RP Metrix LASOM Laboratory Apparatus Scalable Olfactometer Module
Behavioral Experiment Sequencer Operation

1. Introduction and Sequencer Theory

Let's start discussing LASOM sequencer programs with an example. In later sections of this document, the various program elements will be formally defined. For now, don't worry about the precise rules.

```
;[Example 1]
Wait 800
Odor 1,3
Wait 800
Gate 1,14
Wait 800
Dummy 1
UnGate 1,14
Idle
```

In this example, we wait 3 times for 800 milliseconds each time.
In between waits:
we first turn on odor valve 3 on slave olfactometer 1
we then turn on gate valve 14 on slave 1
we then turn both valves off and stop.

Each step has an action word, like 'Wait', and up to 2 parameters. An unused parameter is ignored.

We can do this without labeling the steps, since the wait loops either stall the sequencer or let it continue. But the steps are actually numbered. Soon, you will see that you do not need to worry about the step numbers, but for now let's show them explicitly in example 2:

```
;       [Example 2]
1       Wait    800
2       Odor    1,3
3       Wait    800
4       Gate    1,14
5       Wait    800
6       Dummy   1
7       UnGate  1,14
8       Idle
```

Now let's do something more complicated. Instead of waiting unconditionally, let's wait for some time, but check for an input as we wait. If we see the input, we will branch out of the wait. Now we need the step numbers, as a way to say where to go if we take the branch. We will show step numbers in example 3:

```
;       [Example 3]
1       While           800
2       IfIn            $Beam1,6
3       EndWhile
4       Odor            1,4
5       Idle
6       Odor            1,3
7       Idle
```

In step 2, we read the digital inputs and look at the Beam 1 input. If it is active (the beam is broken), we jump to step 6 were we output odor 3 on slave 1, then stop (with the odor still active). If the beam stays inactive, the time expires and we reach the next step, step 4. So we output odor 4 instead, and stop.

Suppose we want to check two inputs in the loop. We can add a second test and branch, as in this example:

```
;       [Example 4]
1       While       800
2       IfIn        $Beam1,7
3       IfIn        $Beam2,9
4       EndWhile
5       Odor        1,4
6       Idle
7       Odor        1,3
8       Idle
9       Odor        1,5
10      Idle
```

The new step 3 checks the state of Beam 2. If active, we go to step 9. Since we added a step, we had to change the branch target in step 2 to be 7, which is annoying. This problem will go away.

Suppose we want to check if an input is NOT active. See example 5:

```
;       [Example 5]
1       While       800
2       IfIn        $Beam1,7
3       IfNotIn     $DigIn3,9
4       EndWhile
5       Odor        1,4
6       Idle
7       Odor        1,3
8       Idle
9       Odor        1,5
10      Idle
```

You will not see any step numbers when you actually prepare a sequencer program. Instead, you will label important points in the program. Where you need to jump out of a While loop, a label will be used to designate the target step. So let's get rid of the step numbers, and show the same program with labels:

```
; [Example 6]
While       800
IfIn        $Beam1, @Beam1Broken
IfNotIn     $DigIn3, @DigIn3WentAway
EndWhile
Odor        1,4
Idle
Label       @Beam1Broken
Odor        1,3
Idle
Label       @DigIn3WentAway
Odor        1,5
Idle
```


What if we want to change the time delay of the While loop each time we do a trial? We will use a named parameter with a default value (for testing the program). Before we run the program, we can change the parameter value. This is shown in example 7:

```
; [Example 7]
Param       WaitTime1,800
While       WaitTime1
IfIn        $Beam1, @Beam1Broken
IfNotIn     $DigIn3, @DigIn3WentAway
EndWhile
Odor        1,4
Idle
Label       @Beam1Broken
Odor        1,3
Idle
Label       @DigIn3WentAway
Odor        1,5
Idle
```

Now when we want to use this sequencer program in the host, we can specify an override value for "WaitTime1" before we run the program. How that actually happens depends on whether we are using MatLab, Python, LabView, or some other programming environment in the host computer. For each environment, the same things have to happen:

Clear previous settings.
Set any override parameters.
Read the program file, parse it, and compile it.
Load and run the program.
Check results.

Before we talk about how to 'Check results', we can see at least see an example of the preceding steps. Suppose the program from the example above is stored in a LASOM Sequencer (LSQ) source file named "Example7.lsq". In MatLab, we could open a connection to the LASOM board with these statements:

```
h2 = actxcontrol('LASOMX.LASOMXCtrl.1')
invoke(h2, 'DevOpen', 0, 1)
invoke(h2, 'GetLastError')
invoke(h2, 'GetID')
```

Now lets run one trial with the program from the example above, but use 400 milliseconds for the while loop time. The MatLab statements would be:

```
invoke(h2, 'ClearSequence')
invoke(h2, 'SetParamValue', 'WaitTime1', 400)
invoke(h2, 'ParseSeqFile', 'Example7.lsq')
invoke(h2, 'CompileSequence')
invoke(h2, 'LoadAndRunSequencer')
```


Of course, we have no idea whether the program actually did anything, except perhaps by watching lights blink on the LASOM1. We need some way to check results. There are several ways to do that. We can modify the program to output hardware signals when important things happen. When the sequencer starts, all digital outputs are turned off. We can selectively turn them on and off within the program. With an external recording device, we can detect that the sequencer did various things. Example 8 shows that method:

```
; [Example 8]
Param       WaitTime1,800
While       WaitTime1
IfIn        $Beam1, @Beam1Broken
IfNotIn     $DigIn3, @DigIn3WentAway
EndWhile
Odor        1,4
Output      $XlogicOut1, 1
Idle
;
Label       @Beam1Broken
Odor        1,3
Output      $XlogicOut2, 1
Idle
;
Label       @DigIn3WentAway
Odor        1,5
Output      $XlogicOut3, 1
Idle
```

Alternatively, we can use sequencer variables to track progress. We can set and get the variables from the host. If we set a value within the sequencer program, we can check the value from the host. The variables start out with a value of zero. Example 9 shows a sequencer program that changes a variable value based on events.

```
; [Example 9]
Param       WaitTime1,800
While       WaitTime1
IfIn        $Beam1, @Beam1Broken
IfNotIn     $DigIn3, @DigIn3WentAway
EndWhile
Odor        1,4
SetVar      $Var1,1
Idle
;
Label       @Beam1Broken
Odor        1,3
SetVar      $Var1,2
Idle
;
Label       @DigIn3WentAway
Odor        1,5
SetVar      $Var1,3
Idle
```

Now a MatLab host program can check for progress by getting the current value of the variable, with the statement:

```
get(h2, 'SequencerVar', 1)
```


Another method to check for sequencer progress is to monitor the current sequencer step from a host program, and check whether the sequencer has halted.

Yet another method to check results is to obtain the sequencer execution history. Each time the sequencer executes a step within the program, it records a time value from a free running timer. After the sequencer has halted, the host can obtain the timer value for each sequencer step. Steps in a while loop may be executed many times, and the timer values are overwritten each time around the loop. The final timer value recorded for each step will tell us when the sequencer broke out of the loop, perhaps in response to an input change.

These methods will be described later within this document.

This ends the informal discussion of the sequencer. The next sections will present a formal description of the sequencer capabilities and how to prepare a sequencer program file.

2. Sequencer Capabilities

The LASOM Sequencer executes as a sub process within the USB controller on the master olfactometer module. The Sequencer has access to the olfactometer slave section of the master module, and to any additional slave olfactometer modules attached to the master via the LASOM expansion bus. The Sequencer also has access to the master module digital inputs and outputs, including two Leaf connectors, a logic expansion connector, and input and output strobe connectors.

The master module USB controller executes the sequencer one step at a time from its main polling loop. This give the sequencer a chance to take execute a step about every 20 microseconds. The USB controller firmware also maintains an interrupt driven free running time counter based on the processor crystal oscillator. This counter increments at 4 MHz, so that time measurements have a precision of 250 ns. Although the start time and duration of a sequencer step is not precisely controlled, the relative timing between steps can be precisely measured. Relative to the start of the sequencer program, events can be scheduled to a precision of 1 ms or better. The time counter wraps every 65.536 seconds.

The sequencer program can consist of up to 128 steps.

The sequencer can save the free running counter value in up to 4 time latch variables, which permits the sequencer to compute the time delay relative to any of these latched times. The first of these is used to implement waits and while loop timeouts.

The sequencer maintains up to 4 variable locations. The sequencer can write to these variables with constants, or with the state of one of the digital inputs. The host can read these variables while the sequencer is running, to observe the state of the sequencer or external inputs. The host can also write to these variables. The sequencer can drive a digital output based on a variable, or break out of a while loop.

Each time the sequencer gets control from the USB controller, it processes the current step. A step consists of an operation code (op code) and two optional 8-bit parameters. Usually, the sequencer performs the operation associated with the current step, using the parameters if applicable, and then returns control to the USB firmware main process. A few op codes consume more than one step before returning.

In general, one statement line of a sequencer program source file will become one sequencer step in the memory of the USB controller. Some statements require multiple steps to implement. The processes of parsing a sequencer source file, compiling it, generating the list of steps and parameters, and loading this list into the LASOM microprocessor hide the details of the actual step op codes and parameters from the user. The user only needs to know how to prepare a sequencer program source file – the next topic.

3. **Sequencer Programming**

A sequencer program consists of a list of one-line statements and comments.

Lines starting with a semicolon (;) are ignored. These are comment lines.

A program statement begins with one of the action words listed below. Depending on the action, one or two parameters follow on the same line, separated by white space or commas (,). These can be followed by a comment.

These are the possible action words:

Action	Param1	Param2	Description
Idle	<none>	<none>	Stop the sequencer.
Wait	DefinedVal	<none>	Wait unconditionally for P1 milliseconds.
Odor	Slave	OdorValve	On slave P1, open only odor valve P2. Close other odor valves and the dummy valve.
Dummy	Slave	<none>	On slave P1, open only the dummy valve. Close all odor valves.
UnGate	Slave	GateValve	On Slave P1, open gate valve P2.
Gate	Slave	GateValve	On Slave P1, close gate valve P2.
Select	Slave	GateValve	(TBD) On Slave P1, select gate valve P2 for later activity.
Arm	Slave	<none>	(TBD) On Slave P1, enable selected gate valves to open on an active strobe signal.
Strobe	Mask	<none>	Configure the strobe signal, including the software generated strobe signal.
Param	DataSym	DefinedVal	Parameter P1 is assigned the value P2, which can be a number, predefined symbol, or other parameter.
Label	JumpSym	<none>	Parameter P1 is assigned the current step position, for use as a jump target.
Monitor	InIndex	OutIndex	(TBD) Continually route the P1 input state to the P2 output.
EndGen	<none>	<none>	Mark the end of a conditional code generation block. Generate copies.
IfGen	DefinedVal	<none>	Begin a conditional code generation block, 1 copy will be generated.
LoopGen	DefinedVal	<none>	Begin a conditional code generation block, P1 copies will be generated.
Output	OutIndex	DefinedVal	Output P1 if the value of P2 is non-zero.
EndWhile	<none>	<none>	Mark the end a WHILE block. Loop back to the beginning of the block.
WhileAlways	<none>	<none>	Begin an unconditional WHILE loop block.
While	DefinedVal	<none>	Begin a timed WHILE loop block, which will terminate after P1 milliseconds.
IfNotIn	InIndex	JumpSym	Get state of input P1. Jump to label P2 if not asserted.
IfIn	InIndex	JumpSym	Get state of input P1. Jump to label P2 if asserted.
SetVar	VarRef	ValueRef	Set the variable P1 to the value P2.
CompareVar	VarRef	ValueRef	Arithmetically compare variable P1 to the value P2. Save result for later jumps.
AddVar	VarRef	ValueRef	To variable P1, add the value of P2.
SubVar	VarRef	ValueRef	To variable P1, subtract the value of P2.
OrVar	VarRef	ValueRef	To variable P1, bitwise logically or with the value of P2.

AndVar	VarRef	ValueRef	To variable P1, bitwise logically and with the value of P2.
NotVar	VarRef	ValueRef	To variable P1, set to the bitwise logical complement of P2.
IfVar	VarRef	JumpSym	If variable P1 is non-zero, jump to label P2.
InVar	InIndex	VarRef	Get state of input P1, set variable P2 to 1 if asserted, else 0.
VarOut	VarRef	OutIndex	If variable P1 is non-zero, assert output P2, else deassert output P2.
StartTimer	TimerIndex	<none>	Set timer P1 to the current absolute time, so the relative time value is zero.
SinceTimer	TimerIndex	TimerIndex	Set timer P2 to the relative time value of timer P1.
ClearLapse	TimerIndex	<none>	Set timer P1 to zero. It will be used to accumulate relative times.
Clear	Slave	<none>	(TBD) On slave P1, clear all gate selections.
AddLapse	TimerIndex	TimerIndex	To timer P2, add the relative time value of timer P1. Restart P1.
CheckLapse	TimerIndex	DefinedVal	Compare the value of timer P1 to the value P2 milliseconds. Save result.
IfLapse	TimerIndex	JumpSym	If CheckLapse result for timer P1 indicated accumulate time exceeded limit, jump to label P2.
StartDelay	TimerIndex	DefinedVal	Start timer P1, remain active until relative time value exceeds P2 milliseconds.
StopDelay	TimerIndex	<none>	Stop timer P2, make it inactive. Execute programmed actions.
IfDelay	TimerIndex	JumpSym	If timer P1 is still active, jump to label P2.
IfNotDelay	TimerIndex	JumpSym	If timer P1 is no longer active, jump to label P2.
PulseOut	TimerIndex	OutIndex	Program timer P1 to assert output P2 while the timer is active.
GoTo	JumpSym	<none>	Jump to label P1.
Call	FuncSym	<none>	Call the routine P1.
Routine	FuncSym	<none>	Parameter P1 is assigned the current step position, for use as a call target.
Return	<none>	<none>	Return to the step position following the call which entered the current routine.
OnExpiry	TimerIndex	FuncSym	Program timer P1 to call routine P2 when the timer becomes inactive.
IfWasEQ	JumpSym	<none>	Jump to label P1 if previous compared values were equal.
IfWasNE	JumpSym	<none>	Jump to label P1 if previous compared values were not equal.
IfWasGT	JumpSym	<none>	Jump to label P1 if in previous compare value 1 was greater.
IfWasGE	JumpSym	<none>	Jump to label P1 if in previous compare value 1 was greater or equal.
IfWasLT	JumpSym	<none>	Jump to label P1 if in previous compare value 1 was less.
IfWasLE	JumpSym	<none>	Jump to label P1 if in previous compare value 1 was less or equal.
EmitStatus	<none>	<none>	Queue a status message to the USB host.

New action words:

A TX Parameter specifies a timer. Use 2..7. Indices 0 and 1 are reserved for loops.
A DX Parameter also specifies a timer, but used as a delay.

StartTimer	TX	Record current time to TX
SinceTimer	TX,DX	Compute delta time from TX to now, store in DX
ClearLapse	DX	Zero lapse timer DX
AddLapse	TX,DX	Compute delta time from TX to now, add delta to DX, restart TX to now.
CheckLapse	DX,xmilliseconds	Compare accumulated time DX to limit xmilliseconds, save result
IfLapse	DX,label	Based on check result, go to label if limit met or exceeded
StartDelay	DX,xmilliseconds	Record current time to DX, record delay length xmilliseconds
StopDelay	DX	Stop delay if active, as if delay length was reached
IfDelay	DX,label	If delay still active (limit not reached), go to label
InNotDelay	DX,label	If delay no longer active, go to label
PulseOut	DX,output	Enable output bit during active delay. Turns off automatically.

The API function GetSequencerTimerValue(UINT uIndex, float &fSeconds) will return the value for a timer or delay in seconds.

Some scenarios:

Invoke StartTimer for various timers in the sequencer program at various points.
This will record the absolute sequencer time.
Read the final results with GetSequencerTimerValue().
Take differences to measure delays between selected events.

Invoke StartTimer for one timer at the beginning of the sequence.
Invoke SinceTimer for various additional timers in the sequencer program at various points.
Read the delays relative to the first timer using GetSequencerTimerValue().

Invoke ClearLapse for timer/delay T2.
When a mouse poke is first detected, StartTimer T1.
Check for mouse pokes, branching to Poke or NoPoke.
At Poke, call AddLapse T1,T2 to accumulate poke time in T2.
At NoPoke, call StartTimer T1.
Total poke time can be read from host with GetSequencerTimerValue().
To branch based on accumulated poke time, first compare to a limit with CheckLapse T2,xmilliseconds.
Then use IfLapse T2,enoughPoke to branch away if the limit has been reached.

Invoke StartDelay T1,xmilliseconds to start a delay timer, set to expire in xmilliseconds.
Invoke PulseOut T1,xout to turn on output xout while the delay is running.
Invoke StopDelay T1 to stop the delay immediately and turn off the output.
Use IfDelay and IfNotDelay to branch, depending on whether the delay is still active or not.

Version 3 functions:

WhileAlways		Loop to EndWhile forever.
SetVar	V1,Q1	Set variable V1 to Q1
AddVar	V1,Q1	Set V1 to V1 + Q1
SubVar	V1,Q1	Set V1 to V1 – Q1
AndVar	V1,Q1	Set V1 to bitwise logical AND of V1,Q1
OrVar	V1,Q1	Set V1 to bitwise logical OR of V1,Q1
NotVar	V1,Q1	Set V1 to bitwise logical NOT of Q1
CompareVar	V1,Q1	Compare V1 to Q1. Result is GE if V1 >= Q1.
IfWasEQ	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 == Q1
IfWasNE	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 != Q1
IfWasGT	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 > Q1
IfWasGE	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 >= Q1
IfWasLT	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 < Q1
IfWasLE	B1	Jump to B1 based on compare result, e.g. if CompareVar result was V1 <= Q1
GoTo	B1	Jump to B1
Call	R1	Call routine R1, return to next statement
OnExpiry	D1,R1	On normal expiry of delay (or StopDelay), call routine R1
Routine	R1	Begin statements for routine R1
Return		End of statements for current routine. Jump here if multiple returns are needed.

Notes:
V1 selects a $Var and is a $Var1..$VarN variable reference, or a number in 1..N
Q1 refers the value of a $Var, or a constant, it is a $Var1..$VarN variable reference, or a number
N=8 in the current version
WhileAlways loops can be nested
Routines can call other routines
OnExipry should be set before starting the delay timer. One execution per start of timer occurs.

