Tiny 1 (T1) User and Developer's Guide

Rev. 3.3 01/31/11

Revisions

rev.	date	by	changed
2.0	12/4/06	SPJ	Initial
3.0	1/9/11	SPJ	Major update. Explains all commands. Improved guide to scripts
3.1	1/14/11	SPJ	Fills out explanation of expression. Adds 'Script File Formats' section.
3.2	1/30/11	SPJ	Current per script.c rev 1.4. better '$', '$$' printout control. Conditional pause.
3.3			

Table of Contents

Introduction

This manual describes Tiny1, the operating system which runs the IBench testers. First, because I will be asked... why is it called Tiny1? The name comes from the core scheduler/RTOS which runs e.g Sequel. This was a stripped down version of the IBench scheduler, and is sized to run in systems in the 64K/4K range. Hence 'Tiny'. (There's also a 'Nano' 16K/2K systems).

This name was later used to cover object / text UI system which runs in the testers and the Sequel Simulator. More basically though, this system grew over time; when you first choose a name you don't know where things will end.

1 Classes

The Tiny1 system has classes for testgear. Some of these derive from the IBench and have the same behaviour as their IBench namesakes. Others were added with the needs of testgear in mind.

Right now the classes are just listed. I'll fill out the behaviours later. In the meantime go to the command reference. Each class has a command to talk to objects of that class. You can see the behaviour there.

Threads

These are internal software processes, Read status, stopped etc.

Lists

Lists of obejcts

Scalar

A single integer, can be read or written

Vector

A vector, can be read or written

Cal (calibrations)

A single cal number (integer), Can be read, written, stored in Flash, reset to default

CalV (calibration vectors)

Same as Cal but a vector

Sensors

Scaled to standard units, can be read

Actuators

Scaled to standard units, can be read, written, reset

InPin

Input line, 1 or 0

Outpins

An actuator which can be 1 or 0

Outports

An array of up to 16 **outpins**, which can be written or read as a group.

Timers

These are used internally by scripts to genrate asynchronous actions.

Servo

A proportional-integral servo. Each servo can be programmed with a Sensor (to read), an Actuator (to drive), set-points for input and output and proportional and integral gain terms.

2 Editing at the terminal

Commands are entered via a line editor which works with an ANSI terminal. The editor is full-duplex (echoes what's typed).

<Enter> completes a line

<BkSpc> deletes before the cursor

<Del> deletes at the cursor.

<left-arrow / right-arrow> moves the cursor

<Ctrl-Z> cursor to line start

<Ctrl-X> cursor to line end

<Ctrl-A> clear whole line

<Ctrl-E> clear to end of line

<Ctrl-I> Toggle insert / overwrite (default is insert)

The editor stores the 5 most recent lines

<Up arrow>, <down arrow> select the next and previous stored lines

Once <Enter> is pressed the command line is processed. A command may have arguments, each separated by one or more spaces. The command and its args are entered as one line, with CR or CRLF to finish.

Equipment may not have all commands installed. To see whats a particular device has type 'help'. The help text usually describes the command format.

3 (User) Commands

Each T1 class has a command to control / read it. There are system-level commands not specific to any class. These together are the *User Commands,* as they can be send from the terminal at any time. There are also *Script Commands*, which provide control inside scripts and are only recognised when encountered in a executing script. See sec.5 Scripts for information on these.

User commands returns and scripts

Any user command can be used in a script and will have the same effect there as when entered at the terminal. Each user command returns success (1) or fail (0). The result controls the ***lastFailed*** and ***anyFailed*** flags in the script engine. These flags allow scripts to branch on the results from user commands. See 5.6 Conditions and fail flags in the scripts section.

Most user commands return 1 = success, unless they are missing arguments or their argument is a non-existent object. Only *eval* (evaluates expressions) and *compare* (compare variables against limits) would usually be used to control to flow of script execution.

3.1 Summary

System / General

echo	- Full / half duplex; the default is full
help / ?	- Help on any command or object. or a list of all installed commands

quiet	- Silences debug printfs
objs	- List all objects
read	- Reads any readable object
write	- Writes any writable object

Class-specific

sensor	- may be read
thread	- Monitors and controls thread, if they are listed as objects
actuator	- may be written or read; and have a default value
outPin	- an actuator which can be written 1 or 0
outPort	- an array of up to 16 outPins, read or written as a group.
inPin	- input line reads 1 or zero
txtfile	- holds scripts
script	- run scripts
cal	- scalar calibrations. Can be stored, have a default,
calv	- vector calibrations
eval	- Evaluates and expression – See expressions
svc	- Service mode control
compare	- compares scalars or vector against limits
servo	- proportional/integral (PI) servo

3.2 Command reference

Syntax for command formats:

(ss)	= alternative text for a command e.g (?) is the same as 'help'
< nn >	= required argument;
[nn]	= optional argument
[nn]... or < nn >...	= one or more repeats
a | b | c	= alternatives, a OR b OR c
'raw'	= string literal

echo <'on' | 'off' | string to echo>

Character echo on/off. The power up default is echo on. This is the setting you will normally want as you are using the system's built-in line editior.

help (?) <command or object name>

Prints built in help on any command or object.

quiet <1|on |0|off>

Blocks debug printouts. Default is printouts enabled. There are usually not many of these, so leave it alone.

objs

This lists all system objects, grouped by class. Use this command when you can't remeber the exact name or spelling of an object

read <readable scalar or vector>... ['raw'] | ['numsOnly']

This reads and prints one or more readable scalars or vectors. By default it prepends each value with a name and appends each with units, if the variable has units.

```
> read Pres6
: Pres6 = 100.5 kPa

> read v1 Pres6
: v1 = 0.00 Vdc  Pres6 = 100.5 kPa

> read CellTmprs
: CellTmprs = 22.3 22.5 21.6 22.7 22.0 23.1 degC
```

Appending 'raw' prints the unscaled integer representation of the variable

```
> read Pres6 raw
: Pres6 = 10052
```

'numsOnly' omits the units and the name. This is useful for printouts to be pasted into Excel, especially as part of a repeated read in a script. The following line take 5 temperature readings 5 secs apart.

```
> repeatIntvl 5 5.0 : read CellTmprs numsOnly
:  22.3 22.5 21.6 22.7 22.0 23.1
22.4 22.5 21.2 22.7 22.1 23.3
22.2 22.4 21.5 22.8 21.9 23.0
22.6 22.4 21.3 22.8 22.2 23.2
22.1 22.3 21.5 22.6 22.3 23.2
```

Both these args give the raw number(s), unnamed.

```
> read Pres6 v1 raw numsOnly
: 10052   0
```

write <writable scalar or vector> <num>... ['raw']

This writes values to a writable scalar or vector.

```
> write v1 8.4
> read v1                        : v1 = 8.40 Vdc
```

If 'raw' is appended then 'num' is read as the unscaled internal representation of the variable.

```
write v1 2300 raw
> read v1                        : v1 = 2.30 Vdc
```

If 'num' is outside the boundaries of the internal integer representation then it will be clipped.

```
 write v1 120.6
> read v1                        : v1 = 32.77 Vdc
> read v1 raw                    : v1 = 32767
```

sensor <sensor name> <action = ('read' ['src'] | 'report') ['raw'] >

Reads a *Sensor* in various formats. Also dumps the sensor gain and offset settings.

sensor Pres6 read	- pressure sensor 'Pres1' in kPa
.. .. read src	- returns raw input to sensor
.. .. read raw	- pressure sensor 'Pres1' in internal units
.. .. report	- shows gain, offset and status flags

Examples:

```
 sensor Pres6 read              : Pres6 = 100.5 kPa
> sensor Pres6 read raw         : Pres6 = 10055
> sensor Pres6 read src         : 3416
> sensor Pres6 report
      : Pres6 = 100.6 kPa raw = 10058 src = 3416 flags = 0
      Config:  offset = -1055  numerator = 1186  denominator = Pres6Gain =
0.045 V/kPa flags = 2
```

Note that a sensor can also be read with '***read***' (it's a readable scalar)

thread <thread_name> ['run' | 'stop' | 'restart' | 'report'

To control system threads, if your application has them listed as objects. Normally you don't mess with threads. For engineering really.

actuator <actuator name>
<action = 'read' ['raw'] | 'write' value ['raw']| 'reset' | 'lock' | 'free' |'report' >

Controls *Actuators*, such as pumps and valves. Summary

actuator pumpV read	- prints current value e.g 2.8V
.. .. .. read raw	- prints raw value 2800
.. .. .. write 3.3	- pumpV <- 3.3V
.. .. .. write 3300 raw	- pumpV <- 3.3V
.. .. reset	- resets to default
.. .. lock	- blocks application from writing pumpV
.. .. free	- undoes lock
.. .. report	- current value, limits and lock status

For example, a pump drive duty cycle:

```
> actuator Pump1Drv0 read       : 5.0 %
> actuator Pump1Drv0 write 7
> actuator Pump1Drv0 read       : 7.0 %
> actuator Pump1Drv0 read raw   : 700
> actuator Pump1Drv0 report
    :  value = 7.0   locked = 0  limits: 2.0 40.0 %
> actuator Pump1Drv0 lock
> actuator Pump1Drv0 report
    :  value = 7.0   locked = 1  limits: 2.0 40.0 %
```

If a value written to an *Actuator* exceeds the Actuator limits then the value will be clipped

```
> actuator Pump1Drv0 write 60
> actuator Pump1Drv0 read       : 40.0 %
```

'reset' returns the Actiator to it's preset default value

```
> actuator Pump1Drv0 reset
> actuator Pump1Drv0 read       : 5.0 %
```

Note that Actuators can be controlled with ***read*** and ***write*** commands.

```
> write Pump1Drv0 25
> read Pump1Drv0                : Pump1Drv0 = 25.0 %
```

outPin <pin name>
<action = 'read' | ('write' '0'|'1'|'on'|'off'|'set'|'clear') | 'reset' | 'lock' | 'free' | 'report' >

Controls *OutPins* i.e single digtial outputs. These are like Actuators except that they are only1 and 0 (on and off). For e.g valves.

outPin valve3 read	- print current setting of valve3
outPin valve3 write on	- valve3 <- on
.. .. write 1	- same as 'on'
.. .. reset	- puts valve3 into reset state
.. .. lock	- so application can't write valve3
.. .. free	- undoes lock
.. .. report	- current setting and lock status

Examples:

```
> outPin Led1 read              : 0
> outPin Led1 report            : value = 0 locked = 0
> outPin Led1 write on
> outPin Led1 read              : 1
> outPin Led1 lock
> outPin Led1 report            : value = 1 locked = 1
```

inPin <pin name> <action = 'read'>

InPins are digital inputs. They are just read. Note that the 'read' command does the same.

```
> inPin Sw1 read                : 1
> read Sw1                      : Sw1 = 1
```

outPort <port name> <action = 'read' | 'write' <val | 'off'> | 'reset' | 'lock' | 'free' | 'report' >

A group of individual pins can be consolidated as an *OutPort* and accessed all together with a single command. The options are the same as for ***outPin*** except that read and write parms can be a (hex) number which controls multiple inputs and outputs.

compare <n1 = scalar or vector>
<comparision = 'greater''>' | 'less''<' | 'inside' |'equal''==' | 'notEqual' '!=' '<>'>...
<n2 = limit | lo limit> [n3 = hi limit] ['raw']

Compares Scalars or Vectors against limit(s). If the compare condition is **not true** then prints an error message.

Returns: 1 if the condition was true, else 0.

'***n1***' may be (the name of) a *Scalar* or *Vector*. '***n2***', '***n3***' may be *Scalar*, *Vectors* or literal numbers. How '***n2***' and '***n3***' are read depends on the type of variable and type of comparsion.

If '***n1***' is a Scalar then '***n2***', '***n3***' must be either *Scalars* or literal numbers. If the comparsion argument is 'inside' then both arguments are required, else just ***n2*** is needed. If '***raw***' is appended to the command the literals are taken to be internal numbers, otherwise literals are scaled before being compared to the internal, integer, value of the Scalar. Examples:

```
> read v1                       : v1 = 5.00 Vdc
> compare v1 > 3                ; true, so no message printed
> compare v1 > 7                : v1: under limit  val = 5.00 Vdc min = 7.00
> compare v1 inside 7 10.3
    : v1: outside limits  val = 5.00 Vdc min = 7.00 max = 10.30
> read v1 v2 v3
    : v1 = 5.20 Vdc  v2 = 3.30 Vdc  v3 = 6.40 Vdc
> compare v2 inside v1 v3
    : v2: outside limits  val = 3.30 Vdc min = 5.20 max = 6.40
```

If '***n1***' is a Vector then ***n2*** may be a *Vector* OR '***n2***', '***n3***' may be literals. If '***n2***' is a *Vector* then how it is read depends on the type of comparison. If 'greater' or 'less' then '***n2***' compared, element by element against '***n1***'. If 'inside' then '***n2***' is read as a series of number-pairs, each pair being a lower limit followed by an upper limit.

```
> read O2Cells
    : O2Cells = 6.55 4.20 2.30 0.69 1.80 6.88 mV
> compare O2Cells > 5.0
: O2Cells[1]: under limit  val = 4.20 mV min = 5.00
O2Cells[2]: under limit  val = 2.30 mV min = 5.00
O2Cells[3]: under limit  val = 0.69 mV min = 5.00
O2Cells[4]: under limit  val = 1.80 mV min = 5.00
> compare O2Cells inside 5.0 10.0
: O2Cells[1]: outside limits  val = 0.00 mV min = 5.00 max = 10.0
O2Cells[2]: outside limits  val = 2.30 mV min = 5.00 max = 10.0
O2Cells[3]: outside limits  val = 0.69 mV min = 5.00 max = 10.0
O2Cells[4]: outside limits  val = 1.80 mV min = 5.00 max = 10.0
```

cal <calibration name>
<action = 'read' | 'write' value_to_write | 'store' | 'recall' | 'dflt' | 'report' > ['raw']

Controls *Calibrations*. Like the IBench, these have current, stored and default values. For example, 'Ox1Gain' is the amplifier gain tweak in the O2 tester.

cal Ox1Gain read	- print value of 'n1' with units
cal Ox1Gain read raw	- print raw value
.. .. write 3.85	- set to 3.85 (scaled)
.. .. write 3850 raw	- same as above but write raw value
.. .. default	- reset current value to default
.. .. store	- copy current value to NV store
.. .. recall	- copy NV value to current
.. .. report	- show all data on 'Ox1Gain'

Examples:

```
  cal Ox1Gain report
: Ox1Gain: hdl = 21 val = 2300 stored = none min = 2231 max = 2369 dflt = 2300

> cal Ox1Gain write 2340
> cal Ox1Gain store
> cal Ox1Gain report
: Ox1Gain: hdl = 21 val = 2339 stored = 2339 min = 2231 max = 2369 dflt = 2300
```

calv <cal_vector_name>
<action = 'read' | 'write' index value | 'store' | 'recall' | 'dflt' | 'report' > ['raw']

This handles a vector of multiple *Calibrations*. They are written individually but other operations act on the whole group.

eval <unary, binary or ternary expression>

Evaluates and prints the value of an expression. See sec.4 Expressions (below).

servo <servo name> <action = (
'make' <sensor name> <actuator name> [input set point] [output set point]
[proportional gain] [integral gain]
| 'run' | 'stop'
| 'inPoint' <input set point>
| 'outPoint' <output set point>
| 'pgain' <proportional gain>
| 'igain' <integral gain>
| 'report'
| 'showRun'

This command specifies and controls 'servo' objects (see Servo class above).

In the 'make' subcommand, the 1st 2 args are required. One or more of the other parms may be supplied (in order). Any parms not supplied default to zero. Giving all of the parms in 'make' completely specifies the servo. They can also be added or changed individually using the 'inPoint' 'outPoint', 'pgain' and 'igain' subcommands.

'run' starts the servo; 'stop' suspends it. The servo set-points and gains may be changed while the servo is running.

'report' shows all the servo settings; it also display the current sensor reading and actuator settings.

'showRun' enables a runtime printout of the servo activity. The printout shows the sensor and actuator values plus the error accumulator. This helps a user tune up a servo. The printout is canceled when a 'stop' is sent.

Examples; make an empty servo:

```
> servo servo1 make Pres6 Pump1Duty0
> servo servo1 report
: sensor = Pres6 actuator = Pump1Duty0
   inSetPt =  0.0 kPa outSetPt = 0.0 % pGain = 0.0 iGain = 0.0 kPa/%
   running = 0 Pres6 = 99.5 kPa  Pump1Drv0 = 5.0 %
```

Note that 'report shows the current reading from the *Sensor* and the current setting of the *Actuator*. Now make a complete servo:

```
servo servo1 make Pres6 Pump1Duty0 90 4.8 2 1.6
> servo servo1 report
: sensor = Pres6 actuator = Pump1Duty0
   inSetPt = 90.0 kPa outSetPt = 4.8 % pGain = 2.0 iGain = 1.6 kPa/%
   running = 0 Pres6 = 99.5 kPa  Pump1Drv0 = 5.0 %
```

Enable the pump power supply and run the servo. The debug report (every 3 secs) shows the internal integers of the servo in action.

```
> write Pump1Vdrv 6.0           ; Set pump power to 6V.
> servo servo1 run
 Servo: sens 8990  act 280  set 9000  pGain 40 iGain 10
  ofs 480  err -10  acc -3612 min/max 18999
Servo: sens 8995  act 279  set 9000  pGain 40 iGain 10
   ofs 480  err -5  acc -3823 min/max 18999
```

How to set up a (pneumatic) servo

1) On the O2 tester, a servo controls the pump (the A*ctuator)* in response to a pressure or flow read from a *Sensor*. There can be different servo settings optimised for each pneumatic condition.
2) Set up or check the control range for the pump.
   The main pump in the O2 tester has a PWM drive. There's also a bleed to add flow to the pump if it needs that. Set up the valves etc as they will be under the servo. Find the combination of voltage (*Pump1Vdrv*) and base PWM (*Pump1Duty0*) at which the pump has a reasonable control characteristic. Do this by *write*-ing the pump and observing the resulting pressure and flow. Record the *Sensor* and *Actuator* settings at the desired set-point.
3) Using these *Sensor* and Actuator settings, make the servo <*servo servo1 make ....*'>. Start with a low '*pgain*' setting e.g 1.0 and with *igain* = 0.
4) Enable run-time printouts <*servo servo1 showRun*> and run the servo <*servo servo1 run*>. Check that it nails the set point with a fairly small proportional error.
5) Try increasing the proportional gain <*servo servo1 pgain 'nnn'*>. If you go too high the servo will become unstable. Then back off at least x2 - x4 for a stability margin.
6) Try adding some integral gain <*servo servo1 igain 'nnn'*>. This should reduce variations in the set-point over time. Again, if you go too far the servo will wing-wang. Back off.
7) If the 'igain' setting is much lower than the 'pgain' then the servo is probably running too fast. Reduce the run rate using 'runIntvl' until the two numbers are about equal.
8) Remember the settings. they will go in the servo control script.

txtfile 'report' | <fileNum <<'write' ['term']>|'clear'|'list'>>

Handles text stores in Flash. There are a fixed number of these; the number and size of each depends on the application. They are usually used to hold scripts.

txtfile 1	report	- shows numbers, size and contents of all files
..	write	- to stream a text file from terminal
..	write term	- to type text in line by line
..	clear	- erases fileNum'

.. list - prints contents of 'fileNum'

Examples, from the O2 tester.

'report' shows each file and how many bytes it contains If the file has some test it shows the 1st line. So with scripts it's useful to quote the script name and version on the 1st line.

```
> txtfile report
: Bank 0 start 8000 size 2048 contains 0
Bank 1 start 8800 size 2048 contains 70
      ---- $ O2 Gain linearity ver1 12/2/10
Bank 2 start 9000 size 4096 contains 0
Bank 3 start a000 size 4096 contains 0
Bank 4 start b000 size 4096 contains 0
Bank 5 start c000 size 8192 contains 0
Bank 6 start e000 size 8192 contains 0
> txtfile 1 list
:$ Flash Led1 10 times
repeat 10
Led1 = 1
wait 1
Led1 = 0
wait 1
endrepeat
$ done
```

To stream text from a file via the terminal first set the inter-char send interval to at least 1msec. Then select the file to be sent. Make sure it will fit in the store.

```
> txtfile 2 write
: Send ASCII now, > 1msec inter-char interval; Esc to quit
```

Send the file, then hit <esc> once its sent. The terminal will print the file contents:

```
Echo to check:
:$ Flash Led1 10 times
repeat 10
Led1 = 1
wait 1 ..... etc
```

There's no need to erase existing text from a store before sending a new text file.

script <script_number> <'run' | 'write' | 'list'>

To load and run scripts. See sec.5 Scripts (below) for details on this command

4 Expressions (Eval or <)

T1 supports unary, binary and ternary expressions. They can be evaluated on the command line using 'eval' or they can by conditional expressions in scripts. Some quick Examples:

On the command line

```
eval SupplyV             = 12.4V   (does the same as read)
eval SupplyV > 10        = 1 (true)
eval Pres1               = 90kPa
eval Pres1 + 10          = 100kPa
eval Pump1Drv = 6.2      = 6.2V   (Pump1Drv is now 6.2V)
eval v1                  = 1V  (if scalar v1 if defined as scaled to volts)
eval Pump1Drv = v1 + 1   = 7.2V (Pump1Drv is now 7.2V)
eval Pump1Drv raw        = 7200
eval Pump1Drv = 3000 raw  = 3000  (the internal representation for 3V)
```

In scripts:

```
        quit Pres6 > 100    ; will quit if Pres6 > 100kPa
        while Sw1 == 0      ; loop until Sw1 pressed
```

4.1 Number Literals

Literal numbers in expressions may be decimal integers, decimal floating point, hexadecimal (0xnnn) or binary (0bnnn). Floating point numbers are rounded down to their internal integer representations Binary numbers may be diced up by underscores in any position. Examples:

```
        > eval 348                  : 348
        > eval 25.23                : 25
        > eval -1.24E3              :-1240
        > eval 0xBAD                : 2989
        > eval 0b1100_0011          : 195
```

If an expression contains a scaled variable e.g sensor Pres6 in kPa, then literal numbers are read in the units of the variable e.g

```
        > eval Pres6                : 99.9 kPa
        > eval Pres6 > 100.4        : 0
        > eval Pres6 - 50           : 49.9 kPa
```

If 'raw' is appended to an expression and that expression contains a variable then any literal is read as the internal integer representation of the variable.

```
        > eval Pres6 raw            : 9992
        > eval Pres6 > 100          : 0
        > eval Pres6 > 100 raw      : 1
```

Arguments are converted to integers **before** being used in expressions as this is how signal processing happens in most tester apps. e.g

```
        eval 3 + 4  = 7   and eval 3.7 + 4.6 = 7
```

4.2 Binary Operators

T1 supports all the usual 'C'-style operators. Operators and precedence are:

assignment > comparision > arithmetic and logical i.e

[**=, +=, -=, *=**] (assignment (with add / subtract / multiply)) >

[**>, <, >=, <=, ==** (Equal), **!=** (not equal)] >

[**+ -** *(multiply) \ (divide) **&&** (and) **||** (or)]

Within each class evaluation is from left to right ie

```
        > eval 10 + 4 / 2           : 7
```

In logical operations any non-zero number is '1' = TRUE.

```
        > eval Led1                 : 1
        > eval Led1 && 0            : 0          ; 0 is FALSE
        > eval Led1 && 43           : 1          ; 43 is TRUE
```

4.3 Assignment

The l-value for of an assignment must be a writable object. Trying to assign to a non-writable object returns 'fail' = 0;

```
        > eval Led1 = 1          : 1          ; Light Led1
        > eval Pres6 = 30        :  0.0 kPa   ; Illegal operation, returns 0
```

(Illegal operations return 0. They should print error messages too; but right now only a few do.)

4.4 Unary Operators

T1 supports '-' (minus) and '!' (not). These may be prepended to any variable.

```
> eval -Pres6                : -99.9 kPa
> eval Led1                  : 0
> eval !Led1                 : 1
> eval !Pres6                : 0   ; Not of any non-zero value is 0
> eval Pres6 && !Led1        : 1   ; Can combine logical and numeric
```

4.5 Vectors

T1 has operations which return a *Scalar* result from a *Vector*. The syntax is < vector_name.suffix > where 'suffix' is an operation. The operations are 'min', 'max' 'mean' = 'avg'.

Vectors elements can be access by an index, again using the 'dot' notation. INdices are 'C'-style; the 1st element is index 0 (zero).

```
read O2Cells            : O2Cells = 6.47 1.10 0.00 0.69 0.00 6.80 mV
> eval O2Cells.1               : 1.13 mV    ;
> eval -O2Cells.1              : -1.13 mV   ;
> eval O2Cells.max             : 6.78 mV
> eval O2Cells.mean            : 2.50 mV
> eval O2Cells.avg > 3.2       : 0
> eval O2Cells.min - O2Cells.max : -6.80 mV
```

5 Scripts

The T1 system can be built to run scripts. At their simplest these are lists of commands, as they would be sent from the terminal. They are stored in text file areas and are run from there. This allows T1 systems to be programmed with complex behaviour and run this behaviour standalone.

5.1 Loading Scripts

Scripts are held in text stores (files) in Flash. There are usually 2 or 8 of these, from 512 bytes or 8K each. The files are numbered 0,1... At present each will hold one script, which is labeled with the same number as the file.

Scripts can written as text files on a PC and streamed into Flash from a terminal. There are two ways of doing this. The first is to send using the 'txtfile' command. This enters the text into Flash verbatim.

```
> txtfile 1 write <Enter>
: 'Send text now, 1msec inter-char interval; Esc to quit'
```

Now start the file send from terminal. The text will be written to Flash as it is streamed from the terminal. You can interrupt this by pressing 'Esc'. Otherwise T1 will finish up about 2 secs after the last char is sent.

The other option is to use the 'script' version of the same command.

```
> script 1 write <Enter>
: 'Send text now, 1msec inter-char interval; Esc to quit'
```

The difference is that 'script' strips comments and multiple spaces from the text. This allows scripts to be well commented as PC files without overflowing the modest-sized Flash text stores.

Short scripts and text can be hand-typed at the terminal. Adding 'term' to the write command brings up the the same line editor as for normal commands.

```
> txtfile 1 write term <Enter>         ; 'term' = terminal mode
: Enter text by line. <Enter> commits current line to flash. Esc to quit
> $ 1st script line
> $ 2nd script line etc.....
```

Here a line isn't committed until <Enter> so you can correct mistypes. There's no 'script' version of this; it's assumed you us this just for small files.

5.2 Script Format

Scripts are lines of ASCII text, terminated by either LF or CRLF.

Each line can be up to **100** chars long. If a line is longer the extra chars are ignored, with unpredicatable results.

Scripts can contain comments. A comment is from the first semicolon (;) on any line to the end of that line. Comments may be stripped out when the script is stored to Flash (see above) or they may be left in, in which case they are ignored.

Each line can be a command, as it would be typed in at the terminal. Or it can be a script-specific structure, such as a subroutine *<sub> .... 1 or more commands .... <endsub>*.

Script tokens in a line are separated by whitespace, which is one or more <Spc> or <Tab>. Leading and trailing whitespace are ignored. Indentation can be added for clarity but, like 'C' it is ignored.

5.3 Running scripts

Run a script by launching it from the terminal

```
> script 1 run <Enter>
```

The script will run. It may print text to the terminal. When it's done it will display.

```
> End of script
```

If the script engine can't parse a script line it will usually print a message and continue to the next line.

The user can intervene; ***Crtl-Q*** quits a script, ***Ctrl-P*** pauses, ***Ctrl-R*** resumes after a pause.

Boot Script (TinyLib 3/12/07 onwards)

If there is text in File 0, then the script engine will run that (script) automatically when the application starts up.

Switching scripts inline

Scripts may be 'called' from other scripts by using “script nn run” in the caller's text. The application leaves the current script and executes from the start of the new one. This is not a 'function call' There is no return to the original script. Since there's no nesting, there's no limit on the number of scripts which can be chained in this way, and scripts may call themsleves; it's the same a starting the script over.

5.4 Reserved Words

Apart from the commands to control T1 objects, such as *sensor* and *actuator*. scripts have their reserved words:

Timing	- wait, waitfor, upto, quitIfTimeout, elapsed
Time units	- min mins, minute, minutes, hr, hrs, hour, hours
Loops	- repeat, endrepeat, until, while, wend, break, continue,
Branches	- if, else, endif, elif
Controls	- pause, quit, doOnQuit, beep, clrfail
Subroutines	- sub, endsub, call, return
Output	- print, eval
Asynchronous tasks	- doat, doafter, repeatIntvl, wait
expression conditions	- lastFailed, anyFailed, noneFailed, true, false

5.5 User commands in scripts

Any user command can be used in a script and will have the same effect there as when entered at the terminal. Each user command returns success (1) or fail (0). The result controls the ***lastFailed*** and ***anyFailed*** flags in the script engine. These flags allow scripts to branch on the results from user commands. See 5.6 Conditions and fail flags (below).

Most user commands return 1 = success, unless they are missing arguments or their argument is a non-existent object. Only *eval* (evaluates expressions) and *compare* (compare variables against limits) would usually be used to control to flow of script execution.

5.6 Conditions and fail flags

Some script commands always or may test conditions appended to the command. These are:

while, until, if, elif, quit, continue, break.

The condition may be an expression (see Expressions). or it may be '***lastFailed***', '***anyFailed***' or ***'noneFailed'***. These flags are set from the results of the last User commands executed. '***lastFailed***' is true if the last executed user command failed (returned 0); '***anyFailed***' is true if any user command failed since the ***clrfail*** *command* was executed (The *'**clrfail**' command resets the* '***anyFailed***' *flag.)*

'noneFailed' is NOT(***'anyFailed'***).

For ***quit**, **break*** and ***continue*** a condition is optional. If its omitted then the result is always true. ***while***, ***until***, ***if***, ***elif*** require a condition.

Examples with expressions are:

```
quit                    ; always quit

quit Pres1 > 80         ; quit if > 80kPa

while SupplyV > 11      ; scan as long as supply is good
   read O2Cells         ; print O2 cell mV
   wait 20              ; wait 20sec
wend

$ test done
```

Examples with compare:

```
clearFail               ; make sure the 'fail' flag is clear

while SupplyV > 11      ; scan as long as supply is good
   compare Pres1 > 80   ; do all these checks
   compare Pres2 > 10.3
   compare Pres5 inside 16.2 24.1
   break anyFailed      ; until any are bad
wend
```

5.7 Timing script actions

There are several ways of timing script actions. See the reference on the individual commands syntax.

a) Using *wait*

```
eval Power = 1      ; power up DUT
wait 20             ; wait 20sec for power to stabilise
read dcOut          ; read test signal
```

b) Using *waitfor*

```
waitfor Tmpr1 > 30  ; wait for DUT to warm up
read Sens1          ; read test signal
```

waitfor can be used with a timeout and a quit

```
waitfor Tmpr1 > 30  upto 1 hr                ; wait 1 hr for DUT to warm up
waitfor Tmpr1 > 30  upto 1 hr quitIfTimeout   ; quit script if timeout
```

d) with e*lapsed*

elapsed is a variable which returns the time since the script started. It can only be accessed from within scripts.

```
Power = on

while elapsed < 2 minutes        ; For the 1st 2 minutes
   PumpDrv = 6.2                 ; run the pump at this level
wend

PumpDrv = 0                      ; Done; set pump off.
beep
$ test done
```

Specifying times

Times may be specified in seconds minutes or hours. If no units are specified then seconds are assumed.

```
wait 20            ; 20 secs
wait 1.2 mins      ; 1.2 minutes
wait 0.6 hr        ; about 35 minutes
```

The are alternatives for time units

```
20 min  20 mins  20 minute 20 minutes   ; all read the same
2 hr  2 hrs 2 hour 2 hours              ; all read the same
```

The timing resolution on script actions is nominally **20msec.** However execution load and delays from other process mean that you should not use scripts to time accurately below **100msec.**

5.8 Independently timed actions

doat, ***doafter*** *or* ***repeatIntvl*** each queue a line of script to execute at or after a specified time. Unlike normal timings, these delayed actions are independent of execution of the remainder of the script (and of each other) The script may continue and may even finish before the actions are executed. Up to **8** of these actions can be pending at any time. Examples:

```
Power = on
doat 5 minutes : < Power = off          ; auto power off after 5 minutes
call TestValves                         ; but meantime do this

Valve1 = on
doafter 2 minutes : < Valve1 = off      ; auto valve off 2 minutes from now
Heaters = 1                             ; but meantime turn on the heaters
waitfor Tmpr2 > 30                      ; wait for them to warm
read Oxygen3                            ; and read an O2 cell
```

See the command reference for more detail on the syntax of these commands.

repeatIntvl repeats a script line a a specified interval

```
repeatIntvl 10 2 mins : read pres6      ; read Pres6 10 times x 2 minutes
```

If it's already time to execute a *doat* when that line is read, then it is done rightway.

doat or ***doafter*** can execute control statements. What happens depends on where the script is when the command is executed.

```
Power = on
doat 2 minutes : quit            ; quit after 2 minutes whatever
... the rest of the test

while SupplV > 10
   Valve1 = on
   doafter 10 : continue         ; back to 'while' 10secs from now
   Valve2 = on                   ; so Valve2 will be on for 10secs
   waitfor                       ; wait here forever
wend
```

The timing resolution on independent actions is **100msec.**

5.9 Subroutines

Scripts can declare and call subroutines e.g

```
sub PrintMsg1             ; first the subroutine
   $ This is msg 1
endsub

$ main              ; now the main code
call PrintMsg1      ; which calls the subroutine
$ done
```

will print:

```
main
This is msg 1
done
```

Subroutines must be declared ahead of the main code. A script can have up to **5** subroutines (if you need more ask).

Subroutines can contain any commands and syntax which are legal in the main code stream, including loops (`while... wend`) and switches (`if... else`)

Subroutines may be nested 1 **deep** i.e a subroutine can call another subroutine but no deeper. Nested routines do not have to be declared in any particular order; as long as they are ahead of the main code body.

return forces and exit from the body of a subroutine e.g

```
sub SwitchValves
   write Valve1 off
   if s1 == on
      return             ; conditional return from here
   endif
   write Valve3 off      ; else fall though and do this before returning
endsub
```

Subroutines names can be up to **20** characters long

Passing parameters

You can pass up to **6** parameters to a subroutine. The parameters are identified by #1, #2, #3, #4, #5, #6

```
sub SetPumpDrive
   write PumpV1 #1
   write PumpFreq #2
   $ Voltage = #1 Freq = #2
endsub
```

is called as:

```
call SetPumpDrive 5.8 400       ; pump drive to 5.8V and 400Hz
```

You don't have to use all the parameters in the argument list when you call a subroutine. Extra ones are ignored.

Note that the parameters are passed as words (text) and so can be printed out in messages. They don't have to be numbers or variables.

The parameter list can't be more than **40** characters long, including spaces. Comments or other text after the 1st 4 words are ignored (and don't count in the 40 character limit).

5.10 Script command listing

Syntax for command format:

(ss)	= alternative text for a command e.g (?) is the same as 'help'
< nn >	= required argument;
[nn]	= optional argument
[nn]... or < nn >...	= one or more repeats

a | b | c	= alternatives, a OR b OR c
'raw'	= literal

< print | $ | $$ > [condition flag] [text to write]

'$' writes text to the console with a newline. '$' is the shorthand e.g

```
$ This line will print as written
$ and this will print on the next line
```

'$$' will print 2 spaces instead of a newline e.g

```
$$ Ambient pressure too low.
read AmbPres
```

will give:

```
Ambient pressure too low.  AmbPres = 85 kPa
```

If no text is supplied then '$' prints just a newline and '$$' prints just spaces.

'$$' keeps printouts from user commands on the current line until canceled by a '$'.

```
$$ Test pressures are
read Pres6
$$ and also
read Pres7
$ which are OK.
```

will print all on one line

```
'Test pressures are Pres6 = 98 kPa and also Pres7 = 94 Kpa which are OK.
```

The last line was a $, so anything after this will print on a new line.

Use '$$' alone to enter and leave single-line printout (and '$' alone to leave) e.g

```
$$
read SupplyV
$ This should be > 10.5V
```

gives:

```
'SupplyV = 11.3V  This should be > 10.5V'
```

If 'condition flag' is present the printout only happens if the condition is true.

```
compare Pres6 inside 60 75      ; 60kPa < Pres6 < 75kPa?
compare Pres7 inside 60 75      ; 60kPa < Pres7 < 75kPa?
$ noneFailed Pres6, Pres7 are OK
```

wait <time to wait>

Wait a specifiied number of seconds, minutes or hours

```
wait 10                         ; 10 secs
wait 10 mins | minutes
wait 2.3 hr                     ; 2hrs 20 minutes
```

beep [num beeps]

Beep a number of times.

eval (<) [expression] - evaluates an expression

This evaluates expressions in the same way as 'eval', except that it does not print the result to the console. E.g

```
< Pump1Drv = 4.5   or
< counter1 = counter1 + 1
```

pause [conditon] ['beep']

Pauses until Ctrl-R is hit (does the same as Ctrl-P). If there's a condition this must be true to pause

```
pause Pres1 < 14.0        ; pause if pres too low
pause SupplyV < 12 beep   ; let user know
```

Note that pause on a condition isn't the same as '*waitfor*'; '*waitfor'* repeatedly checks the condition and continues once its true. 'p*ause'* checks just once and stops if the condition is true.

If 'beep' is in the argument list then will beep 3 times when it pauses.

quit [conditon] / doOnQuit

Exit the script engine, either unconditionally or conditionally e.g

```
quit dcOut > 3.0                ; quits if true
quit                            ; quits unconditionally
```

If 'doOnQuit' starts a line anywhere after the 'quit' then execute from after that line to the end of the script. 'doOnQuit' to user to specify closeout action(s) e.g shutting off a pump, which always get done at the end.

```
Valves = 0x03                   ; Set some valves
read O2Cells                    ; Take readings
quit O2Cells.max > 10.2         ; May quit from here
Valves = 0x1B                   ; or may continue, with a new valve setting
read O2Cells                    ; and take another reading
doOnQuit                        ; Anything after here will always be executed
Valves = 0x00                   ; so valves will always be turned off
```

repeat [count] ..... until [exit condition]

There are two ways to this. Either repeat a fixed number of times:

```
repeat 10
   read detectorAmpltude        ; do this 10 times
   read Pres6                   ; and this too
until
```

or repeat until an exit condition is reached.

```
repeat
   eval PumpV = PumpV + 0.3
until  Pres1 > 90
```

Unlike the while loop; a repeat always executes at least once.

If 'repeat ' and 'until' have no arguments then then the code loops forever. E.g this code will toggle 'valve3' *ad nauseum*.

```
repeat
   write Valve3 0
   wait 10
   write Valve3 1
until
```

while <condition> [statements...]   wend

Evaluate the while, loop as long as it is true

```
while( SupplyV > 10 )
   read  detectorAmpltude
wend
```

<u>Loops cannot be nested</u>; though they can be inside if-else statements

continue [condition]

If inside a loop, continue to the top of the loop. If condition is supplied then continue only if it is true

break [condition]

If inside a repeat or while loop, break to the 1st statement past the loop. If a condition is supplied then break only if it is true

```
while( SupplyV > 10 )
   if  detectorAmpltude < 4.7
      break
   endif
wend
```

This can be shortened to:

```
while( SupplyV > 10 )
   break detectorAmpltude < 4.7
wend
```

if <condition> [statements...] elif <condition> [statements....] else. [statements.....] endif

Execute the statement block defined by the true condition. If more than one condition is true then the 1st true one is executed. Statement blocks can be empty.

```
if v1 > 10
   $ bias voltage too high
   quit
elif v1 < 4
   $ bias too low
   quit
else
   read sensor1
   read sensor2
endif
```

If - else - endif cannot be nested. For example, the following is illegal:

```
if v1 > 10
   if v2 < 5.4                  ; this 'if' is nested
      $ bias voltage too high
      quit
   else
      write Pump1V 6.2
   endif
else
   read sensor1
endif
```

However they can be nested indirectly via a subroutine.

```
sub CheckBias            ; subroutine 'CheckBias ' contains if-endif
   if v2 < 5.4
      $ bias voltage too high
      quit
   else
      write Pump1V 6.2
   endif
endsub

if v1 > 10
   call CheckBias        ; 'CheckBias' may be called from inside if-endif
else
   read sensor2
endif
```

waitfor <condition> ['upto' timeout] [quitIfTimeout]

Wait on a condition with an optional timeout and an optional quit.

```
waitfor Key1 == 1 upto 10       ; up to 10 secs for keypress
waitfor PumpI > 0.3             ; wait as long as necessary
waitfor Key1 == 1 upto 1 minute quitIfTimeout
```

doat / < time > : <action>

Queue a script line to be executed at some some later time. The time specified is from the start of the script. Unlike e.g *waitfor* this is an asynchronous action. The script doesn't wait for the action but continues rightaway. Up to **8** of ***doat, doafter*** or ***repeatIntvl*** can be queued at any time.

```
doat 2 minutes : < Valve3 = 0    ; turn off 2 mins after start of script
$ this line will be executed rightway, no wait
```

doafter < time > : <action>

Same as ***doat*** except that the time is relative to 'now', rather than start of script..

```
doafter 30 : PumpV = 6          ; write PumpV 30 secs from now
$ this line will be executed rightway, no wait
```

repeatIntvl / <repeatCnt> < repeat_interval > ['wait'] : <action>

Queue a script line to be executed '*repeatCnt*' times with '*repeat_interval* '. The first execution is '*repeat_interval*' after the script line is run.

Like doat and doafter this is by default an asynchronous action. The script doesn't wait for the action but continues rightaway. If however 'wait' is specified then the script engine doesn't continue but pauses until all repeats have been executed.

This command is useful for taking repeated readings of a variable. Here we don't wait for some readings to complete before continuing with the script

```
repeatIntvl 5 2.0 mins : < read pres6         ; Read 6 times x 2 minutes
$ this line will be executed rightway, no wait
```

This script waits for the readings to complete.

```
repeatIntvl 5 2.0 mins wait : < read pres6    ; Read 6 times x 2 minutes
$ this line is not excuted until all reads are done
```

sub <subroutine name> [body to execute] endsub

Declares a subroutine.

```
sub routine1
   $ 1st line to execute
   $ 2nd line
   $ etc...
endsub
```

See the 'Subroutines' section for details

call <subroutine name>

Calls a subroutine.

return

returns from within the body of a subroutine

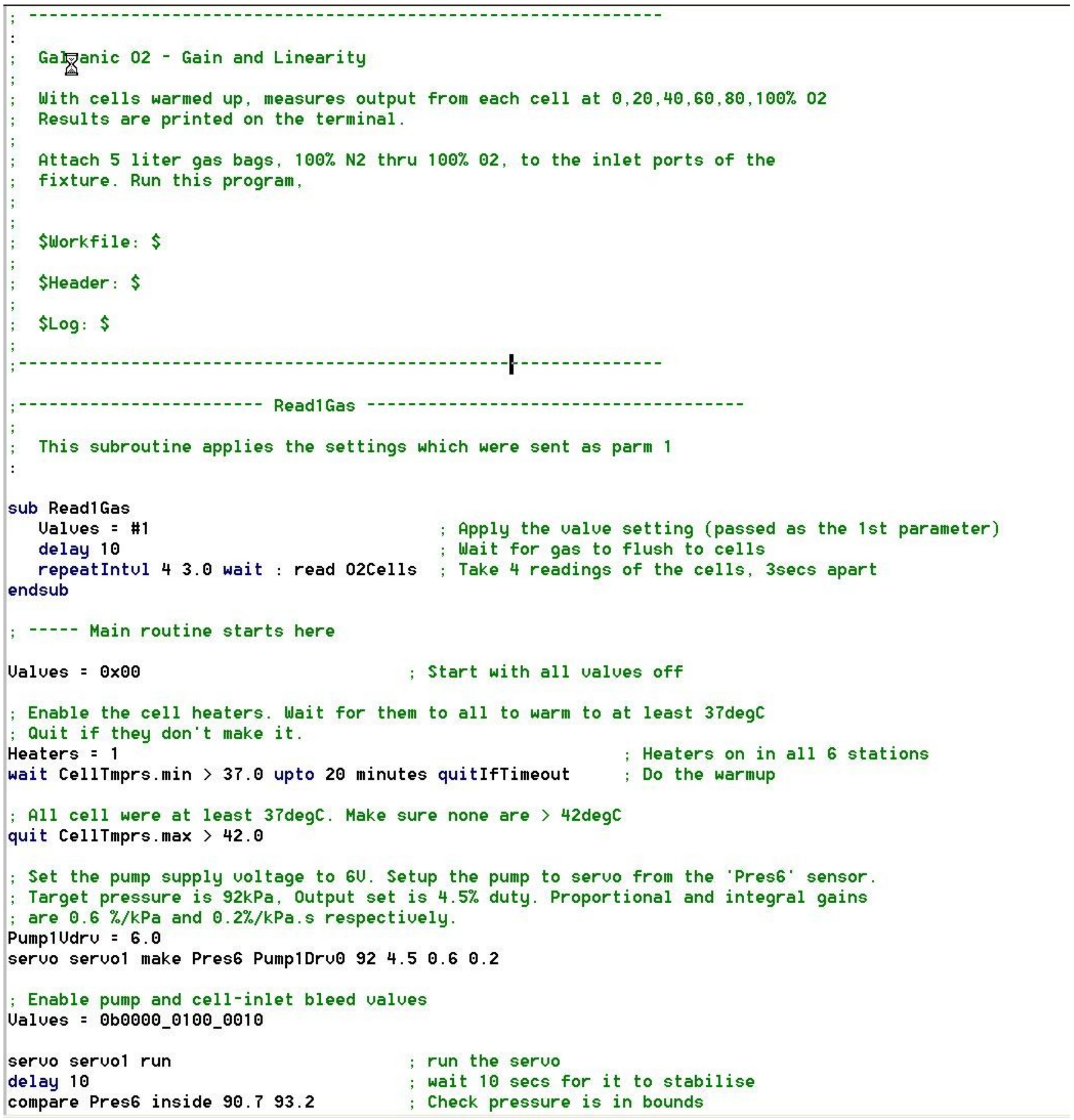
6 Script File Format

Below is an example of a script file. Recommended file conventions and formats are:

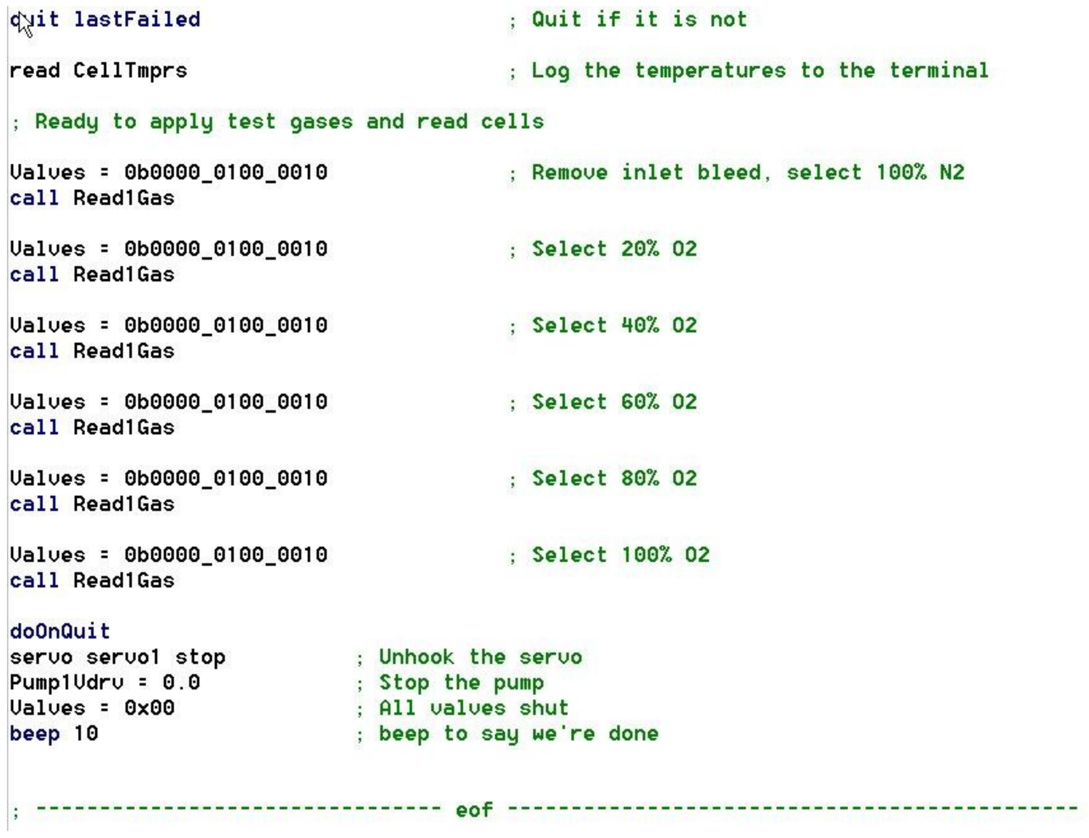
- Filenames extension to be '.t1s' (Tiny1 script). This will be correctly colored by Codewright (if the Chromacoding lexer is set up).

- Indentation with spaces only. **<u>Do not use tabs</u>**. Indent <u>3 spaces</u> per level.

  (Right now T1 may not parse through tabs correctly all the time)
- Line delimiter is CRLF, not LF alone.
- Put the filename, description, version and date as a T1 quote as the 1st script line e.g
  `$ Galvanic O2 Gain Tester rev 2.0 2/3/10`
  This will retained when the script is read into Flash and will be quoted by '*`txtfile report`*' which shows the 1st line of each file.
- Add '----------------------------- eof ------------------------------ at the end of the page
- Use headings in front of any subroutines and the main routine
- Add the Serena log tags. If the file becomes a production it will be under version control.



```
; ---------------------------------------------------------------
;
;  Galvanic O2 - Gain and Linearity
;
;  With cells warmed up, measures output from each cell at 0,20,40,60,80,100% O2
;  Results are printed on the terminal.
;
;  Attach 5 liter gas bags, 100% N2 thru 100% O2, to the inlet ports of the
;  fixture. Run this program,
;
;
;  $Workfile: $
;
;  $Header: $
;
;  $Log: $
;
;----------------------------------------------------------------

;------------------------ Read1Gas ------------------------------------
;
;  This subroutine applies the settings which were sent as parm 1
;

sub Read1Gas
   Valves = #1                           ; Apply the valve setting (passed as the 1st parameter)
   delay 10                              ; Wait for gas to flush to cells
   repeatIntvl 4 3.0 wait : read O2Cells  ; Take 4 readings of the cells, 3secs apart
endsub

; ----- Main routine starts here

Valves = 0x00                         ; Start with all valves off

; Enable the cell heaters. Wait for them to all to warm to at least 37degC
; Quit if they don't make it.
Heaters = 1                                                ; Heaters on in all 6 stations
wait CellTmprs.min > 37.0 upto 20 minutes quitIfTimeout     ; Do the warmup

; All cell were at least 37degC. Make sure none are > 42degC
quit CellTmprs.max > 42.0

; Set the pump supply voltage to 6V. Setup the pump to servo from the 'Pres6' sensor.
; Target pressure is 92kPa, Output set is 4.5% duty. Proportional and integral gains
; are 0.6 %/kPa and 0.2%/kPa.s respectively.
Pump1Vdrv = 6.0
servo servo1 make Pres6 Pump1Drv0 92 4.5 0.6 0.2

; Enable pump and cell-inlet bleed valves
Valves = 0b0000_0100_0010

servo servo1 run                      ; run the servo
delay 10                              ; wait 10 secs for it to stabilise
compare Pres6 inside 90.7 93.2        ; Check pressure is in bounds
```



```
quit lastFailed                        ; Quit if it is not

read CellTmprs                         ; Log the temperatures to the terminal

; Ready to apply test gases and read cells

Valves = 0b0000_0100_0010              ; Remove inlet bleed, select 100% N2
call Read1Gas

Valves = 0b0000_0100_0010              ; Select 20% O2
call Read1Gas

Valves = 0b0000_0100_0010              ; Select 40% O2
call Read1Gas

Valves = 0b0000_0100_0010              ; Select 60% O2
call Read1Gas

Valves = 0b0000_0100_0010              ; Select 80% O2
call Read1Gas

Valves = 0b0000_0100_0010              ; Select 100% O2
call Read1Gas

doOnQuit
servo servo1 stop           ; Unhook the servo
Pump1Vdrv = 0.0             ; Stop the pump
Valves = 0x00               ; All valves shut
beep 10                     ; beep to say we're done


; -------------------------------- eof ---------------------------------------------
```

End of File