dScript

User Manual Version 4

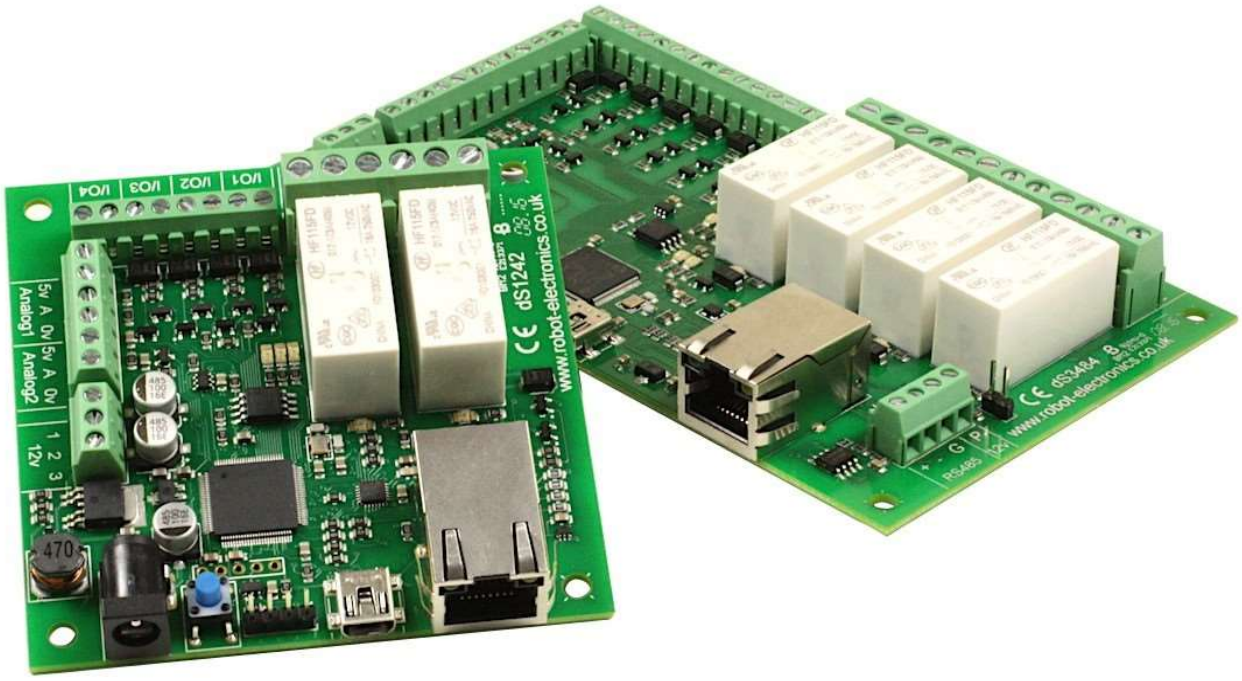


Table of Contents

Documentation History

V4.06	Added support for dS2832 module. Fixed bug where access to local variables that had the same name as global variables would access the global one.
V4.07	Added string compare Changed threadsleep so zero restarts immediately, previously never restarted. Added system.ThreadHandle to retrieve handle to current thread Added system.SleepTimer(ThreadHandle) to get a threads sleeptimer (read only) Added Saving of the Flash configuration variables (Module → Save/Restore)
v4.08	Added file associations Bugfix – NTP (Real time clock) startup failed under some circumstances.
V4.09	Added 3 new commands to Binary and AES Binary command sets to control all relays in one command. Set all, set selected and clear selected.
V4.10	No change. Updated to match application version with dSx42 modules.
V4.11	Added support for notifications in the supplied apps.
v4.12	Bugfix – Adding steps to the sequencer could crash the board.
V4.13	Bugfix – Notifications sometimes incorrectly reported R1 changed.
V4.14	Bugfix – Flex input (used on dS378, dS2824, dS2832 & TCP184) notifications gave continuous outputs.

The cheat sheet, quick reference guide

Variable types

int8	int16	int32	string
eeint8	eeint16	eeint32	eestring
flint8	flint16	flint32	flstring

Array's

int8[w]	int8[w,x]	int8[w,x,y]	int8[w,x,y,z]

System variables

system.despatchcounter
system.ModuleID
system.VerMajor
system.VerMinor
system.FlashPending
System.Random

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Modulus
<<	Shift left
>>	Shift right
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Assignment operators

=	Assignment
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
//=	Modulus
<<=	Shift left
>>=	Shift right
&=	Bitwise AND
|=	Bitwise OR
^=	Bitwise XOR

Logical operators

>	Greater than
>=	Greater or equal
<	Less than
<=	Less or equal
==	Equal
!=	Not equal

Command keywords

if	then	elseif	else	endif
thread	endthread	function	return	endfunction
for	to	next		
do	loop	while	until	
select	case	else	endselect	
threadstart	threadsleep	threadsuspend		

I/O objects

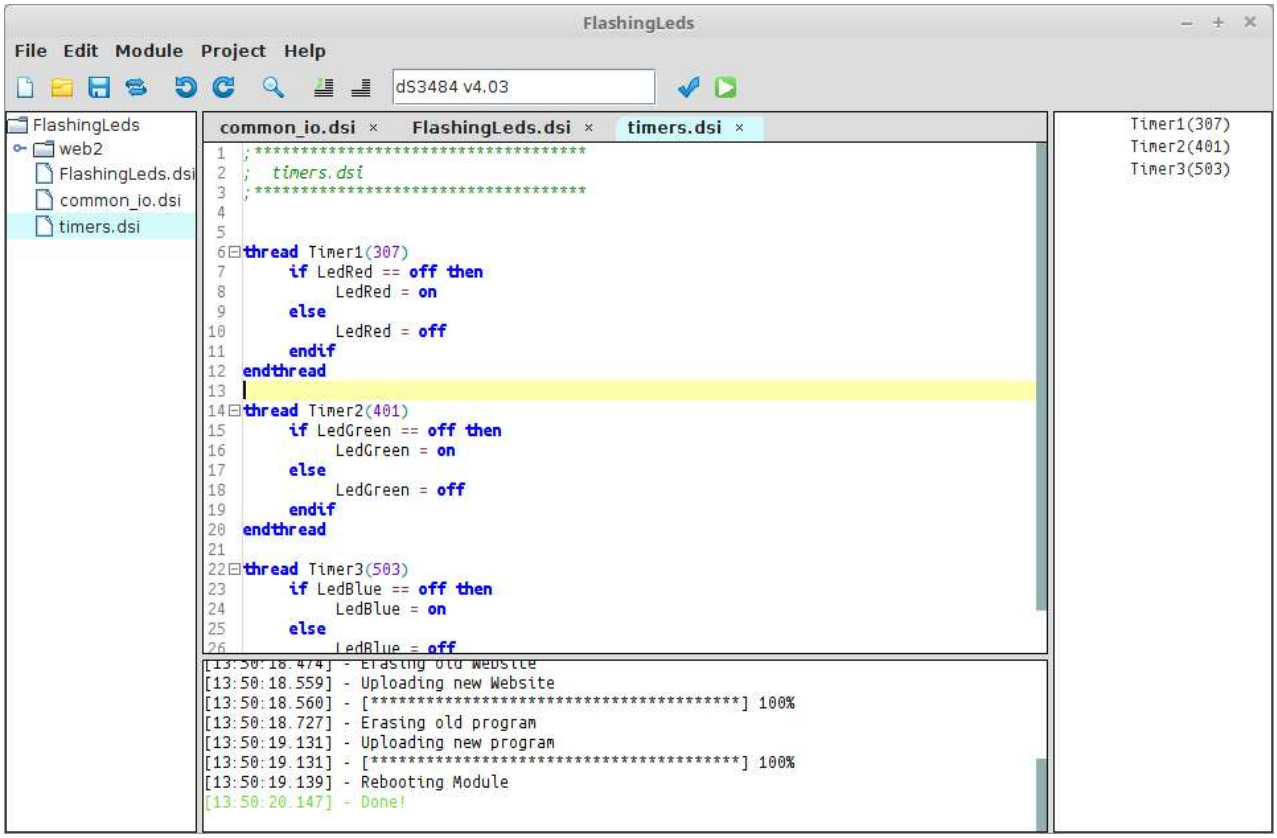
analogport	digitalport	flexport	serialport
tcpip	utc	http	email
easymail			

Introduction

This document describes the dScript programming language that underpins all of our dScript modules.

dScript, derived from devantech Scripting language, is a multi-threaded language for our internet connected modules. dScript compiles to efficient byte codes which are interpreted by the modules on-board runtime firmware. dScript is an editor, compiler and programmer for all dScript enabled modules.

Although dScript is not BASIC, the syntax of dScript will be familiar to anyone who has used Visual Basic or the small Basic chips and modules that are available.



Getting started

This section will guide you through downloading and installing the dScript IDE, connecting a dScript module, compiling and uploading firmware to the module and getting a first look at controlling the module from a webpage.

As from version 3.01, the dScript ide is available for Windows, Linux and macOS. Download the version for your system. For Linux select the deb or RPM version is available

PC requirements

Windows 7 or later, Linux or macOS
Network connection for viewing web pages
USB port to program the module.
You will also need an HTML editor of your choice.

Download and install the dScript IDE software as appropriate for your OS. You should remove the old version before installing the new one.

Support files

https://www.robot-electronics.co.uk/dscript.html

The dScript support files are supplied as a zip file that can be download and unzipped into a temporary folder, inside the temporary folder will be four folders:

Documentation
Examples
Utilities

The Documentation folder contains a copy of all dScript manuals. Note that IDE version numbers will match documentation version numbers. The IDE version can be found in Help→About.

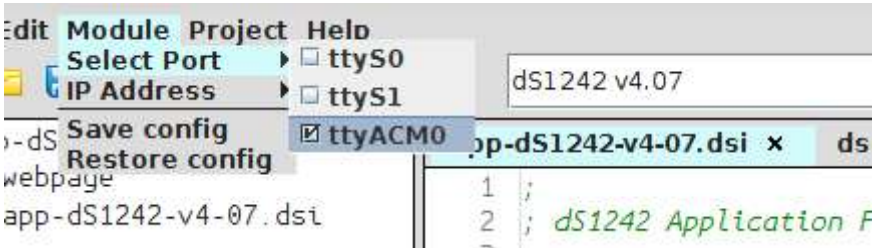
Copy the Examples directory to a convenient location on your computer, it contains both dScript source code examples and associated web pages.

The utilities folder contains the Devantech Module Finder as well as C# and java examples for all dScript modules.

Connecting the hardware

Connect an Ethernet cable from your network to the Ethernet connector on the dScript module, then attach a USB cable from the PC to the USB connector on the module. Now plug in the 12vdc power supply into the DC jack connector and switch on. If prompted by windows navigate to the USBdriver folder and install the USB driver.

Now run the dScript IDE and it should find your module and report its firmware version in the tool strip.



If it states “none”, go to Module→Select port→ and select the port you have the module connected to. This will be remembered for next time.

First program

A nice simple and colourful first program is the FlashingLeds example. Click the open button (or go to File->Open) and navigate to Examples->FlashingLeds and open the FlashingLeds.dsj file. *.dsj are dScript project files. The white triangle on the green button is the build, load and run button, click this to compile and run the program on the module. You should see a non-repeating pattern of Led flashing (well it does repeat, but you'll get bored watching it long before it does).

Second program

Lets try something more interesting, open the WebsiteGauges project from the examples directory. Look at the program and note the tcpip.ip address, you may need to change this and the subnet mask below to suit your network. Specifically, you need to make sure the PC is on the same subnet as the module. In this case 192.168.0.x and that the IP address is available. Once you've checked the IP address, you can build and run the program. When the process is completed you should now run your web browser and enter address 192.168.0.137/index.htm.

The left gauge measures the board temperature using the on-board temperature sensor. The right gauge is the boards supply voltage.

Click the buttons, and the relays on the board will toggle.

dScript IDE

The dScript IDE (Integrated Development Environment) is used to write the programs that run on the modules, it has syntax highlighting to make the programs easier to read and invokes the program and website compilers. All threads and functions are listed in the right panel and are for easy navigation of your code. Clicking a name will take you to that section in the main editor window. The message panel shows the status of the compiler, programmer and any errors found.

Most of the buttons should be fairly obvious to you, they are:

New, Open, Save, Swap, Undo, Redo, Find, Comment & Uncomment.

The Swap button swaps between the current and previous projects you have opened, this is useful when referring or copying from another project. The File menu also includes "Save As", and the Find button includes options for replacing words. The Undo and Redo buttons will be greyed out when there is nothing to undo/redo.

The comment buttons Comment & Uncomment are for selected blocks of text, which is very useful when testing programs as you can easily choose to ignore sections.

After the text box showing which module is currently connected, is the build button (blue tick), this will build your program and website but will not upload them to the module, reporting any errors found.

The white triangle on the green background is the build, load and run button. It first checks the modules runtime version and if this is different to that required by this version of the editor, it will automatically upload the new firmware. The user program and any associated website are compiled and uploaded to the module, and run.

Version numbers

The version numbers of the Manual (shown top right in the header above), the IDE and the module firmware should all be synchronised and show same version.

For example:

User Manual v4.03
IDE v4.03 (found in Help->About)
Module Firmware v4.03 (displayed in IDE toolbar when connected to module)

A copy of the module firmware is built into the IDE, it is compared to the module version when you load your dScript program, if needed it will then automatically update the module firmware to the IDE's version. Therefore any updates are taken care of automatically, you can even revert to a previous version by using the older IDE.

Compiler

There are actually two compilers.

The first compiles your code, then it translates your program into a byte code sequence which is loaded onto the module.

The second compiles your website to an efficient form to store in the on board flash chip. The compiler also inserts the necessary AJAX scripts which keep live data refreshed on the web page, additionally it inserts the security scripts which prevent unauthorised devices accessing your module.

Both compilers are built into and are part of the dScript editor.

Conditional Compiling

#ifdef
#else
#endif

These three compiler directives may be used to conditionally compile sections of code. See the "FlashingLeds" example where we use them in the common_io file to select the correct I/O definitions for the various modules.

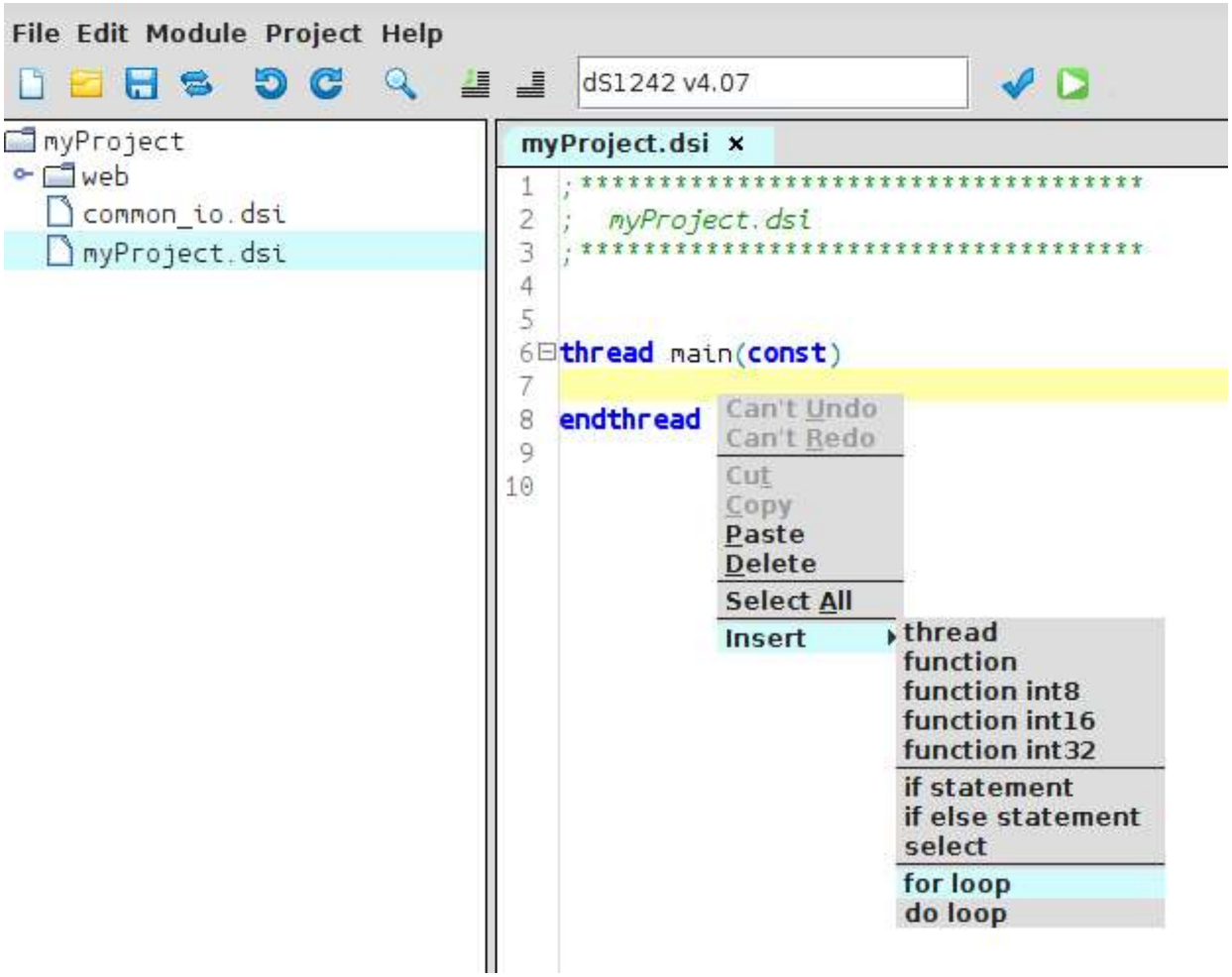
For example:

```
#ifdef dS1242 || dS2824 || dS3484 || dS378
   analogport TS1      100          ; on-board temp sensor
   analogport PSU      101          ; DC power voltage
#endif
```

Project structure

dScript projects are a folder containing a project file (*.dsj, one or more code files (*.dsi) and a sub folder for the website files. The File→New Project dialog will create all this for you. This will contain two code files, common_io.dsi has all the definitions for the relays, I/O etc. The other has the same name as the project and contains an empty "main" thread ready for you to start writing. The starting point for your code is the thread "main". It need not be the first thread, but you must include it somewhere in your code. You can create new code files as required, or import code files from other places. Imported files are copied to your project folder, the original is untouched.

Right clicking anywhere in the editor window and selecting insert from the sub menu will give you a selection of blank templates for many dScript commands and functions.

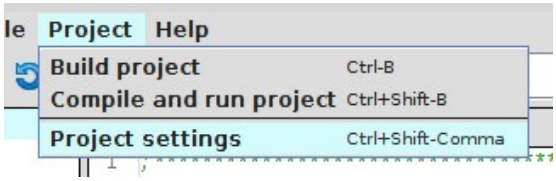


Program Structure

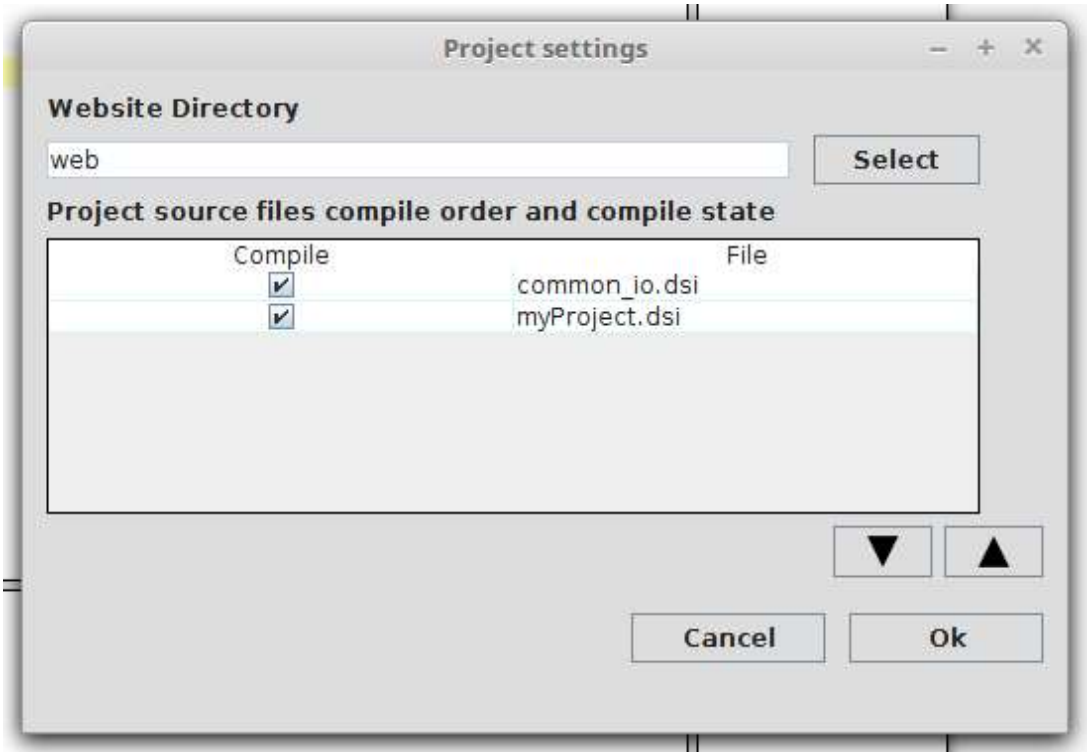
All dScript programs begin with the declarations needed, followed by the program instructions.

The declarations are used to give meaningful names to all the variables and I/O ports that you will be using. The instruction section contains all the code to make your application work, the starting point for your code is the thread "main". It need not be the first thread, but you must include it somewhere in your code.

If you are using the common_io.dsi file that is automatically generated when you create a new project, make sure it is first in the compile order. You set the compile order in the project settings dialog.



The project settings dialog lists the code files in the order they will be compiled. Use the up/down arrow buttons to change the order. You can also select which files will be compiled.



Declarations

This section is used to declare the variables and I/O you are going to use.

An example:

```
string s1[100]
serialport LCD05 1 10 90
```

Instructions

```
thread main(const)
   s1 = "   Hello World"
   s1[0] = 12
   s1[1] = 19
   s1[2] = 4
   LCD05.Write(s1,0,s1.Length)
   threadsleep 300
endthread
```

Comments

Comments may be added to your program in order to document what you are doing.

A comment begins with a semicolon ;

All characters from the ; until the end of the line are ignored by the compiler, they are not downloaded to the board and do not take up program or data space.

Comments are highlighted in green within the editor.

Constant declarations

```
const Minus17  -17
const LogArraySize  24

int32 TemperatureLog[LogArraySize]
int32 StartTemp
```

Declaring numbers that can be used many times in your program as `const` can save time if you need to change it. Use the `const` anywhere in your program where you would otherwise use the number.

```
thread main(const)
  StartTemp = Minus17
  ...
```

Constants can also have an expression as the parameter. All elements of the expression must themselves be constant, which means you cannot use variables.

```
const FILTCTRL  4
const FILTREF   1<<(FILTCTRL-1)
const FILTMOD   1<<FILTCTRL
```

All expressions are evaluated from left to right, there is no operator precedence. If you need to change the order of evaluation then enclose that section in parenthesis. In the example above (FILTCTRL-1) is evaluated first and the result used to shift the 1 left by that many places. Any named constants used in the expression must have already been declared.

Note that all constant expressions are evaluated by the compiler when the program is compiled, not on the board at runtime.

There are a few built in constants intended for use with relays and I/O:

on, active	these evaluate to numerical 1
off, inactive	these evaluate to numerical 0

Variables

The are two variable types, integer variables and string variables. Variable names must start with a letter and may be followed by letters (A-Z, a-z), digits (0-9) or an underscore (_). No other characters are permitted in variable names. All variables are global and may be accessed from any subroutines or threads.

Local variables

Local variables are variables that are declared within a thread or function. They exist only while the function is running and cannot be accessed from outside the function. They are created when the function is called and destroyed on return. Therefore you cannot store data in a local variable and expect it to persist between function calls. On entry to a function local variables are not initialised and contain random data.

If a local variable has the same name as a global variable then it is the local variable that is used and the global variable is not available to that function.

For each call to a function, a new set of local variables is created. Multiple calls to the same function will not interfere with each other, even if they are called at the same time from different threads.

Integer variables

There are three types of integer variables used to store numbers.

```
int32   X2
int16   X3
int8    X4
```

int32 integer variable types are stored as 32 bit signed numbers. The range of values that can be stored is from -2147483648 to 2147483647.

Variables of type int16 hold 16 bit signed numbers. The range of values that can be stored is from -32768 to 32767.

Variables of type int8 hold 8 bit signed numbers. The range of values that can be stored is from -128 to 127.

It is recommended that 8 and 16 bit variables are used only when you absolutely must. This is likely to be when locating variables in eeprom memory which is limited. For general use there is no point in trying to save memory if your application is only going to be using part of it. Internally dScript uses 32 bit numbers for all calculations anyway, so always try to use int32 types.

You can have any number of integer variables up to the limit of the on-board RAM memory. Variables can be named as you choose. Short names like x and y are fine, but there are no built in limits to the length of the name, so be descriptive in your choices. Variable names are case sensitive so:

int32 mycounter

int32 myCounter

are two different variables.

Variable names are not loaded into the module. All variables are compiled to an index code, so long variable names neither consume valuable memory space on the module or slow down program execution.

Variable arrays

A variable array is like a row of variables. To declare an variable array place the size of the array in square brackets immediately after the variable name.

```
int32 TemperatureLog[24]
```

This declares a row of 24 individual variables (or elements) which can be accessed by using an index number. The first element is at position 0 and the last element is at position 23 (not 24).

To read or write an array element, specify the element number in the square brackets.

```
x = TemperatureLog[2]
TemperatureLog[2] = 19
```

The array index is checked at runtime and any attempt to write to an element outside the bounds of the array is ignored and attempts to read an element outside the bounds of the array will return 0.

The size of an array is limited only by the available memory.

Arrays may have multiple dimensions. To declare a two dimensional array place the sizes of the dimensions, separated by a comma, in square brackets immediately after the variable name.

```
int32 myGrid[10,10]
```

This declares a two dimensional array with 10 elements (numbered 0-9) on each side of the grid. 100 elements in total. When reading or writing, both dimensions must be specified.

```
x = myGrid[3,7]
myGrid[3,7] = 123
```

Arrays may have up to 4 dimensions.

```
int32 my3Dgrid[10,10,10]
int32 my4Dgrid[10,10,10,10]
```

Arrays may also be used with int8 and int16 variable types.

```
int8 dataLog[10000]
int16 HourlyTemps[24]
```

String variables

A string variable is used to hold ASCII character strings. The length of the string variable may be up to the lesser of 65535 bytes or the remaining memory. The size of a string variable is declared by placing the size in square brackets immediately after the name.

```
string myStringName[100]
```

declares a string called myStringName which is can hold up to 100 characters.

You can have any number of string variables up to the limit of the on-board RAM memory. Strings can be named as you choose, there are no built in limits to the length of the name so be descriptive. string names are case sensitive so:

```
string  myStringname[100]
string myStringName[100]
```

are two different variables.

String variable names are not loaded into the module. All variables are compiled to an index code, so long variable names neither consume valuable memory space on the module or slow down program execution.

String parameters

Summary;

string.Length	
string.Size	
string.Mid(0,3)	Extracts middle text from string. (start, count)
string.ToLower(0,3)	Same as Mid, also converts result to lower case
string.ToUpper(0,3)	Same as Mid, also converts result to upper case
string.GetNumAscii(Idx)	Gets number from ASCII string, returns int, bumps idx.
string.GetNumBinary(Idx, 4)	gets number from binary string, bumps idx, bytes to get.
string.GetText(Idx)	extracts text from string, from idx to 1st non-alpha char.
string.BooleanEval()	evaluates boolean expression, returns true(1) or false(0)

string.Length
string.Size

These return the length and size of a string as an integer.

```
string myBuffer[100]
int32 Length
int32 Size

myBuffer = "Hello World"
Length = myBuffer.Length
Size = myBuffer.Size
```

Length will have the length of the string myBuffer, in this case it is 11,
Size will have the original declared size of myBuffer, 100 in this case.

string.Mid(0,3)
string.ToLower(0,3)
string.ToUpper(0,3)

Mid(position, count) Returns *count* characters from *position* of a string

```
string myNewBuff[100]
string myOldBuff[100]

myOldBuff = "engineer"
myNewBuff = myOldBuff.Mid(2,3)
```

This will result in myNewBuff containing "gin".
Note that the starting index is zero based, so the 'g' is at position 2, not 3.

Parameter 2 is optional, in that case text from *position* to end of text is returned.
myNewBuff = myOldBuff.Mid(4) will load myNewBuff with "neer"

Both parameters may be omitted.
myNewBuff = myOldBuff.Mid() will copy all text from myOldBuff into myNewBuff.

ToLower and ToUpper work in an identical way to Mid, except the extracted text is converted to lower or upper case.

string.GetNumAscii(Idx)

Gets number from ASCII string and returns an integer. Any spaces at the beginning of the number are skipped. The ASCII number is acquired up to the first non-digit character and the index variable bumped to that point.

```
string myBuffer[100]
int32 idx
int32 rly
idx = 5
myBuffer = "Relay 12 on"
rly = myBuffer.GetNumAscii(idx)
```

This will result in rly containing the number 12 and idx containing 8.

string.GetNumBinary(Idx, 4)

Gets number from binary string and returns an integer. Any spaces at the beginning of the number are skipped. Idx is the position in the string. The 2nd parameter is the number of bytes in the number and may be 1, 2 or 4. Idx is incremented to the position after the last byte acquired.

```
string myBuffer[100]
int32 idx
int32 pos
idx = 0
myBuffer[0] = 12
myBuffer[1] = 34

pos = myBuffer.GetNumBinary(idx, 2)
```

This will result in pos containing 3106 and idx will be 2.
(3106 because 12 is 0x0c and 34 is 0x22, so the 16 bit number is 0xc22 or 3106 decimal).

This instruction is useful for extracting the parameters from a modbus command stream.

string.GetText(Idx)

Gets text from an ASCII string. Any spaces at the beginning of the number are skipped. This is different from string.Mid which extracts a fixed section of text. String.GetText can extract a variable length of text from the Idx position to the first non-alphanumeric character. It can be used for extracting a word of unknown length.

```
string myNewBuff[100]
string myOldBuff[100]
int32 idx

myOldBuff = "Relay 2 on"
idx = 7
myNewBuff = myOldBuff.GetText(idx)
```

This will result in myNewBuff containing "on".

string.GetNumAscii(Idx) and string.GetText(Idx) are useful for extracting commands from an incomming command string.

```
string cmdBuff[100]
string cmd[100]
string action[100]
int32 relayNum
int32 idx

cmdBuff = "relay 2 on"
idx = 0
cmd = cmdBuff.GetText(idx)
relayNum = cmdBuff.GetAscii(idx)
action = cmdBuff.GetText(idx)
```

Will result in cmd containing "relay"
relayNum containing 2
action containing "on"

string.BooleanEval()

This will evaluate a boolean equation in the string and returns true(1) or false(0). We use this in the supplied application to provide some automation to the relays and also to trigger emails.

The types that can be used in boolean equations are:

1. Relays, R1 – R24
2. Digital I/O's, D1 – D8
3. Analog inputs, A1 – A8
4. Counters, C1 – C8
5. Ping timers, P1 – P4
6. Sequencer outputs, K1 – 12
7. Scheduler outputs S1 – S8

The simplest equation is R1. This is true when R1 is active and not true when R1 is in-active. If you enter R1 in the relay 2 automation box it will simply follow whatever R1 does.

The exclamation mark ! is used as a "not". So !R1 is true when relay 1 is in-active.
Enter !R1 in the relay 2 automation box it will follow the opposite of R1. Relay 2 will be active when relay 1 is inactive.

The same applies to the digital I/O's. Enter D2 in the relay 2 automation box and the relay will follow the input.

Analog inputs are compared with a value to obtain a true/false boolean result. In this example we have set I/O8 to be an analogue input with a 5v reference. Then we can enter A8<1000 in the relay2 automation box. This will turn on relay 2 when the input A8 falls below 1000. If A8 is connected to a temperature sensor and R2 controls a heater – well, you get the idea. Analog comparisons use the "less than" < and "greater than" > symbols only. There is no equal or not equal. Checking for equality on a potentially jittery analogue input is not really useful.

As well as "not" !, you can use "and" & and "or" | in your equations.
Enter D2&D3 and the result is true only when both D2 and D3 are active.
Enter D2|D3 and the result is true when either D2 or D3 is active.

What happens here:
D2|D3&D4
The answer is that boolean expressions are evaluated left to right. So D2 is ORed with D3 and the result ANDed with D4. You can change the order of precedence by using parenthesis ().
D2|(D3&D4)
will now and D3 with D4 and the result is Ored with D2.

To demonstrate a real world example, take the analog example above where we compared A8 with 1000 to operate R2. Whilst this would work its a not a good solution as the relay would jitter badly when A8 was jittering between 999 and 1000. What we need is some hysteresis. To do that we will use R2 in its own equation.
(A8<1000&!R2)|(A8<1234&R2)
Assume R2 in inactive (off). The 2nd half of the equation (A8<1234&R2) will have no effect.
So when A8 falls below 1000 the relay comes on. Now the 2nd half of the equation is true, and will stay true until A8 climbs above 1234.
So the relay becomes active when A8 is below 1000 and inactive above 1234.
We have hysteresis!

string.CRC(length)
string.CRC(length, polynomial)

This will return the CRC of the string. Length is the number of bytes, starting at zero, to perform the CRC calculation on.
Optionally the polynomial may be specified. The default polynomial if none is specified is 0xA001.

The default polynomial is x16+x15+x2+1 (0x8005) reflected to 0xA001, also known as CRC16-ANSII and is the one used for MODBUS.
A different polynomial can be used by specifying its reflection in the second parameter.

String1.Compare(String2)

Compares String1 with String2, returning -1, 0 or 1.

Returns zero if both strings are equal.
Returns 1 if String1 is longer than String 2
Returns 1 if String1 is greater than String2
Returns -1 if String1 is shorter than String 2
Returns -1 if String1 is less than String2

Usage:

```
int32 result
string t1[20]
string t2[20]

    t1 = "test212"
    t2 = "test211"

    result = t1.Compare(t2)
```

Will set result to 1

String byte arrays

Strings can be used as byte variable arrays. Array elements are unsigned bytes and can hold positive values between 0 and 255. Strings used as byte arrays cannot store negative values.

```
string myByteArray[10]
```

The individual elements of this string may be accessed by

```
myByteArray[0] = 5
```

which will load the number 5 into the first element of the array. The elements of the 10 byte array are numbered 0 to 9. Note that the size of the index is checked at run time, not compile time.

```
string myByteArray[10]
int32 idx
int32 x
myByteArray[idx] = 5
x = myByteArray[idx]
```

The runtime module will check the value in idx is within the range for the string. Outside of this limit the instruction will do nothing in the first case and load x with zero in the second.

Non-volatile EEPROM variables

dScript modules have a 512 byte EEPROM for storing a small number of variables when power is switched off.

Both string and integer variables can be located in this non-volatile memory. To declare a non-volatile variable use eeint32, eeint16 or eeint8. For example:

```
eeint32 HourCounter
```

To declare a 4 byte array for emulating latching relays use eeint8

```
eeint8 RelayStore[4]
```

Only variables and strings can be stored in non-volatile memory, I/O ports and other objects cannot.

When you read or write to a non-volatile variable, you are really reading and writing to a cache memory. The underlying EEPROM is only written when something actually changes. This means you can update your variables as often as you like with the same value. For instance to update "RelayStore" with the status of Relay1 you could put the following in a loop.

```
if(Rly1) then RelayStore[0] = 1 else RelayStore[0] = 0 endif
```

The loop may be executed as fast as you wish, only a change in the state of RelayStore[0] will get written to the EEPROM. As EEPROMs have a limited write endurance, this prevents unnecessary wear.

A simple and eloquent method to update RelayStore is to give it its own thread.

```
eeint8 RelayStore[2]

thread RelayUpdate(50)
        if(Rly1) then RelayStore[0] = 1 else RelayStore[0] = 0 endif
        if(Rly2) then RelayStore[1] = 1 else RelayStore[1] = 0 endif
endthread

thread main(const)
        if(RelayStore[0]) then Rly1 = 1 else Rly1 = 0 endif
        if(RelayStore[1]) then Rly2 = 1 else Rly2 = 0 endif
        threadstart RelayUpdate

        do
            ... Do things ...
        loop
endthread
```

In the start up section of “main” there are a couple of lines of code to initialise the relays with their stored settings. The RelayUpdate thread is then started which will keep the relays updated every 50mS.

Take care when making variables non-volatile. EEPROMs have a limited write endurance of about 1000000 (1 million) and if you change the values too frequently you can quickly reach

the end of their life. dScript can write a new EEPROM value every 320mS. If you do so you will reach end of life for the EEPROM in less than 4 days!

Try to limit updates. An hour counter which is updated every 10 minutes will last 19 years. The EEPROM used has a minimum write endurance of 1000000, it could be much more.

Non-volatile Flash variables

All modules have 16k (16384) bytes of flash memory reserved for variables. Flash memory has less endurance than EEPROM at just 100,000 erase/write cycles. This memory should therefore be used for configuration data that does not change often.

Flash memory variables are declared in a similar war to EPROM variables, in this case by putting fl at the beginning of the variable type. So;

```
int32       normal int32 variable located in RAM
eeint32     int32 variable located in EEPROM
flint32     int32 variable located in Flash memory
```

All flash variables are read at power-up into cache RAM. Flash variables may be read as often as you wish as the data is coming from the cache at full speed.
When you write to a flash variable, you will be writing to the cached copy. This will also start a 5 second timer running, or if the timer is already running it will re-start it to 5 seconds. This gives you the chance to update all required variables. 5 seconds after the last write, the actual write to flash will automatically take place, updating the entire block in one go. Its done that way because the flash chip can only erase an entire 8k block, not individual bytes or variables. We must therefore erase and re-write the full 8k block.

There is a system variable that you can read to check when the flash write is done.
`system.FlashPending`
Zero means the flash write is done.
Non-zero means the time is counting down to the pending write.

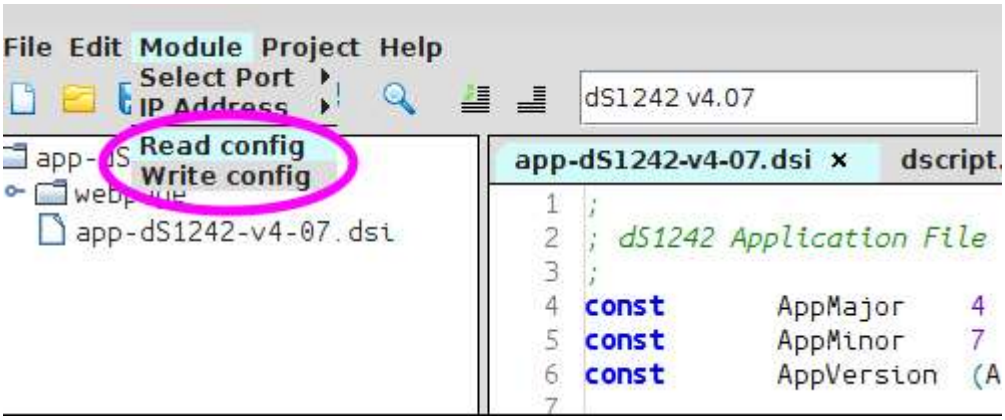
Make sure you do not switch off the module before the flash gets written or you will lose all your changes.

Saving and Restoring Flash Variables to your PC

This feature is included from version 4.07.

Flash variables are stored in a 16k block of RAM. This RAM block is automatically loaded from the flash memory on power-up. When you write to a flash variable, you are writing to the RAM. This is automatically transferred to the Flash 5 seconds after the last write. The 5 second delay is to give you time to update many variables without continuously writing to the flash, which has a limited write cycle life.

The Read config and Write config commands are in the Module menu. Read config will read the full 16k block and save it to a file on your PC. Write config will take the selected config file and load it to the module.



Flash variables are allocated to the 16k block by the compiler in the order they are declared in the dScript program. So if you add flash variables or change their order, then the saved data block will no-longer correctly overlay the variables, and restoring it will result in corrupted flash variables. Its not the specific variables that are being save/restored, but the memory block in which they reside.

The data block is not encrypted, so you may want to restrict access to these files.

System Variables

System variables are special purpose variables.

system.DespatchCounter

This is a counter that increments each time a dScript instruction is executed. It does not matter which thread is executing, all instructions are counted. The DespatchCounter cannot be written to. When read, its current value is returned and then the DespatchCounter is reset to zero. See the **Performance** chapter for details on the use of this variable.

system.ModuleID
This read only variable contains a number identifying the module.

Module	ID
dS3484	30
dS1242	31
dS2824	34
dS2832	42
dS378	35
TCP184	36

system.VerMajor
System software major version, read only. For example 2

system.VerMinor
System software minor version, read only. For example 17, so version would be 2.17

system.FlashPending
The number of mS to go before a flash write takes place. Its set to 5 seconds (5000mS) when a flash variable is written.

system.Random
Returns a 32-bit random number.

system.MyIP
Returns an int32 representing the ip address. The MyIP example shows how to get and convert this int32 into a dot-decimal string.

0x5200A8C0 is 192.168.0.82
Note the high byte 0x52 is the last decimal (82) in the string.

system.Booting

Returns non-zero after power-up until the Ethernet stack is ready for use. Returns 0 when ready, normally after 8-10 seconds.

system.Restart

Not actually a variable at all. When read this will trigger a module reset by issuing a reset instruction to the CPU.

X = system.Restart

system.ThreadHandle

Returns a handle for the current thread.

system.SleepTimer(ThreadHandle)

Returns the current value of a threads sleeptimer.

Thread1 should use system.ThreadHandle to get its thread handle and store this in a global variable (int16 or int32).

Thread2 can then use system.SleepTimer (The Handle var) to get Thread1's sleeptimer.

Used so a thread can determine how long it will be before another thread wakes up.

Special Variables

These are documented here for completeness, they are not normally of any use to your dScript programs.

setcounters

setpingtimers

setkoutputs

Their purpose is to provide the actual addresses of variables that are used by the boolean expression evaluator. For example "setcounters" is followed by a list of the 8 counters available in the app. It links the C1, C2 etc, used in expressions to the actual variable that's holding the data.

Likewise, "setpingtimers" is followed by the 4 pingtimer variables linked to P1-P4, and "setkoutputs" is followed by a variable that holds the sequencers 12 single bit outputs linked to K1-K12.

In each case the parameter variables must be global int32's.

setpingtimers and setkoutputs were added in v4.07 and the old "counters" renamed to "setcounters".

Note that if you are using a program based on a pre-4.07 version, you will need to rename "counters" to "setcounters" for it to compile on version 4.07 onwards.

Functions

A function is just a named section of code that you can call from other parts of your program. It is declared like this:

function [Return type] Function Name([Parameter1, … , Parameter n])

….

endfunction

All functions start with "function" and end with "endfunction".

The return type and the parameters are optional. So in its simplest form a function call is just:

GoDoSomething()

A function with a return type can return a value. The following will read an analogue temperature and convert it to degrees C.

```
function int32 GetTemperature()
int32 BrdTemp
   BrdTemp = ((TS1*3223)-500000)/1000
   return BrdTemp
endfunction
```

The caller can get the temperature with:

```
int32 AirTemp
AirTemp = GetTemperature()
```

Parameters are the way to pass data to a function. For example, to add two numbers:

```
int32 result
result = AddNumbers(2, 3)
```

And the function would be declared as:

```
function int32 AddNumbers(int32 a, int32 b)
   return a+b
endfunction
```

Parameter passing

Parameters such as numbers or integers are passed by value. This means that it is the value only that is passed to the function which will then store the value ready for use. If you change the value, only the local copy will change. The original variable will be unchanged.

String and array parameters are passed by reference. It is not the data, but a reference to the original variable that is passed. The function does not have a copy, instead accessing the original. Any changes will be seen by the caller.

Network Parameters

There are a number of network specific parameters that need to be setup in your program. The following three parameters are required in all cases where you are using Ethernet.

tcpip.hostname "myModulesHostname"

This is the name of the module as it will be seen on your network

tcpip.ip "192.168.0.136"

The module has a single IP address that is used for both web pages and direct tcp/ip communications.

tcpip.mask "255.255.255.0"
This is normally set as above. If in doubt ask you network administrator to assign you a permanent IP address and mask for the module.

tcpip.port 17494
If you want to use a tcp/ip server then you will also require a port number. Choose an unused high number in the range 2000 up to 65535 for this.

tcpip.dns1 "192.168.0.1"
tcpip.dns2 "8.8.8.8"
tcpip.gateway "192.168.0.1"

If you are going to be using a tcp/ip client to control other modules on the internet then you will need to set up a gateway address. This is normally the address of your internet router.

The DNS server addresses are required if you are going to be using names instead of IP addresses. Again, this will likely be your routers address but you can also use Google's DNS server as shown in dns2 above.

tcpip.dhcp off (on)

Disable or enable DHCP. Enabling the DHCP server is generally a bad idea as the DHCP server can assign a different IP address each time, and you really need it to be fixed. If in doubt ask your network administrator to assign you a permanent IP address and mask for the module.

Using variables for the network parameters.

```
eestring    System_IP[16]
tcpip.ip    System_IP
```

These MUST be located in eeprom by using `eestring` or flash by using `flstring` rather than `string`.

This is because the parameters are loaded before your dScript program starts running and your program would never get the chance to initialize them. Variables in eeprom/flash are available immediately.

Pinging a remote host.

DNSResolve(RemoteHost, PingIP)

RemoteHost is a global string containing the host to resolve. This may be a name such as "example.com" or an IP address in dot notation such as "192.168.2.34".

PingIP is a global int32 that will be loaded with the IP address for use with the ping command.

DNSResolve should be used at startup of your program to resolve any required IP addresses, not called continuously during operation.

tcpip.Ping(PingIP, Timer)
Up to 4 IP addresses may be pinged.
PingIP is a global int32 containing the IP address to ping, generated by DNSResolve.
Timer is a global int32 that will contain the response time in mS or zero if no response was received.

The eeprom variables must be declared before the tcpip parameter that uses them.

```
eestring    System_HostName[21]
eestring    System_IP[16]
eestring    System_SubNet[16]
eestring    System_Gateway[16]
eestring    System_DNS1[16]
eestring    System_DNS2[16]
eeint32     System_CmdPort

tcpip.ip         System_IP
tcpip.mask       System_SubNet
tcpip.hostname  System_HostName
tcpip.port       System_CmdPort
tcpip.dns1       System_DNS1
tcpip.dns2       System_DNS2
tcpip.gateway    System_Gateway
```

All eeprom variables are assigned addresses in eeprom in the order they are declared in the program, so to be sure the parameters do not move in memory don't declare any eeprom variables before the tcpip parameters. Make sure these ones are first so any following eeprom variables will not affect their position.

The use of variables for the tcpip parameters allows your program to update the network configuration. This will then take effect next time the module is reset.

To initialize your network, place the following at the start of your dScript program.

```
System_HostName = "SetupTest"
System_IP = "192.168.0.123"
System_SubNet = "255.255.255.0"
System_Gateway = "192.168.0.1"
System_DNS1 = "192.168.0.1"
System_DNS2 = "8.8.8.8"
System_CmdPort = 17494
```

That will initialize the variables for use after the next re-boot. After that you can comment out or delete the initialization.

Analogue & Digital I/O channels

These consist of the relays, digital I/O's and analogue inputs as well as on-board I/O such as the LED's, temperature and voltage analogue inputs, and the virtual I/O's which can be used as flags.

Digital I/O

All Digital I/O such as relays, led's, general purpose I/O (on the dS1242 & dS3484) or virtual I/O are declared with the digitalport declaration. See module documentation for specific details of the port numbers.

Examples:

```
digitalport Rly1          1
digitalport Rly2          2

digitalport LedBlue      33
digitalport LedGreen     34
digitalport LedRed       35

digitalport Flag1       100
digitalport Flag2       101
```

To output to the port just write:

```
Rly1 = on
LedBlue = on
Rly2 = off
Flag1 = on
```

There are 64 virtual I/O ports in the range 100 to 163. These are not associated with any real I/O port, they exist only in the modules memory. Flag1 and Flag2 above are examples of virtual I/O. They are useful to hold the state of remote I/O, of just as flags to control program flow.

Analogue I/O

All Analogue inputs on the dS1242 & dS3484 are declared with the analogport declaration (All other modules use flexport – see below). See Module documentation for specific details of the port numbers.

Examples:

```
analogport  TS2      2   ; ext temp sensor
analogport  TS1      100 ; on-board temp sensor
analogport  PSU      101 ; DC power voltage
```

Analogue ports do not store any values, so each time you refer to the port a new analogue to digital conversion is performed.

```
int32 Volts
Volts = PSU
```

Will convert analogue channel PSU and store the result in the variable Volts.

Flexible I/O

The I/O on the dS378, dS2824 and TCP184 modules can be either digital or analogue input. They are defined using the `flexport` command rather than `digitalport` or `analogport`. Just the I/O lines on these modules use `flexport`. All other ports such as the on-board LEDs use `digitalport` and the on-board temperature/voltage sensors use `analogport`.

The dS1242 and dS3484 modules do not have flexible I/O and cannot use the `flexport` command. The modules should only use `digitalport` or `analogport` commands to define the I/O's.

The `flexport` command is similar to the `digitalport` and `analogport` commands, except that one additional parameter is required to define the ports function. This may be one of the 5 pre-defined constants shown below.

Command	Name	Port	Function	
flexport	IO1	1	digitalpullup	
flexport	IO2	2	digitalnopullup	
flexport	IO3	3	analogref3	; dS378 only
flexport	IO4	4	analogref4	; dS2824/dS2832
flexport	IO5	5	analogref5	; dS2824/dS2832

The dS378 can only use analog input type `analogref3` because is has a 10-bit, 0-3.3v input range. The dS2824 has a 12-bit input powered from a 5v supply. It can use the supply as a reference (`analogref5`) giving a 0-5v input range or an internal 4.096v reference (`analogref4`) for a 0-4.096v range.

The `flexport` command fixes the I/O type at the start of the program and they are fixed for the duration of the programs life. They cannot be changed “on-the-fly”.

Operators

Arithmetic & bitwise operators

Symbol	Operation	Example
+	Addition	X = A + B
-	Subtraction	X = A - B
*	Multiplication	X = A * B
/	Division	X = A / B
//	Modulus (remainder of a division)	X = A // B
<<	Shift left (0 shifted into lowest bit)	X = A << 3
>>	Shift right (sign bit extended into highest bit)	X = A >> 3
&	Bitwise AND	X = A & B
|	Bitwise OR	X = A | B
^	Bitwise XOR	X = A ^ B

Assignment operators

Symbol	Operation	Example
=	Assignment only	X = A
+=	Addition	X += A
-=	Subtraction	X -= A
*=	Multiplication	X *= A
/=	Division	X /= A
//=	Modulus (remainder of a division)	X //= A
<<=	Shift left (0 shifted into lowest bit)	X <<= 3
>>=	Shift right (sign bit extended into highest bit)	X >>= 3
&=	Bitwise AND	X &= A
|=	Bitwise OR	X |= A
^=	Bitwise XOR	X ^= A

Logical operators

Logical operators return true or false.

Symbol	Operation	Example
>	Greater than	A > B
>=	Greater or equal	A >= B
<	Less than	A < B
<=	Less or equal	A <= B
==	Equal	A == B
!=	Not equal	A != B

An example of using logical operators is the "if" command.

```
if A > 5 processA()
```

Expressions

Expressions are a sequence of variables and operators. They are used to assign values to either numeric or string variables.

Numeric expressions

X = A + B

This will add the variables A and B together and assign the result to variable X. An expression may have any number of operations.

X = A + B * C

Expressions are always evaluated from left to right. No precedence is given to multiply and divide as some compilers do. If A=2, B=3 and C=4 then the expression above will yield 20.

2 + 3 = 5

5 * 4 = 20

Parenthesis may be used to change the order the expression is evaluated.

X = A + (B * C)

will give the result 14

This following expression will use the raw value from the ADC (Analogue to Digital Converter) and scale this to read temperature in degrees C.

Temperature = ((ADC2*3223)-500000)/10000

Although the parenthesis is not actually required here as we want to evaluate from left to right, it is included for clarity. The following works just as well:

Temperature = ADC2*3223-500000/10000

X = X + 1
X += 1

The above expressions are functionally identical

String expressions

The ~ operator is used to join two strings together to make one string.

S1 = "This is "
S2 = "compiler!"
S3 = S1 ~ "the dScript " ~ S2

results in S3 containing "This is the dScript compiler"

Numeric variables in string expressions

Numeric variables can be included in string expressions.

X = 24
Y = 31
S1 = "Variable X contains " ~ X

results in S1 containing "Variable X contains 24"

S1 = "Look at this -> " ~ X + Y

results in S1 containing "Look at this -> 55"
The + in string expressions is addition, NOT concatenation.

Decimal, hexadecimal and binary formatting

By default variables are placed in strings in decimal format as above. They can also be inserted in hexadecimal or binary formats. To do this place the control character in curly braces immediately before the variable.

S1 = "X in hex format is " ~ {H} X

results in "X in hex format is 18"

or

S1 = "X in binary format is " ~ {B} X

results in "X in binary format is 11000"

and you can use decimal

S1 = "X in decimal format is " ~ {D} X

results in "X in decimal format is 24"

Padding numeric output

Sometimes when displaying numeric output, you will want to use a constant width, regardless of the number. To achieve this just add a number following the format control character.

{D3} will display the number as three digits.

```
  4
 34
234
6234
```

That worked well except for the last number. If a number is larger than the number of places allocated it will use as many as needed to display the result. Make sure you plan for the largest expected number. Using {D4} here will work.

```
   4
  34
 234
6234
```

Numbers can also be preceded with 0 instead of a space. Put a 0 between the control character and the pad count, like this {D04}

```
0004
0034
0234
6234
```

X = 42

S1 = "Variable X contains " ~ {B08} X
results in "Variable X contains 00101010"

S1 = "Variable X contains " ~ {B04} X
results in "Variable X contains 101010"

S1 = "Variable X contains " ~ {D04} X
results in "Variable X contains 0042"

S1 = "Variable X contains " ~ {H04} X
results in "Variable X contains 002A"

Formatting negative numbers

When the number is negative a '–' sign is placed in front of the number. If you are specifying a width then this '–' sign counts as one of your characters.

X = -59
S1 = "Variable X is " ~ {D} X
results in "Variable X is -59

S1 = "Variable X is " ~ {D04} X
results in "Variable X is -059

The '-' sign is never displayed when formatting in binary or hexadecimal formats. This is normal, because when using binary or hex you want to see what is contained in that variable without regard to what the value represents. As the variables are 32 bits wide the full 32 bit value is used.

S1 = "Variable X is " ~ {H02} X
results in "Variable X is FFFFFFC5

S1 = "Variable X is " ~ {B08} X
results in "Variable X is 11111111111111111111111111000101

If you are certain your value is contained in a smaller number of bits then the upper bits can be masked out. For example you may want to display the value that has come in from a serial port. You know this is only 8 bits so the upper bits can be masked out with:

X = X & 0x000000FF
which is the same as:
X = X & 0xFF

now,

S1 = "Variable X is " ~ {H02} X
results in "Variable X is C5

and,

S1 = "Variable X is " ~ {B} X
results in "Variable X is 11000101

Inserting control codes into a string

The {c} control is used to insert a specific byte into your string. This is useful for inserting things like CR/LF at the end of a line.

S1 = "Ok" ~ {c}13 ~ {c}10

dScript commands

return

return is used at the end of a function. It will return any data following the return statement. It can be used on its own, or may be followed by an expression.

return variableName
will return the data contained in the variable “variableName” back to the caller.

if - single statement execution

The "if" command is used to make a decision on what to do next. The simplest form of the if command to conditionally execute a single instruction.
if <condition> statement.
if A > B DoSomething()
however any single instruction may be executed.

This simple form of the if command does not require endif and cannot use else to provide an alternative operation.

if - multiple statement execution

the syntax of the if command is:

if <condition> then

  statements

elseif <condition>

  statements

else

   statements

endif

The compiler used the "then" keyword that follows the <condition> to determine if this is the single or multiple format of the "if" statement.

The "elseif" command can follow "then" or "elseif" commands and must terminate with "elseif", "else" or "endif".

The "else" command can follow "then" or "elseif and must finish with the "endif" command.

The final block of statements must be terminated with the "endif" command.

An example:

```
LedRed = off
LedBlue = off
LedGreen = off
x = 2
if x == 1 then
   LedBlue = on
   a = 21
elseif x == 2
   LedRed = on
   a = 45
else
   LedGreen = on
   a = 193
endif
```

The “if” condition may have multiple comparisons separated with “and” or “or”.

```
if a=3 and b=5 or c=7 then .. .  .
```

The expression is evaluated left to right.

for, next

The for/next commands are used to execute a set of instruction a specified number of times. The syntax is:

```
for <variable> = start to end
   statements
next
```

An example:

```
for X = 1 to 10                ; X counts up each loop
   A = A + 1
next
```

If end is greater than start, the count will increment by one each loop. If the start is greater than end the count will decrement by one each loop;

```
for X = 10 to 1                ; X counts down each loop
   A = A + 1
next
```

for, next loops may be nested like this:

```
for X = 1 to 10                ; outer loop
   for Y = 1 to 10             ; inner loop
      A = A + 1                ; A gets incremented 100 times (10 loops of 10)
      ... more statements ...
   next
next
```

do loop

The do loop will execute forever:

```
do
   statements
loop
```

Optionally the do loop can have conditional tests at the beginning or end of the loop (but not both).

The variations for the do loop command are:
do [statements] loop
do while <condition> [statements] loop
do until <condition> [statements] loop
do [statements] loop while <condition>
do [statements] loop until <condition>

An example:

```
A = 0
do
   A = A + 1
loop while A<100
```

The difference between putting the conditional at the beginning or end of the loop is if the conditional test (while or until) is placed after the loop command the loop will always be executed at least once, even if the test is false. Placing the test before the do means the loop does not get executed if the condition is false.

```
A = 5
do while A<3          ; this test is false
   A = A + 1          ; so this command never gets executed
 loop
```

```
do
   A = A + 1          ; this command will get executed once
loop while A<3        ; even though this test is false
```

select case

The syntax for the select command is:

select <expression>
  case <expression>
    [statements]
  case <expression1> to <expression2>
    [statements]
  case is <comparison operator> <expression>
    [statements]
  else
    [statements]
endselect

When no case expressions match, the else statements are executed.
When more than one case expression matches, only the first matching block will be executed.
Control then passes to the instruction following the endselect command.

An example:

```
a = 4
select a
   case 4
      LedRed = on
   case is > 3
      LedGreen = on
   case 1 to 7
        LedBlue = on
endselect
```

Here, all three case expressions match, but only the Red LED will light up.

```
a = 4
select a
   case 5
      LedRed = on
   case 1 to 3
        LedGreen = on
   else
      LedBlue = on
endselect
```

Here, none of the case expressions match, so "else" bock will execute (the Blue LED lights up).

Select/case may also be used with strings

select <string variable>
  case <string constant>
    [statements]
  case <string constant>
    [statements]
  else
    [statements]
endselect

For example

```
string cmd[100]

cmd = "green"

select cmd
   case "red"
      LedRed = on
   case "green"
      LedGreen = on
   case "blue"
        LedBlue = on
endselect
```

This will light up the green LED.

Serial ports

dScript Modules may implement one or more serial ports, these are declared at the start of your program. Because it takes time to send data out over a serial port we provide each port with both receive and transmit FIFO (first in first out) buffers. Once declared a serial port will receive and transmit in the background. Received data will be placed in the Rx Fifo ready for when your program wants to fetch it. Transmit data can be sent to the Tx Fifo and will then be sent out in the background. You decide how big these Fifo buffers are, up to the limit of available RAM. The default format for a newly declared serial port is 9600 baud 2 stop bits.

serialport Name PortNumber RxFifoSize TxFifoSize
serialport LCD05 1 10 90

Name
The first parameter is the port name. This is the name you will use to refer to this serial port in the rest of your program. In the above example we will connect an LCD05 display module, so naming the port LCD05 provides a descriptive name to make the program more readable.

PortNumber
Some modules may have three serial ports numbered 1, 2 and 3. This selects which physical port to use.

RxFifoSize
This is the size of the receive or Rx FIFO. Here the LCD05 will only ever send a few bytes back in response to a command, so 10 bytes is plenty.

TxFifoSize
This is the size of the transmit or Tx FIFO. In the case of the LCD05 which is a 20x4 character display, we want to send a complete screen of data plus a few control codes as well, so the size is set to 90 bytes.

The above example names serial port 1 as "LCD05" with an Rx buffer size of 10 bytes and a transmit buffer size of 90 bytes.

Writing data to a serial port

The "Write" command is used to send serial data. The syntax is:
mySerialPortName.Write(<string name>, <start position>, <number of bytes to send>)

Data to be written to the serial port should be placed into a string - think of it as a byte array of the data to send.

```
string s1[100]
serialport LCD05 1 10 90
   s1 = "Hello World"
   LCD05.Write(s1,0,11)
```

Here we are sending 11 bytes from string s1 beginning at the first location (string indexes are zero based) to serial port 1 which we have named as LCD05.

A better way is to place the byte count into the the write command is to used the string length parameter.

```
 LCD05.Write(s1,0, s1.Length)
```

The length of a string is the number of bytes up to and **excluding** an 0x00 value. This 0x00 is automatically placed at the end when you assign text to a string.

```
S1 = "Hello"
```

will require 6 bytes, placing the 5 characters of the text in positions 0-4 of the string. Position 5 will be 0x00.
S1.Length will return 5.

Sometimes you will want to include some non ASCII commands. In the case of the LCD05 you might want to send initialisation or cursor control commands. Here's our "hello world" example again, this time we have put three spaces in front of the text.

```
string s1[100]
serialport LCD05 1 10 90
   s1 = "   Hello World"     ; three spaces before the text
   s1[0] = 12               ; LCD05 command to clear screen & home the cursor
   s1[1] = 19               ; LCD05 command to turn backlight on
   s1[2] = 4                ; LCD05 command to hide the cursor
   LCD05.Write(s1,0,s1.Length)
```

After loading the "Hello World" text into s1, we then replace the three spaces with our LCD05 commands. The LCD05 will clear the screen and set the cursor position to the top left. It will turn the backlight on and then hide the cursor. The "Hello World" text will then be printed starting at the cursor position (top left of the display).

Reading data from a serial port

The "Read" command is used to read serial data, the syntax is:
mySerialPortName.Read(<string name>, <start position>, <number of bytes to read>)

Data read from the serial port is placed in a string - think of it as a byte array holding your incoming data. The following program will read and display the firmware version number of the LCD05.

```
   S1[0] = 15                ; LCD05 command to read firmware version
   LCD05.Write(s1,0,1)       ; send it
   threadsleep 5             ; wait for data to arrive
   LCD05.Read(s1,0,1)        ; get it.
   C = s1[0]
   s1 = "  LCD05 V" ~ C      ; format for display - note the two leading spaces
   s1[0] = 2                 ; for the set cursor command
   s1[1] = 41
   LCD05.Write(s1,0,s1.Length) ; display the version number
```

The number of data bytes available in the receive buffer can be read with **serialport.BytesToRead**. Instead of sleeping for 5mS we could have written:

```
do
   C = LCD05.BytesToRead     ; get number of bytes in receive buffer
while C < 1
```

There are two serial port transmit buffer parameters:

serialport.BytesToWrite will return the number of data bytes in the transmit buffer that are still to be transmitted.

serialport.BytesAvailable will return the number of free bytes in the transmit buffer that you can write to.

Baud rate

The default serial parameters are 9600, no parity, 2 stop bits. The baud rate can be changed with serialport.BaudRate = <new baud rate>

```
serialport RS485 2 100 200
RS485.BaudRate = baud34800  ; change baud rate to 34800

baud38400 is a predefined baud rate. The full list of predefined common baud
rates are:
baud1200     (16666) 1200 baud
baud2400     (8332)
baud4800     (4166)
baud9600     (2082)
baud31250    (639)
baud34800    (520)
baud57600    (346)
baud115200   (173)
baud250k     (79)    250000 baud
baud500k     (39)
baud1M       (19)    1000000 baud
baud2M       (9)
baud4M       (4)
baud5M       (3)
baud10M      (1)
```

The numbers in brackets are the actual divisor values used to control the baud rate. If you need a special baud rate within the above range you can calculate this using the formula:

divisor = (20000000/<baud rate>)-1

solving for 9600 baud:
divisor = (20000000/9600)-1 = 2082.333 (2082 with an error of 0.016%)

If you needed 62500 baud:
divisor = (20000000/62500)-1 = 319 (with an error of 0%)
RS485.BaudRate = 319 ; sets 62500 baud

The baud rate can also be read back.
Baud = RS485.BaudRate
The value read back is the divisor value (the numbers in brackets in the above definitions).

Stop bits

The default is 2 stop bits.

```
serialport RS485 2 100 200
RS485.StopBits = 1  ; set 1 stop bit
RS485.StopBits = 2  ; set 2 stop bits
```

Only 1 or 2 is allowed, any other value is ignored and StopBits will remain unchanged.

Break

Some systems require a "break" to be transmitted as a start of frame marker. A break is defined as the Tx line low for longer than one character frame. The break time used by dScript is 18 bit times. For Example a baud rate of 19200 has a bit time of 52uS. We therefore send a break of 18*52uS = 936uS.

A break is not queued in the transmit FIFO. It is initiated immediately only if the transmit FIFO is empty. If there is a transmission in progress when the break is requested it will be ignored. You should check all previous transmissions are complete by checking BytesToWrite is zero. To start the break use serialport.SetBreak(). While the break is in progress reading RS485.Break will return 1 (on). It will change to 0 (off) when complete.

```
do loop while RS485.BytesToWrite > 0  ; wait for last Tx to go
RS485.SetBreak()                      ; send the break
do loop while RS485.Break == on
< send data bytes >
```

Parity

Parity can be set to none, odd or even by setting 0, 1, or 2
Default is no parity.

```
RS485.Parity = 0        ; no parity
RS485.Parity = 1        ; even parity
RS485.Parity = 2        ; odd parity
```

TCP/IP ports

Both client and server tcp/ip ports are available. The difference between them is that a client port initiates communication while a server port listens for an incoming connection. Both are capable of transmitting and receiving. The internal sdrt6

T/*/IP buffer size is 512 bytes.

TCP/IP server

To operate a tcp/ip server the module will require an IP address and a port number to listen on. The default IP address is 192.168.0.123 and the default mask is 255.255.255.0. If you have a DHCP server on your network (most probably your router) then the module will use the IP address issued by that server. In that case the default IP is not used. However it is more likely you will want to use a fixed IP of your own choosing. Defining your own IP address overrides both the default and DHCP addresses. The following code will assign the IP address and mask:

```
tcpip.ip    "192.168.0.136"
tcpip.mask  "255.255.255.0"
```

The port number is assigned the same way:

```
tcpip.port  17494
```

Note that as this is a server, you do not need gateway or DNS addresses. The server never needs to go out on the network, it just responds to incoming connection requests.

To receive and process incoming messages you will need to set up a thread like this:

```
thread myThreadName(tcpip)
```

for example:

```
thread TcpipCmds(tcpip)
```

The tcpip type informs the compiler that this thread will be triggered when a tcp/ip message comes in. To receive the incoming message you need to define one integer and one string variable.

```
int32   tcpLength
string  tcpBuf[1024]
```

The thread will be:

```
thread TcpipCmds(tcpip)
        tcpip.Read(tcpBuf, tcpLength)
        . . . . .
        <Process Message>
        . . . . .
endthread
```

The read command will read the incoming message into the supplied buffer and report the length in the supplied `int32`, this is to notify you how many bytes have arrived. You do not need to check for an incoming message, the thread will start automatically as soon as it arrives. After you have finished processing the message just suspend the thread, it will start

again from the beginning when a new message arrives.

Here is a simple tcp/ip server example which just increments the first byte of the message and sends it back to the client.

```
tcpip.ip    "192.168.0.136"
tcpip.mask  "255.255.255.0"
tcpip.port  17494

int32   tcpLength
string  tcpBuf[1024]

thread TcpipExample(tcpip)
   tcpip.Read(tcpBuf, tcpLength)
   tcpbuf[0] = tcpbuf[0] + 1
   tcpip.Write(tcpBuf, tcpLength)
endthread

thread main(const)
   threadstart TcpipExample
endthread
```

The server uses tcpip.Write() to respond to the incoming connection. The two parameters are the string containing the data to send and the number of bytes to send. ie. tcpip.Write(myResponse, 5).

Note. tcpip.Read and tcpip.Write are used in the server code only, they are not used by tcp/ip client code.

Further examples.

Have a look at the tcpipServer1 example in the example folder. This will demonstrate how to set up a tcp/ip server program. For a more comprehensive example look at the tcpipServer2 example. When run on a dS3484 , this emulates most of the functions of our ETH484 module.

TCP/IP client

The tcp/ip client can initiate a communication to a tcp/ip server, we will use an ETH002 as an example. To do this you need to define a client port like this:

```
clientport  ETH002 "192.168.0.96" 17496 5000
```

The first parameter is the name of the client port that you will use in your program to refer to this port. In this case an ETH002, so that's what we will name it. Next is its IP or DNS address followed by the port number. The last parameter is the timeout in mS that we will wait for a response. You may define any number of client ports but there is only one socket so only one port can be connected at a time.

If the ETH002 is on the same network, that is all that is needed. However if the ETH002 were at another location on the internet, we also need to set up a gateway address.

```
tcpip.gateway     "192.168.0.1"
```

In this case the gateway is the address of our router since that is how we get out to the wider internet.

If you have created a server name for your target device, such as yardlights.ddns.net then you will also need to set up a dns address such as:

```
tcpip.dns1 "192.168.0.1"
tcpip.dns2 "8.8.8.8"
```

The first is normally your router address since this will point to the dns given by your isp. Here we have used Google's public dns server for the second one.

This example uses an analogue input on the dScript module to control an ETH002 relay which could be anywhere on the internet.

```
tcpip.hostname  "dS3484"
tcpip.ip        "192.168.0.136"
tcpip.mask      "255.255.255.0"
tcpip.port      17494
tcpip.dns1      "192.168.0.1"
tcpip.dns2      "8.8.8.8"
tcpip.gateway   "192.168.0.1"

clientport      ETH002 "yardlights.ddns.net" 17494 5000

analogport      Adc1         1

digitalport     LedBlue      33
digitalport     LedGreen     34
digitalport     LedRed       35

string  clientOutBuf[10]
string  clientInBuf[10]
int32    clientLength
```

```
int32   yardLightState
thread main(const)
        yardLightState = 2          ; neither on or off to force initial setup
        threadstart YardLightCtl
endthread


thread YardLightCtl(100)
        if Adc1>600 then
            if yardLightState != on then
                clientOutBuf[0] = 0x20      ; Relay Active
                clientOutBuf[1] = 1         ; Relay 1
                clientOutBuf[2] = 0         ; not timed
                ETH002.Write(clientOutBuf, 3, clientInBuf, clientLength)
                if clientLength == 1 then
                    if clientInBuf[0] == 0 then
                        yardLightState = on
                        LedBlue = on
                        threadstart FlashRedLed
                    endif
                endif
            endif
        elseif Adc1<400 then
            if yardLightState != off then
                clientOutBuf[0] = 0x21      ; Relay Inactive
                clientOutBuf[1] = 1         ; Relay 1
                clientOutBuf[2] = 0         ; not timed
                ETH002.Write(clientOutBuf, 3, clientInBuf, clientLength)
                if clientLength == 1 then
                    if clientInBuf[0] == 0 then
                        yardLightState = off
                        LedBlue = off
                        threadstart FlashRedLed
                    endif
                endif
            endif
        endif
endthread


thread FlashRedLed(const)
        LedRed = on
        threadsleep 300
        LedRed = off
endthread
```

Further examples.

For an example of a tcp/ip client, have a look at the multimodule example. Here we use a dScript module as a client to send commands to an ETH008 (which is a server).
Note. The commands tcpip.Read and tcpip.Write are NOT used for client code, they are used by server code only. In client code the name of the client port is used instead. The client does not have a read command, we use ETH008.Write to both send a message to the ETH008 and get its response.

ETH008.Write(TxBuf, 1, Rxbuf, Count)

The four parameters are the buffer containing the data to send, the number of bytes to send, a buffer to receive the response (can be the same buffer if you don't mind it being overwritten) and an int32 variable to receive the count of bytes in the response.

Reading the MAC Address

Reading the MAC address can provide your program with a globally unique number that you can use as a serial number or identifier.

```
string s1[100]
string s2[10]
tcpip.ReadMacAddr(s2)
```

This will read the 6 byte MAC address into the string s2.

If you want it formatted in the traditional way as 6 hex bytes separated with a : character.

```
s1 = {h02} s2[0] ~ ":" ~ {h02} s2[1] ~ ":" ~ {h02} s2[2] ~ ":" ~ {h02} s2[3] ~
":" ~ {h02} s2[4] ~ ":" ~ {h02} s2[5]
```

To display the result on an LCD05:

```
LCD05.Write(s1,0,s1.Length)
```

and you will get:
D8:80:39:BC:E6:4A (or whatever your MAC address is).

UTC clock

Coordinated Universal Time (**UTC**) is an international time standard. Your internet connected module has access to this time by using its UTC port. To use the UTC port add the following in your dScript header area.

```
UTCport      UTC 0 1
```

The first parameter (UTC) is the name of the port that you will use in the program to refer to it. The second is the time zone offset. Here in the UK we want to use GMT so just set this to zero. Most of Europe are on CET (Central European Time) which is 1 hour ahead of UTC so this would be set to 1. The time zone offset can include minutes. For example India which is UTC +5:30 would use:

```
UTCport      UTC 5:30 0
```

The third parameter is a flag for daylight saving time. This only works for countries that put the clock forward 1 hour on the last Sunday in March, and back 1 hour on the last Sunday in October. Set to 0 for UTC or 1 for UTC+daylight saving time.

Accessing the time is done by reading the 7 UTC port variables.

UTC.Year	This is 0 - 99. For the year 2015 it will be 15.
UTC.Month	0 - 11 for January to December.
UTC.Day	1 - 31 for the day of the month
UTC.Wday	0 - 6 for Sunday to Saturday.
UTC.Hour	0 - 23
UTC.Minute	0 - 59
UTC.Second	0 - 59

Note that after powering up the module it takes 15-20 seconds to synchronise with internet time servers, during this time all the above will read zero. The following will display the date and time on an LCD05.

Take a look at the provided “Time” example which displays the current time and date on an LCD05 module.

HTTP web server

Using the dScript web server, you can write and upload your own web pages to the module. Your website can display your own variables in any manner you choose, using AJAX techniques to keep the variables live and updated. You can use buttons on the web page to control things on the module and, of course, your own branding applied to the page. You can use CSS to format your website and include any images, logo's required. Modules have at least 2MB of storage for web pages.

Note that all web files (html, css, javascript, pictures etc) must be stored in one directory. The web compiler does not support a directory structure with multiple levels.

Lets start with a very simple web page, this will be a single button on the page called "Yard Lights". The following html should be saved as index.htm and stored in a webpage directory on your computer.

```
<!DOCTYPE HTML>
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dweb Simple Test Page</title>
</head>
<body>
    <button>Yard Lights</button>
</body>
</html>
```

For convenience we will want our module at a fixed IP address, so also save the following dScript program.

```
tcpip.hostname  "dS3484"
tcpip.ip        "192.168.0.137"
tcpip.mask      "255.255.255.0"

thread main(const)
  do
  loop
endthread
```

It just sets the IP address and mask, make sure your computer is on the same subnet (192.168.0.xxx in this case). The do loop is currently empty as we have no program to run yet. Make sure the web page directory is selected using the blue world web symbol in the menu bar, then load your program.

Type 192.168.0.137/index.htm into your browser and you should see your button at the top left. Note that our web server will (deliberately) NOT serve a default page. You must type in the full page name that you want. Typing the IP address only will result in a file not found error.

You now have your first web page up and running, so lets make that button do something. We will use it to toggle a relay on the module. Also we will get the actual state of the relay from the module and use this to set the button colour.
First we need to add the following to the dScript program to define the relay:

```
digitalport Rly1     1
```

so we have:

```
tcpip.hostname  "dS3484"
tcpip.ip         "192.168.0.136"
tcpip.mask       "255.255.255.0"

digitalport Rly1     1

thread main(const)
  do
  loop
endthread
```

The button line on the web page needs some more code, change it to:

```
<button id="Rly1" onmousedown="newAJAXCommand('dscript.cgi?Rly1=2');">Yard
Lights</button>
```

We have given the button the id "Rly1" so we can refer to it to update the colour. We also added onmousedown="newAJAXCommand('dscript.cgi?Rly1=2');"

The general format for changing something on the module is
"newAJAXCommand('dscript.cgi?name=value');"

Where a name is a defined resource in the dScript program and value is what you want to assign to it. In the case of `digitalport` objects, 0 will clear it, 1 will set it and 2 will toggle it. dscript.cgi is the file that processes these commands, you do not need to supply this file as it's built in.

We also need to add just a little javascript to make it happen. It's a single function containing one line of code (and the start/end script tags)

```
<script type="text/javascript">
function ajaxUpdate()
{
   document.getElementById('Rly1').style.backgroundColor =
      (getValue('Rly1')=='1') ? 'rgba(255,0,0,0.2)' : 'rgba(0,0,255,0.2)';
}
</script>
```

ajaxUpdate is a callback function you must include, it is called whenever new variable data arrives from the module so you can update your web page.

getValue('Rly1') is a function you call from within ajaxUpdate to retrieve the value of your variables. The variable name must be identical to the one used in your dScript program.

The last thing to add is a call to startAJAX() after the web page has been loaded. Do this by modifying the <body> tag like this:

```
<body  onload="startAJAX()">
```

The new html page looks like this:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dweb Simple Test Page</title>
</head>
<body  onload="startAJAX()">
    <button id="Rly1" onmousedown="newAJAXCommand('dscript.cgi?Rly1=2');">Yard
Lights</button>
<br> Temperature is

<script type="text/javascript">
function ajaxUpdate()
{
    document.getElementById('Rly1').style.backgroundColor =
      (getValue('Rly1')=='1') ? 'rgba(255,0,0,0.2)' : 'rgba(0,0,255,0.2)';
}
</script>
</body>
</html>
```

The final items we'll add to this simple web page is a couple of variables, the temperature from the on-board sensor and the input power supply voltage.

Add the following immediately below the button tag.

```
Temperature: <div id="Temperature">?</div>
DC Voltage: <div id="DCvolts">?</div>
```

Then add the following in the ajaxUpdate function immediately below the existing line.

```
document.getElementById('Temperature').innerHTML =
   Number(getValue('Temperature')/10).toFixed(1);
document.getElementById('DCvolts').innerHTML =
   Number(getValue('Volts')/10).toFixed(1);
```

In the dScript program we need to declare the two analogue ports, the two variables Temperature and Volts and add a couple of lines in the do loop to calculate the temperature and

voltage from the incoming ADC values.

```
analogport  TSensor 100 ; on-board temp sensor
analogport  PSU     101 ; DC power voltage

int32 Temperature
int32 Volts

    Temperature = ((TSensor*3223)-500000)/1000
    Volts = PSU*18369/100000
```

The final dScript program is:

```
tcpip.hostname  "dS3484"
tcpip.ip        "192.168.0.136"
tcpip.mask      "255.255.255.0"

digitalport Rly1    1
analogport  TSensor 100 ; on-board temp sensor
analogport  PSU     101 ; DC power voltage
int32 Temperature
int32 Volts

thread main(const)
  do
    Temperature = ((TSensor*3223)-500000)/1000
    Volts = PSU*18369/100000
  loop
endthread
```

You now have a simple web page which you can toggle a relay and show the relay state by the button colour, and display module temperature and incoming voltage.

The calculations for converting the analogue readings into into degrees C and volts were derived like this:

The ADC gives a 10 bit (0-1023) result over the range 0-3.3v. The ADC is returning 3.3/1024 = 3.223mV/bit, or 3223uV/bit. The ADC value is therefore multiplied by 3223 to give us the input voltage in uV. The temperature sensor gives an output of 10mV/degree C, or as we are working in uV, 10000uV/degree C. The temperature sensor has an offset of 0.5v or 500000uV so we need to subtract this from the result. Now we can divide by 10000 go give us the temperature, as an integer, in degrees C. However we only divide by 1000 which means the number is 10 times bigger than it should be. 21.5 degrees will be 215. We will use Javascript in the browser to divide this by 10 and display the result to 0.1 degrees.

A similar trick happens with the input voltage. We use 4k7 and 1k resistors in series to divide down the supply voltage to a safe level for the processor. So 12v will feed 12*(1/(4.7+1)) = 12/5.7 = 2.1v to the ADC. This will convert to 2.1/3.3*1024 = 652. We need change 652 to 12000000uV (or 12v), so we multiply by 12000000/652 = 18405. Because that is full of rounding errors to make the numbers small and readable the actual answer is 18369.

12000000/(((12/5.7)/3.3)*1024) = 18369.
So the voltage is PSU*18369uV. We can divide this by 1000000 to get the volts, however as with the temperature, we only divide by 100000 to give a result 10 times larger. Again, Javascript will divide this by 10 and display the result to 0.1 volts.

Web page security

Now that you can control your module from any browser on any PC/Laptop/Tablet/Phone from anywhere on the planet, what is to stop anyone who knows your incoming IP address from opening your gates, turning lights on or your heating off and generally messing up your life?

Well one way is to change the web page file name from "index.htm" to something harder to guess. How about "EyRz2G5xXu94e.htm". This is almost like using the file name itself as a kind of password. Actually this is pretty good security and I recommend you use it. It is called "security by obscurity", but on its own is still not quite good enough. It is not safe from the so called "man in the middle" attack. This could be when using a cafe/hotel wifi. The web traffic could be monitored and your page name become visible: no more security. Even someone looking over your shoulder could be enough to compromise security.

Web pages could be secured with a password. However unless an SSL connection is used, that password is transferred as plain text and offers no real protection. We could use password/SSL in combination. That would be secure, but at the cost of huge inconvenience.
1. The SSL firmware would add extra cost to every module.
2. You would need to purchase a secure certificate for every module, or use a self signed certificate and dismiss the browser warning every time.
3. Every time, you would still have to enter your username and password.

We wanted a solution that did not add cost or inconvenience to the operator. One where you, and only you, can go straight to the web page and view status or make changes.

Our solution is to store a password on your browser. Only your device/browser combination can access the web pages. In operation this password is never transferred over the network. The server sends three independent random numbers which select three random characters from the password, these are hashed and the hash stored as a cookie. It is this cookie that enables the web page to be displayed, the cookie is then deleted when the browser is closed and also invalidated by the server after a few seconds of inactivity or when you log out. Your web page will likely be updating variables continuously so the page will stay alive as long as you are viewing it.

To add security to our simple web page, add the following to the dScript program header section:

```
html.password "bQq#dm@$%^5*xZ5tY0wN!fi38H_Y3"
```

The password can be anything you want from the ASCII character set in the range 32-126 (0x20-0x7E) excluding " which is the string terminator, and up to 200 characters in length.

With the password in place all requests for pages will require authorisation. If your browser

has the password stored in it, the page will be served. If not, then you will see a page which says you are not authorised to view it.

Installing the password on your browser

Add the following below the `html.password line:`

```
html.setup on
```

This command will enable a built in web page that loads the password into your browser. Here is the complete dScript program for our simple website demo.

```
tcpip.hostname  "dS3484"
tcpip.ip        "192.168.0.137"
tcpip.mask      "255.255.255.0"

html.password "bQq#dm@$%^5*xZ5tY0wN!fi38H_Y3"
html.setup on

digitalport Rly1     1
analogport  TSensor 100 ; on-board temp sensor
analogport  PSU     101 ; DC power voltage
int32 Temperature
int32 Volts

thread main(const)
  do
    Temperature = ((TSensor*3223)-500000)/1000
    Volts = PSU*18369/100000
  loop
endthread
```

Load this into the module and navigate your browser to (in this case) 192.168.0.137/_pw.htm

This built in file _pw.htm contains your password and the javascript needed to load it into the local storage on your browser. When done you can go to

192.168.0.137/index.htm

Now you, and only you, can see your webpage.

Don't forget to change `html.setup on to html.setup off` (or delete that line) to disable access to the password setup page _pw.htm.

Logging out.

To log out add the following to the html on your page:

```
<a href="_loggedout.htm">Log out</a>
```

_loggedout.htm is a built in file that contains the "Logged out" message and a small piece of javascript to delete the authorisation cookie. More importantly, the filename is recognised by the module as a special file and it will invalidate the associated authorisation code.

Here is the index.htm file with the log out code included:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dweb Simple Test Page</title>
</head>
<body>
    <button id="Rly1" onmousedown="newAJAXCommand('dscript.cgi?Rly1=2');">Yard
Lights</button>
    <br> <br>
     Temperature: <div id="Temperature">?</div>
     <br>
     DC Voltage: <div id="DCvolts">?</div>
     <br>
     <a href="_loggedout.htm">Log out</a>

<script type="text/javascript">
function ajaxUpdate()
{
    document.getElementById('Rly1').style.backgroundColor = (getValue('Rly1')=='1') ?
'rgba(255,0,0,0.2)' : 'rgba(0,0,255,0.2)';
    document.getElementById('Temperature').innerHTML = Number(getValue('Temperature')/
10).toFixed(1);
    document.getElementById('DCvolts').innerHTML = Number(getValue('Volts')/10).toFixed(1);
}
</script>
</body>
</html>
```

If you wish, you can write your own _loggedout.htm file and load along with your other web pages. It will be used in preference to the built in file. It must be named _loggedout.htm otherwise it will not be recognised by the module as a special file to delete the authorisation cookie. Also to delete the cookie on the browser you should include the javascript:

```
<script>
  document.cookie = "authorisation=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
</script>
```

Accessing your Webpage from the internet

Here's the problem:
You have your webpage up and running on your local network, for example 192.168.0.150, and you can access the webpage and control the module.
You just go to 192.168.0.150/index.htm, and the page is there.

However you can't get directly at that page from your phone when you are away from home. You can't access it on 192.168.0.150 because your network is not publicly accessible, its a private network address. You will have another IP address. This is the one your ISP gave you for your internet connection, and is the public IP address of your router on the internet. If you don't know what it is you can type "my ip" into Google's search bar and it will tell you. This is the IP address you will use to access the modules webpage.

Everything on the internet uses an IP address and a port number.
When you access a webpage in your browser all you enter is the IP address (or more likely a domain name, but its ultimately translated to an IP address). You don't normally have to enter a port number but its still required. Your browser simply uses the default port number, which is 80 for the web, unless otherwise specified our modules also use port 80 for the webpage. However its a good idea to use a different port number for our boards as will will avoid conflict with any web server you may have on your network.
Pick a number, I'll choose 19321 as our port number. Just make sure its different from the TCPIP.port number.

We need to tell the module what the new html port number is. Do that by including the following in the dScript program:

```
html.port  19321  ; change webpage port number from the default of 80
```

After you have re-loaded the program you can access the webpage with:
192.168.0.150:19321/index.htm

Note that as we have changed the modules html port we need to tell the browser how to find the page with the new port number. Do that by inserting a ':' character and the port number between the IP address and the page name as shown above.

Assuming your routers internet IP address is 86.87.88.89 (I made that up – replace with your actual IP address) you will access the page from anywhere with address:
86.87.88.89:19321/index.htm

However first you have to set up your router to do that.
It's called "port forwarding" or "virtual server", but whatever your router calls it, you need to set it up so that all incoming connections on port 19321 are forwarded to port 19321 on local IP address 192.168.0.150.

Unfortunately there are so many routers out there we cannot give details on all of them. You should consult your routers manual or search Google for details on your specific router.

Email

Emails can be sent from your dScript program to alert you to any events you chose. Emails are sent in plain text, not SSL/TLS.

To send emails from your own domain, you need to set up an email account for your board to use. We set up an dS3484 account on our devantech domain for the test. The following 5 lines are added to the header section of the program.

```
email.from      "dS3484@devantech.co.uk"
email.server    "smtp.devantech.co.uk"
email.port      25
email.username  "dS3484@devantech.co.uk"    ; account was deleted after the test
email.password  "iBa4t_31"                  ; don't use, it won't work
```

The above 5 lines define the account the module will use to send emails.
You will also need to set up the recipient address like this:

```
emailport Devantech "sales@devantech.co.uk"
```

You may have as many recipients as you wish:

```
emailport Devantech "sales@devantech.co.uk"
emailport MyFriend "myfriend@hisdomain.com"
```

Two string and one integer variables are needed to send the email. The string variables are loaded with the subject and message text, the integer variable will receive the response code. This will be zero if the email was sent successfully.

```
string Subject[100]
string Msg[200]
int32 Response
```

To send the email:

```
Subject = "Gate sensor triggered"
Msg = "Any additional information can go here"
Devantech.Send(Subject, Msg, Response)          ; send the email
```

If you do not have your own domain, Google provides a restricted free public smtp server at aspmx.l.google.com. The catch is: you can only send emails to a gmail account, but as gmail accounts are free its no problem.

```
email.from     "dS3484@devantech.co.uk"
email.server   "aspmx.l.google.com"
email.port     25

emailport Devantech "devantech15@gmail.com"
```

Note that username and password are not required or supplied above, and the recipient is a gmail address.
A complete program example to send emails:

```
tcpip.hostname  "dS3484"
tcpip.ip        "192.168.0.136"
tcpip.mask      "255.255.255.0"
tcpip.port      17494
tcpip.dns1      "192.168.0.1"
tcpip.dns2      "8.8.8.8"
tcpip.gateway   "192.168.0.1"

email.from      "YardController@My.Home"
email.server    "aspmx.l.google.com"
email.port      25

emailport Devantech "devantech15@gmail.com"
emailport Personal "MyOwnAddress@gmail.com"

string Subject[100]
string Msg[200]
int32 Response
int32 EmailSent

digitalport IO1          41
digitalport IO2          42

digitalport LedBlue      33  ; Blue Led,    sending email(s)
digitalport LedGreen     34  ; Green Led,  success
digitalport LedRed       35  ; Red Led,     failed

thread main(const)
        LedGreen = off
        EmailSent = 0
        do
            if IO1 then
                if EmailSent==0 then
                    LedBlue = on
                    Subject = "Gate sensor triggered"
```

```
                    Msg = "Any additional information can go here"
                    Devantech.Send(Subject, Msg, Response)  ; send the email
                    Personal.Send(Subject, Msg, Response)   ; another email
                    EmailSent = 1
                    LedBlue = off
                    if Response==0 then
                        LedGreen = on
                    else
                        LedRed = on
                    endif
                endif
            endif
            if IO2 then
                LedRed = off        ; reset so we can send emails again
                LedGreen = off
                EmailSent = 0
            endif
        loop
endthread
```

EasyMail

Easymail is a very easy to use and secure email system for our modules.

To use easymail , place the easymail keyword in the header to declare the recipient and from addresses. You may have as many recipients as you wish.

```
easymail    mySelf  "me@example.com", "dScript-mail.uk"
easymail    notMe   "notme@example.com", "dScript-mail.uk"

The from address should be left as dScript-mail.uk
```

You need a couple of strings for the subject line and message body.

```
string  subject[100]
string  message[500]
int32   eResponse
```

and you can send an email:

```
subject = "Main Gate Opened"
message = "Additional information can go here"
mySelf.Send(subject, message, eResponse)
```

The response code of sending the email will be in eResponse.
The codes are:

0	The email was sent successfully
non-zero	The email failed

Easymail has a limit of 100 emails/hour/module. If you hit this limit your emails will be ignored (they will fail with error code 11) until time allows more to be sent.

Easymail uses port 7200.
Normally routers only block incoming ports by default with outgoing ports left open, so easy-mail will work ok. If you have a corporate system that blocks all outgoing ports then you will need to open port 7200 to get easymail to work.

Multi-threading

What is multi-threading?

Multi-threading is the ability to run many parts of your program at the same time. It is like having a separate CPU for each part, each running its own little program. Each part or section of your program is called a thread. dScript is a native multi-threading compiler. Any number of threads may be created, up to the limit of available RAM memory.

Threads are created with the thread declaration and finish with endthread, like this:

```
thread myFirstThread(1000)
int32  mySecondsCounter
       mySecondsCounter = mySecondsCounter + 1
endthread
```

This tells the compiler that you are declaring a new thread called myFirstThread. The 1000 attribute means this thread will be triggered to run every 1000mS. Threads may run constantly or in response to internal or external trigger events.

Endthread terminates the thread definition and suspends thread execution until it is triggered again each second (1000mS). The next time it is triggered it starts again at the beginning. This very simple thread therefore implements a counter which counts up by one each second. dScript programs always have at least one thread. When a program starts it jumps to the "main" thread. This is just another thread to dScript, and the only one to start running automatically. All other threads you declare are created in the stopped state. They will not run until they are started with threadstart. This will typically be done in “main”, but threads may be started from any other running thread.

```
thread main(const)
       threadstart myFirstThread
       do
       … other instructions …
       loop
endthread
```

As with all threads, main should loop forever (as shown) or suspend itself with threadsuspend or just allowed to stop at endthread.

Thread commands.

thread, threadstart, threadsleep, threadsuspend, endthread

thread <name> (<trigger option>)
This is used to declare a thread, it is followed by the name of the thread. Trigger option selects a way to automatically run the thread, the options are:

thread myFirstThread(const)
This thread is a constantly running thread and has no trigger options, if it suspends itself with threadsuspend then it will not run again unless re-started by another thread.

thread myFirstTimer(500)
When a number is used as the trigger option, it is a time in mS. There are 1000mS in one second, so this example runs every 500mS or half second. The thread should have completed its task and have suspended itself with threadsuspend before the trigger event comes around again.

```
thread myTcpipCmd(tcpip)
```
This thread will be triggered when an incoming tcpip packet arrives.

```
thread myInterrupt1(input 41)
```
Threads may be triggered when an input changes state. This example is triggered when the input becomes active.

```
thread myInterrupt2(!input 42)
```
Putting a ! before the input will cause the thread to be triggered when the input goes inactive.

```
thread myInterrupt3(^input 43)
```
A thread may be triggered on both active and inactive transitions by putting a ^ before the input.

If another interrupt occurs whist the pin triggered thread is running then the thread will be run again as soon as it suspends. Only one trigger is stored, if multiple triggers come in they will be ignored. Generally you should arrange your timing so that the thread has ample time to execute before another trigger occurs.

```
thread pulseOutput(myVar)
```
This thread will be triggered when myVar becomes non-zero, The variable myVar must have been previously declared. myVar must be reset to zero before the thread ends or it will immediately re-trigger.

threadstart <name>
This is used to start a thread. Threads are created in a stopped state (not a suspended state) and will not run until the threadstart command is issued. Note that threadstart will not actually start the thread unless it is declared as a const thread. For all other threads it "arms" the thread so that it will run on its next trigger event.

threadstart myFirstThread
This will arm the thread and make it ready to run. A const thread will start immediately. All others will be armed and will run when their trigger event occurs.

threadsleep <time mS>
This causes the current thread to sleep for a number of mS. Threads which are sleeping do not consume CPU processing time. When the time expires the thread will resume executing instructions with the instruction immediately following the threadsleep command.

threadsleep 0
In v4.07, this will resume execution immediately. In previous versions the thread would stop.

threadsleep 50
This will cause the current thread to suspend execution for 50mS

threadsuspend
This stops execution of the current thread. It may be used to conditionally stop a thread if there is no more processing to do. The thread is suspended rather than stopped, it still remains armed and will start over at the beginning on the next trigger event.

endthread
The end of the current thread definition. All threads must end with this.
If program execution reaches this point and the thread has not already been suspended, this command will also suspend the thread.

main

The only thread you must have is "main". This is the beginning of your program and dScript will automatically start running at "main" after a power up. "main" may be located anywhere in your program, it does not have to be at the start. “main” is the only thread that is automatically started by dScript. All other threads should be explicitly started within the program. It must be declared as type “const”.

It is declared like this:

```
thread main(const)
…
do stuff
…
endthread
```

Note - Although threads are declared in a similar way to functions, the similarity ends right there.
You cannot call a thread. They are started and then run independently.
You cannot return from a thread – there is no-where to return to. The one (optionally two) “parameters” are to declare the thread type and stack size.

Performance

For those interested in how fast the module executes dScript commands, there is a system variable (system.DespatchCounter) that counts up by one each time a dScript instruction is executed. It counts all instructions in all threads so its a system total. The following program will display the execution speed on an LCD05 module.

A timer thread is set up to read the system.DespatchCounter once per second, reading the system.DespatchCounter will automatically reset it to zero. Therefore reading it each second gives the total instructions executed in the last second. A loop in the main thread displays the count on the LCD05.

If you run this program you may have a bit of a shock. It will say you are executing just 11 instructions per second, why so few?

dScript is a multi-threaded system, and those threads can be suspended (with the threadsuspend command) or sleeping (with the threadsleep command). There are two threads running, the Timer1 thread and the do loop in the main thread. The timer thread runs once per second and has just two instructions. That accounts for 2 of our 11 instructions.

Take a look at the do loop in the main thread. It has two 500mS sleep instructions used to flash the red led. So this thread also only runs once per second and has 9 instructions The "do" is not an executable instruction, the compiler just notes its position for the "loop" instruction to jump back to. In total just 11 instructions need be executed per second. This demonstrates the power of dScript. By not wasting CPU time running around endless loops waiting for something to happen, the CPU is available on demand when an event calls for service.

So how fast can it go?

To load the CPU to its maximum we need a thread that does not suspend or sleep, that just keeps running forever. Notice that we have a thread called "AlwaysRunning" but its threadstart command is currently commented out, so its not actually running yet. Remove the comment and load the program again. Now you will see a massive increase in speed, over 50,000 instructions/second. That is running two reasonably complex calculation instructions in the "AlwaysRunning" Thread, both of which includes an analogue to digital conversion, and the loop instruction to do it again.

Now comment out the two calculation instructions just leaving an empty do-loop and run the program again. This time you will see around 476,000 instructions/second.

A real program will contain a mix of complex and simple instructions and 100,000 to 200,000 is a reasonable expectation, however it does depend on the program. You can put your own mix of instructions within the do-loop and see what you get.

```
digitalport LedRed      35

serialport LCD05 1 10 90

int32   Rate

string  s1[100]

analogport  TS1      100 ; on-board temp sensor
analogport  PSU      101 ; DC power voltage

int32 BrdTemp
int32 Volts

thread Timer1(1000)
        Rate = system.DespatchCounter
endthread

thread AlwaysRunning(const)
        do
            BrdTemp = ((TS1*3223)-500000)/1000
            Volts = PSU*18369/100000
        loop
endthread

thread main(const)
        threadsleep 100
        threadstart Timer1
;       threadstart AlwaysRunning

        s1 = "   Instruction Despatch"
        s1[0] = 12
        s1[1] = 19
        s1[2] = 4
        LCD05.Write(s1,0,s1.Length)

        do
            s1 = "  Rate = " ~ {D6} Rate ~ "     "
            s1[0] = 2
            s1[1] = 21
            LCD05.Write(s1,0,s1.Length)

            LedRed = on
            threadsleep 500
            LedRed = off
            threadsleep 500
        loop
endthread
```

Notes