layers deep.

---

## **DATA MANIPULATION**

### **Introduction**

Within the body of your function, you are likely to need to do some or all of the following:

- Test for certain conditions and act in response to the result
- Set certain conditions which are then used elsewhere within the script
- Call other functions which make things happen
- Create information which will be used within the script

All of the above operations are performed on *variables*, which were introduced earlier. To enable us to perform these operations we have several programming devices at our disposal:

- Operators
- Conditional statements
- Loops

The first task is to consolidate our understanding of the rules surrounding variables that are available to us in IntelliScript.

### **Variables – types and rules**

As mentioned earlier, two *types* of variable are permitted in IntelliScript – string and floating point.

#### **String Variables**

String variables can be a maximum of 256 characters long and may contain any 8-bit character.

## Operators

Operators are essentially a series of familiar mathematical symbols, such as '=', '+', and so on, along with some less familiar but useful ones.

Here is a complete list of the operators available to you within the IntelliScript language:

Operator	Operation	Example
//	comment	// comment
/* ... */	comment	/* comment */
[]	index an array	a=str[4];
;	end of statement	a=5; // assign 5 to 'a'
{}	compound statement	f() {a=5; b=7; }
=	assignment	a=5; // assign 5 to 'a'
( )	parenthesis	a= (5+3) * 2; // a=16
~	bit-wise complement	a= ~b; // e.g. if b=0xaa, then a=0xff
-	unary arithmetic negate	a= -b; // e.g. if b=5 then a=-5
*	arithmetic multiply	a= 2*3; // a=6
/	arithmetic divide	a= 8/2; // a=4
%	arithmetic modulo	a= 23%5; // a=3
&	bit-wise AND	a= 12&7; // a=4 (1010b AND 0111b)
	bit-wise OR	a= 12 7; // a=15 (1010b OR 0111b)
^	bit-wise XOR	a= 12^7; // a=13 (1010b XOR 0111b)
+	arithmetic addition	a= 2+3; // a=5
-	arithmetic subtraction	a= 5-2; // a=3
!	unary logical NOT	if (!a) b=3; // if a==0 then b=3
&&	logical AND	if (a && b) c=3; // if a!=0 and b!=0 then c=3
	logical OR	if (a    b) c=3; // if a!=0 or b!=0 then c=3
==	equals	if (a==2) b=3; // if a is 2 then b=3
!=	not equals	if (a!=2) b=3; // if a is not 2 then b=3
>	greater than	if (a>2) b=3; // if a is greater than 2 then b=3
<	less than	if (a<2) b=3; // if a is less than 2 then b=3
>=	greater than or equal to	if (a>=2) b=3;
<=	less than or equal to	if (a<=2) b=3;

The detail of the use of all of these operators is beyond the scope of this document. These operators use commonly-accepted programming principles, such as those found within the C programming language, Javascript, PHP and others. Further details may be obtained within specialist publications for programmers, such as those available from Wrox ([www.wrox.com](http://www.wrox.com)), New Riders ([www.newriders.com](http://www.newriders.com)) and O'Reilly ([www.oreilly.com](http://www.oreilly.com)). We can provide more specific recommendations on request.

## String manipulation

The manipulation of string data within IntelliScript is extremely straightforward and provides a powerful way of interpreting and formatting incoming and outgoing data. A number of dedicated functions for dealing with strings are provided within the IntelliScript language. A brief explanation follows. Further illustrations may be observed within our library of samples scripts for the SuperHub.

### Array-like behaviour of strings

Strings can be accessed like 'arrays', ranging from 0-255. For example, to access the 5<sup>th</sup> character in a string:

```
float a;  
a = str[4];
```

If a string is accessed in this way, and the index is larger than the length of the string, then the string length is extended to accommodate.

*For further details of arrays, which lies beyond the scope of this document, we would recommend further reading from the wealth of general programming manuals on the market.*

### Concatenation

To concatenate (add together) two string variables, simply use the '+' operator. So if I had a string variable called msg1 and another called msg2 whose values were, respectively "Hello" and "World", I could add them together and place them in another variable, msg3, as follows:

```
msg3= msg1 + msg2;
```

Now, msg 3 would have the value "HelloWorld". I could achieve the same in several ways, such as:

```
msg3= msg1 + "World"; or msg3= "Hello" + msg2;
```

If we know the ascii character code for a character (see appendix), we can insert it into a string using the '\ ' operator. For example, the ascii code for a space is 32. So to put a space between the two words above, we could use:

```
msg3 = "Hello\032" + msg2; //msg3 is now set to "Hello World"
```

Note that the ascii character code should always be a three-digit number. See the appendix for a listing of ascii character codes.

We can add numerical variables to string variables in the same way and they will be treated as strings for that operation. For example, if we have a variable x whose value was 5 and we performed the operation:

```
msg= "Preset" + x;
```

The string variable 'msg' now has the value "Preset5".

**midstr** – Returns a subset of the contents of a string variable

```
midstr(string source, float start, float length)
```

'source' is the name of the variable to be operated upon; 'start' is the start-point within the string; 'length' is the number of characters to be 'extracted' from the string. E.g.:

```
string newword;
newword=midstr("Testing", 3, 4);
```

Will return "ting". Note that the very start of a string is position '0'.

**strlen** – Returns the length (no of characters) of a string variable

E.g.

```
float count;
count=strlen("Howdedoody");
```

Will return a value of 10.

### Converting numbers to strings

The **format** function has been provided to enable numbers to be converted to strings. The function syntax is as follows:

```
format(num, outputtype, reserved);
```

Example:

```
string portno;  
float rport;  
rport=3;  
portno = format(rport,1,0);
```

This will give the string `portno` the value "3".

## Decision-making in IntelliScript

### Conditional Statements

Conditional statements are the decision-makers within your code. For example, you can test the value of a variable and, depending on the result, take a different 'fork in the road', or, in other words, execute a different section of code.

In order to fire a preset in a piece of equipment, you might create a variable called `PresetNo` and a function called `FirePreset`, which acted on the value of `PresetNo`. Before calling `FirePreset`, you might want to check that `PresetNo` is in the range 1-16. You would do this as follows:

```
if (PresetNo<=16 && PresetNo>=1){  
    FirePreset(PresetNo);  
}
```

You might want to control what happens if these conditions are not met (error-trapping). This is done using the 'else' statement. For example, if `PresetNo` is not in the valid range, we might want to set an error flag:

```
if (PresetNo<=16 && PresetNo>=1){  
    FirePreset(PresetNo);  
}else{  
    errFlag=1;  
}
```

Many programming languages will provide an 'else if' facility within conditional statements. IntelliScribe's syntax for this is as follows:

```
if (PresetNo<=16) && (PresetNo>=1){  
    FirePreset(PresetNo);  
}else if(PresetNo==0){  
    errFlag=1;  
}else{  
    errFlag=2;  
}
```

Here, the preset will be fired if it is in the range 1-16 inclusive. Otherwise, if it is '0' then the `errFlag` variable is set to 1 and if it is any other value, the `errFlag` variable is set to 2.

### Loops

Loops enable you to repeat a section of code until a certain condition is met. For example, if you had a network of controllable devices on one of the serial ports and these devices each had a unique ID then you might want to repeat a section of code, once for each connected device.

The following loop statements are available within IntelliScript

```
while (condition){ }
```

For example, you might want to send a message out of a serial port 5 times. This would be done as follows:

```
x=1;
while(x<=5){
    SerialSend(1, "This is the message");
    x = x + 1;
}
```

What happens here?

First of all, the variable 'x' is initialised to the numerical value 1.

Then we enter a loop, which is repeated for as long as the variable 'x' is less than or equal to 5. However, each time we go through the loop (known as an *iteration*), the value of x is incremented. Each time the loop has been executed, the condition is re-tested and, if true, the loop runs again. The outcome is that the loop is executed five times because after the fifth iteration, the value of x is incremented to 6 and the condition is no longer true.

Great care should be taken not to create endless loops. An example of such a loop is shown below:

```
x=5;
while(x>=5){
    SerialSend(1, "This is the message");
    x = x + 1;
}
```

Here, the value of x is initialised at 5 and the loop will continue to run while x is greater than or equal to 5. After each iteration, x is incremented, so it will never go below 5!

The consequence of an endless loop is that the Rabbit processor inside the SuperHub would be permanently occupied with executing this loop and other functions within the script would not be able to run, which would mean that control of your a/v installation would completely break down. So you can see why great care should be exercised when creating loops.

---

## SPECIAL FUNCTIONS

Note that a complete list of *all* functions that are embedded within the IntelliScript language may be found in the Appendix.

### Initialisation

The HubInit function is used to set operation conditions for your installation. It is called automatically by the SuperHub on power-up.

### Serial Comms

A number of functions are provided for configuring and handling the serial ports fitted on the SuperHub.

#### Configuring

**SetBaud** – this allows you to specify the baud rate of a serial port. For example, to set the baud rate of RS232 port 1 at 19200, you would

do the following:

```
SetBaud(1,"19200");
```

Note that normally these should be part of the initialisation process, contained within the `HubInit` function.

In the case of the Ethernet port, this can be configured for Telnet operations, as some audio-visual devices allow control over Telnet. This is initialised using the `OpenTelnet` function. You simply provide the port number and IP address that you want to Ethernet port to communicate with. So, for example, to communicate over Telnet port 23 to a device with an IP address of 192.168.1.101 then you would simply initialise the port as follows:

```
OpenTelnet(23,"192.168.1.101");
```

Then you can use the `SerialSend` function as normal to send data to this device (`SerialSend` port no 9).

### **Sending data**

To send data to a serial port, you use the `SerialSend` function. So, for example, to send the message "System Reset" to port 3, you would do the following:

```
SerialSend(3,"System Reset");
```

### **Receiving data**

When incoming data appears on a serial port, the `SerialReceive` event is automatically fired. You should use this function in conjunction with the `SerialGet` function to deal with incoming data. For example, to echo messages coming into port 1 onto port 2 you would do the following:

```
SerialReceive(float rport)
{
    if (rport==1) {
        string message;
        message = SerialGet(1);
        SerialSend(2,message);
    }
}
```

Note that you can also define the criteria for a 'valid' message. These criteria will be used to determine when the `SerialReceive` event needs to be fired. You can provide values for the length of a message, the 'timeout' period and the character(s) used to denote the end of a message. See the Appendix table of embedded functions for more details and our library of sample scripts for some illustrations of the use of this facility.

### **SuperHub LEDs**

You have the facility to specify the function of the 4 'User LEDs' on the front panel of the SuperHub. By default the functions are: 1- Power; 2 – Panel not found; 3 – Script Activity; 4 – Serial Port Activity. This can be useful for carrying out diagnostic functions on your installation. For example, in order to turn User LED 1 'ON', you would use the following statement:

```
SetHubLed(1,1);
```

## Real-time clock/calendar

SuperHub includes a real-time, battery-backed clock and calendar, for automated control of the audio-visual system. Three basic 'alarm' types are provided for - easily extended within your script by manipulating the configuration string. However, the three basic alarm types are as follows:

**One-time** - This will fire once only and you can specify the date and time that this will happen. To set an event for 16:40 on the 12/12/03, you would do it as follows:

```
SetAlarm(1,"O 12/12/2003 16:40");
```

**Daily** - These events will fire on a daily basis at the time specified. You simply specify the time. So set a daily alarm at 18:00 as follows:

```
SetAlarm(2,"D 16:40");
```

**Weekly** - These events will fire once a week, starting on the day specified, at the time specified. So, for example, to perform an event every Tuesday at 4pm, starting on the 7<sup>th</sup> January you would do the following:

```
SetAlarm(3,"W 07/01/2003 16:00");
```

Note that the `SetAlarm` function is used to specify when the `Alarm` event is automatically fired within the SuperHub. You then use the `Alarm` event to determine what happens as a result of the 'event'. So, if you wanted to call the `FirePreset` function when the above weekly event occurs, do this as follows:

```
alarm(float alarmno){
    if(alarmno==3){
        FirePreset();
    }
}
```

You also have the facility to get the current time and/or date with the `GetTime` and `GetDate` functions. So you could use these within the `HubInit` function, for example, to ensure that automation proceeds as normal in the event of a power failure.

## Delay

Use this embedded function to introduce a delay your code. If, for example, you want to manage the rate at which messages are sent out of a serial port to a third-party piece of equipment. If you wanted, for example, to introduce a delay of 300 milliseconds into your code, you would do it as follows:

```
delayMs(300);
```

## Device-specific functions

For certain devices, it is more efficient for us to provide an embedded function than to construct your own functions, particularly if they involve complicated mathematics. For example, we have created the `libConvertBSS` function to format a string variable for output to a BSS Soundweb audio processor. The Soundweb protocol uses checksums and we found that more efficient performance could be achieved by creating a dedicated function rather than calculating the checksum 'by hand' each time a message has to be transmitted. Other custom functions can be provided on request.

---

## INTELLIPLATE-SPECIFIC FEATURES

A number of events and functions have been provided to enable you to more easily interface the SuperHub to our Intelliplate range of wall plates. These are as follows:

### Rotary controls

Intelliplate panels featuring programmable rotary controls are easily handled by SuperHub using IntelliScript's embedded functions and events.

#### Configuring Encoders

For the best performance, we recommend using the `ConfigEncoder` embedded function to set your encoders up. Using this facility will result in the best performance of your wall panels. Simply choose a panel and set a min and max value for the encoder, which usually corresponds to the values accepted by the protocol of the third-party device. So, for example, if you are controlling a device with a range of 0-255 then you would configure the encoder on panel 5 as follows:

```
ConfigEncoder(5,0,255,0);
```

Using this facility will automatically tie the corresponding LED column to this range of values.

#### Reading Encoders

Rotary controls on Intelliplate panels are easily read using the `Encoders` embedded event, which is called triggered whenever the SuperHub detects that a rotary control has been manipulated. The format for this event is:

```
Encoders (float PanelNo, float Change){  
}
```

The first parameter (`PanelNo`) is the address of the panel that has been manipulated. The second parameter (`Change`) is the no of 'clicks' that have been detected. Each rotary control issues 48 clicks for every 360 degrees of rotation. If `ConfigEncoder` has set a min and a max value for this encoder, then SuperHub will automatically calculate the new value for that encoder, within the range specified, corresponding to the amount of rotation applied to the control. This is the most efficient way to handle encoders, as it allows you to immediately take the new value for the encoder 'position' and send this out of a serial port to adjust a device.

For example:

```
Encoders(float epanel, float change){  
    if (epanel==1) {  
        SoundwebLevel(1,1,change); // output channel 1  
    } else if (epanel==2) {  
        SoundwebLevel(1,2,change); // output channel 2  
    }  
}
```

#### Setting Encoders

Encoder values are set when they are turned but you can also set them within your code. This would be useful if, for example, you wanted one panel to follow the action of another panel. This is done using the `SetEncoder` function. This is used in conjunction with the `ConfigEncoder` function, where the min and max range are set. So, if you had an encoder on panel 6 with a range of 0-255 and you wanted to set its value to 123, you would do it as follows:

```
SetEncoder(6,123);
```

## Switches

Switch actions are detected using the `Buttons` embedded event. This event is triggered whenever the SuperHub detects that an Intelliplate panel switch has been manipulated. The format for this event is:

```
Buttons (float PanelNo, float button, float state){
}

```

The first parameter is the address of the panel that has been manipulated. The second parameter is the 'number' of the switch within that panel which has been manipulated (1-8). The third parameter detects which action has been performed on the button as follows:

1 = Pressed  
0 = Released

So if you wanted to detect when Button3 on Panel 2 had been pressed you would do it as follows:

```
Buttons (float PanelNo, float button, float state){
    if ((PanelNo==2)&&(button==3)&&(state==1)){
        SerialSend(2, "Got it!");
    }
}

```

## LEDs

Intelliplate Panel LEDs are activated in one of two ways, depending on whether they are associated with rotary controls (LED columns) or with switches (individual LEDs) – the `SetEncoderLeds` and `SetButtonLed` embedded functions. The syntax for these is as follows:

```
SetEncoderLeds (float panelno, float level){
//panelno = panel address (1-31)
//level = LEDs to illuminate (0-6)
}

```

Note that if the `ConfigEncoder` function has been used to set a rotary up, then you will not need to use the `SetEncoderLeds` function as the LED column will automatically indicate the value of the encoder function within the specified min and max range.

```
SetButtonLed (float panelno, float button, float state){
//panelno = panel address (1-31)
//button = button on panel (1-8)
//state = LED status (1=ON, 0=OFF)
}

```

So, for example, if we wanted to turn the LED associated with button 1 on for Panel 3, we would do it as follows:

```
SetButtonLed (3, 1, 1){
}

```

## Panels

A function – `GetNoofPanels` – is provided to enable you to use the number of panels that that SuperHub has been programmed to control. This could be used, for example to write a block of code which updates all current panels.

---

## DE-BUGGING TOOLS

### Overview

There are several methods that you can use to assist in de-bugging your scripts and systems.

1. First of all, there are some rudimentary tools provided in the IntelliScribe authoring software, such as checks for matching pairs of brackets, indentation and so on.
2. Secondly, a compilation result report is generated when the SuperHub compiles your script. This can be helpful in highlighting errors within your script.
3. You can use indicators within your system, such as the front panel LEDs on the SuperHub to indicate certain conditions.
4. The automatically generated Error Log may provide information which you can use to troubleshoot your installation
5. Finally, you can deliberately put code into your script which is used for de-bugging purposes. For example, you could have a message sent out of one of the serial ports indicating information of interest such as variable values, events being triggered, incoming data and so on.

### Configuring the Master RS232 Port for De-bug Functions

It may be that no serial ports can be 'spared' for de-bugging purposes so we have provided a facility to use the front-panel Master RS232 port as a de-bugging port during configuration of your system. In order to do this, you will need to put the Master RS232 port into 'de-bug mode'. This is done using the menu system interface on that port. This done by first selecting option '2. Scripting' and then '3. Enter debug mode'.

### The DebugPrint Function

Then you will need to put instructions into your script to send relevant information to the de-bug port. This is done using the `DebugPrint` embedded function. The syntax for this is very simple and is shown below:

```
DebugPrint(string debugstring);
```

The parameter 'debugstring' is simply the information that you want to output from the debugport. So, for example, if you wanted to send out an alert every time a valid message was received on serial port 1, you would do it as follows:

```
SerialReceive(float x){  
    if(x==1){  
        DebugPrint("Got a message on Port 1!");  
    }  
}
```

You could make this more sophisticated by outputting the actual data that had just come in, as follows:

```
SerialReceive(float x){  
    string incoming;  
    string outgoing;  
    If(x==1){  
        incoming = SerialGet(1);  
        outgoing = "Just got this message on Port 1 - " + incoming;  
        DebugPrint(outgoing);  
    }  
}
```

```
}  
}
```

So each time a valid message is received on Serial Port 1, the message will be echoed out of the Master RS232 port, whenever that port is in Debug mode, for you to inspect using a simple terminal emulator program.

Note that when the Master RS232 port is not in debug mode, the SuperHub processor will ignore `DebugPrint` instructions.

---

## FURTHER READING

We would refer you in the first instance to our dedicated web site – [www.intelliplate.com](http://www.intelliplate.com) - which contains a wealth of information useful in familiarising yourself with our products. With regard to IntelliScript, this includes software, sample scripts, tutorials and more. The following publications are available for download:

- SuperHub Cut Sheet
- IntelliScribe Authoring Software and manual
- SuperHub installation manual
- IntelliScript sample scripts

General programming manuals are available from a wide range of specialist publishers, which may be found on web sites such as [amazon.com](http://amazon.com), [www.barnesandnoble.com](http://www.barnesandnoble.com) and so on.

Finally, a multitude of resources are available free of charge on the internet, including discussion groups, tutorials, code examples and more. We would strongly recommend taking the time to research here for any advanced information you require about general programming principles. Using a search engine such as [www.google.com](http://www.google.com) and entering keywords such as 'php scripting tutorial' will reveal all manner of useful information, which you can bring to bear on your project.

---

## APPENDIX

## Embedded functions

Function	Parameters	Description
DebugPrint(string debugstring)	debugstring - string to be output	Outputs debugstring from the front-panel RS232 port, when in debug mode.
SetBaud(float portno, string baud)	portno - 1-6 for RS232; 7-8 for RS485 baud (for RS232) - any standard baud 300-115200 baud (for RS485) - any standard baud 600-230400	Set baud rate of serial port.
SerialSend(float portno, string output)	portno - 1-6 for RS232; 7-8 for RS485; 9 for Telnet output - string to send out of serial port	Transmits data out of serial port.
string SerialGet(float portno)	portno - 1-6 for RS232; 7-8 for RS485; 9 for Telnet	Returns next message for processing from serial port - or the next 256 characters of the message if it is too long.
SetEndOfMsg(float portno, float eom)	portno - 1-6 for RS232; 7-8 for RS485; 9 for Telnet eom - character marking end of message (0-255)	Defines how received messages end for a particular serial port. If set to zero (default) then the feature is disabled.
SetMsgTimeout(float portno, float time)	portno - 1-6 for RS232; 7-8 for RS485; 9 for Telnet time - length of 'silence' in milli-seconds marking end of message (0-10000)	Defines timeout for a particular serial port. Defaults to 100ms
SetMsgLength(float portno, float length)	portno - 1-6 for RS232; 7-8 for RS485; 9 for Telnet length - defines the length of a message (0-255)	Defines the length of received messages for a particular serial port. If set to zero (default) then the feature is disabled.
SetEncoderLeds(float panel, float level)	panel - panel number to control (1-31) level - number of leds to switch on (0-6)	Displays level on encoder leds on wall panels.
SetButtonLed(float panel, float button, float state)	panel - panel number to control (1-31) button - which button's led to control (1-8) state - whether to switch led on (1) or off (0)	Control individual button leds on wall panels.
SetHubLed(float ledno, float state)	ledno - which led to control (1-4) state - whether to switch led on (1), off (0) or system function (2)	Control user leds on the front of the SuperHub
string format(float num, float format, float x)	num - number to convert to string format - format of string: 0 - binary; 1 - integer; 2 - float x - reserved for future use (set to 0)	Convert a number (float) into a string.
string midstr(string src, float start, float length)	src - input string start - start point for new string (0 is first character) length - length of new string	Create a new string from a section of the input string
float strlen(string src)	src - input string	Returns the length of the input string
float GetNoofPanels()		Returns the number of panels the superhub is configured for.
DelayMs(float time)	time - number of milli-seconds to wait (0-5000)	Wait for 'time' number of milli-seconds.
SetAlarm(float alarmno,	alarmno - which alarm to set (1-8)	Set alarm. This will

string config)	config - configuration string: "O dd/mm/yyyy hh:mm" - one time alarm alarm "D hh:mm" - daily alarm "W dd/mm/yyyy hh:mm" - weekly alarm	cause "Alarm()" to be called at the specified time.
OpenTelnet(float port, string ipaddress)	port - IP port number (0-65535) ipaddress - IP address to connect to, e.g. "192.168.1.101"	Open telnet port, this must be done before calling "SerialSend(9,ssss)"
string GetTime()		Returns current time as a string in hh:mm:ss format.
string GetDate()		Returns current date as a string in dd/mm/yyyy format.
ConfigEncoder(float panel, float min, float max, float law)	panel - panel number to control (1-31) min - minimum output value max - maximum output value law - reserved for future use (set to 0)	This function configures an encoder to automatically update encoder leds. It also configures the encoder to output an <i>absolute</i> value rather than just changes (see 'Encoders' event below)
SetEncoder(float panel, float value)	panel - panel number to control (1-31) value - value to set level	This function allows the setting of the internal <i>absolute</i> value of an encoder, assuming ConfigEncoder has already been called.
string libConvertBSS(string input)	input - string to convert for sending to BSS unit  (e.g. BSS Soundweb)	Returns original string with STX, ETX and checksum added. Also, control characters are converted appropriately.

### Embedded Event list

Event name	Description
HubInit	This event is called at power-up.
SerialReceive(float x)	This event is called when a complete message has been received on serial port x.
Alarm(float x)	This event is called when alarm x has occurred.
Encoders(float panel, float value)	This event is called when a rotary is turned. 'panel' is the panel number (1-31) and 'value' is the number of clicks turned (positive for clockwise rotation, negative for anti-clockwise rotation, there are 48 'clicks' in a full rotation). Alternatively, if 'ConfigEncoder' has been called for this panel, then 'value' is a number ranging between 'min' and 'max' (from the ConfigEncoder call), describing the <i>absolute</i> position of the encoder.
Buttons(float panel, float button, float action)	This event is called when a button is pressed or released. 'panel' is the panel number (1-31), 'button' is the button number pressed (1-8) and action is whether the button was pressed (1) or released (0).

## Language Limits

### Functions

- Functions can have up to eight parameters – data must be ‘returned’ through global variables.
- Functions must either be declared or defined before use. Declaring a function is to describe the function without writing it, e.g.:

```
func1(float a, string b);
```

This is so the compiler knows the name (in this case func1).
- Functions may be ‘nested’ up to 32 deep.

### Variables

- All variables are globally available, i.e. any variable can be accessed from anywhere else in the script.
- Variables must be declared before they can be used.

### Names

- ‘Names’ include function names and variable names – they share the same storage.
- Names must start with a letter (a-z, A-Z), subsequent characters can be numbers or letters (a-z, A-Z, 0-9) but no punctuation.
- Names must not exceed 24 characters.
- Names are not case sensitive, i.e. Msg and msg are the same variable.
- Every function and variable must have a unique name.
- Names must not conflict with keywords (listed below).
- Names must not conflict with embedded function names (listed below).

Examples of **valid** names:

message, message2, PresetNo

Examples of **invalid** names:

232port, message\$, thisvariablenameiswaywaywaytoolong

### Keywords

if, else, while, float, string

### Common Errors:

Here are some pitfalls that you may fall into when first familiarising yourself with the IntelliScript language:

Variables:

**Multiple declarations** – remember that if a variable is used by a function as a parameter, it should not:

- Be declared anywhere else
- Be used as a parameter within any other function (other than function calls, of course)

**No declaration** – all variables must be declared in one of the following ways:

- Explicitly e.g.

```
float x;
```
- Implicitly within a function title e.g.

```
BiampMute(float muteio, float state){}
```

**Usage precedes declaration** – note the sequence in which variables are declared, initialised and used within your script is very important. You cannot use or initialise a variable until it has been declared.

**Invalid initialisation** – variable initialisation should only take place within function bodies e.g.

```
BiampInit()  
// desc: Initialise system for communicating with Biamp  
{  
    inputID=2;  
    outputID=3;  
}
```

Note that the only valid operation outside of functions is declaring variables (and, of course, making comments).

## Functions

**Invalid Name** – A function name should be short, have no punctuation, start with a letter and be unique within your script. Take care not to use the names of embedded functions, which you may not necessarily be using, or be aware of, in your script. For a listing see Appendix.

**Parameter Mismatch** – When calling a function you should ensure that the variable *quantity*, *type* and *sequence* matches those in the title line of the function being called.

**Bracket Mismatch** – Ensure that all opening brackets, of any type, are complemented by the same number of closing brackets within the function.

## Conditional Statements

**Equality tester** – note that in using the equality tester within 'if' statements, the assignment operator often gets used by mistake. So the syntax should be `if(a==b){}` rather than `if(a=b){}`.

'Else If' and 'If Then' used – though common in other languages, these statements are not valid in IntelliScript and should be replaced simply by `else{}` and `if{}` respectively.

**Incorrect nesting** – Remember to indent correctly and that '}' always relates to the last occurrence of '{'. Using the IntelliScribe tool may help you to avoid this problem.

## Loops

**Endless loop** – this will kill your script and you need to be very careful that your condition will always be met at some point in the near future when entering a loop.

**Condition met too early** – note that it is common for mistakes to be made in setting the test conditions for the loop. It take care to ensure that the conditions are not met earlier than you want them to by careful use of the <, > and = operators.

**ASCII Character codes**

Decimal (DEC), Hexadecimal (HX), and Octal (OCT) codes for ASCII Symbols (Sym)

DEC	HX	OCT	Sym	DEC	HX	OCT	Sym	DEC	HX	OCT	Sym	DEC	HX	OCT	Sym
000	00	000	NUL	032	20	040	SP	064	40	100	@	096	60	140	`
001	01	001	SOH	033	21	041	!	065	41	101	A	097	61	141	a
002	02	002	STX	034	22	042	"	066	42	102	B	098	62	142	b
003	03	003	ETX	035	23	043	#	067	43	103	C	099	63	143	c
004	04	004	EOT	036	24	044	\$	068	44	104	D	100	64	144	d
005	05	005	ENQ	037	25	045	%	069	45	105	E	101	65	145	e
006	06	006	ACK	038	26	046	&	070	46	106	F	102	66	146	f
007	07	007	BEL	039	27	047	'	071	47	107	G	103	67	147	g
008	08	010	BS	040	28	050	(	072	48	110	H	104	68	150	h
009	09	011	TAB	041	29	051	)	073	49	111	I	105	69	151	i
010	0A	012	NL	042	2A	052	*	074	4A	112	J	106	6A	152	j
011	0B	013	VT	043	2B	053	+	075	4B	113	K	107	6B	153	k
012	0C	014	NP	044	2C	054	,	076	4C	114	L	108	6C	154	l
013	0D	015	CR	045	2D	055	-	077	4D	115	M	109	6D	155	m
014	0E	016	SO	046	2E	056	.	078	4E	116	N	110	6E	156	n
015	0F	017	SI	047	2F	057	/	079	4F	117	O	111	6F	157	o
016	10	020	DLE	048	30	060	0	080	50	120	P	112	70	160	p
017	11	021	DC1	049	31	061	1	081	51	121	Q	113	71	161	q
018	12	022	DC2	050	32	062	2	082	52	122	R	114	72	162	r
019	13	023	DC3	051	33	063	3	083	53	123	S	115	73	163	s
020	14	024	DC4	052	34	064	4	084	54	124	T	116	74	164	t
021	15	025	NAK	053	35	065	5	085	55	125	U	117	75	165	u
022	16	026	SYN	054	36	066	6	086	56	126	V	118	76	166	v
023	17	027	ETB	055	37	067	7	087	57	127	W	119	77	167	w
024	18	030	CAN	056	38	070	8	088	58	130	X	120	78	170	x
025	19	031	EM	057	39	071	9	089	59	131	Y	121	79	171	y
026	1A	032	SUB	058	3A	072	:	090	5A	132	Z	122	7A	172	z
027	1B	033	ESC	059	3B	073	;	091	5B	133	[	123	7B	173	{
028	1C	034	FS	060	3C	074	<	092	5C	134	\	124	7C	174	
029	1D	035	GS	061	3D	075	=	093	5D	135	]	125	7D	175	}
030	1E	036	RS	062	3E	076	>	094	5E	136	^	126	7E	176	~
031	1F	037	US	063	3F	077	?	095	5F	137	_	127	7F	177	DEL

See [www.asciitable.com](http://www.asciitable.com) for more information

## INDEX

**A**

Alarm event 16  
 AND operator 11  
 ASCII codes 25  
 Automated events 16

**B**

Bitwise operators 11  
 Buttons event 18

**C**

Calling functions 5  
 Calendar events 16  
 Comments 8  
 Concatenation of strings 12  
 Conditional Statements 13  
 ConfigEncoder function 17  
 Converting numbers to strings 12

**D**

Data types 10  
 Debugging your code 19  
 DebugPrint function 19  
 DelayMs function 16

**E**

Embedded events 22  
 Embedded functions 21  
 Encoders event 17

**F**

Format function 12  
 Functions
 

- Naming 7
- Calling 7
- Passing parameters 5
- Limits 23

**G**

GetDate function 16  
 GetNoofPanels function 18  
 GetTime function 16

**H**

HubInit event 14

**I**

If...else statements 13  
 In-line comments 8  
 Intelliplate panels 17  
 IntelliScribe Editor 3

**K**

Keywords list 23

**L**

LEDs
 

- Encoder LEDs 18
- SuperHub User LEDs 15
- Switch LEDs 18

 LibconvertBSS function 16  
 Library functions 5  
 Line termination 9  
 Logical operators 11  
 Loops 13

**M**

Message formatting 11  
 Midstr function 12

**N**

NOT operator 11

**O**

OpenTelnet function 14  
 Operators 11  
 OR operator 11

**P**

Parameters 7  
 Passing variables 5

**R**

Real-time clock 16  
 Rotary controls 17

**S**

Scheduled events 15  
 Scope of the document 26  
 SerialGet function 15  
 Serial Ports 14  
 SerialReceive event 15  
 SerialSend function 14  
 SetAlarm function 16  
 SetBaud function 14  
 SetButtonLed function 18  
 SetEncoder function 17  
 SetEncoderLeds function 18  
 SetEndofMsg function 14  
 SetHubLed function 15  
 SetMsgLength function 14  
 SetMsgTimeout function 14  
 Strings 10  
 StrLen function 12  
 Switches 18  
 System functions 5

**T**

Telnet port 14  
 Text editors 3  
 Troubleshooting 19

**V**

## Variables

- Declaring 5
- Setting 5
- Types 5
- Scope 5
- Operations on 10
- Passing 5

**W**

while loops 13

---

**Scope** – this document (revision 1.00) relates to Audace Ltd's SuperHub running embedded software V1.00

**Contact** –

Audace Ltd, 50 Mt Ambrose, Redruth, Cornwall UK TR15 1RA  
 Tel: +44(0)1209 214147  
 Fax: +44 (0)870 705 1692  
 Email: sales@audace.co.uk  
 Web: www.audace.co.uk

© 2003 Audace Ltd, E&OE.