AN<nnnnn>

FSM Designer for the State Configurable Timer

Rev. 1.01 — 25 October 2010 **Application note**

Document information

Info	Content
Keywords	Microcontroller, LPC, State Configurable Timer, SCT, FSM Designer
Abstract	Graphical design tool for the State Configurable Timer in LPC microcontrollers. Includes a C code generator.

founded by Philips

Revision history

Rev	Date	Description
00.01	20100702	First draft
00.02	20100729	Second draft :-)
00.03	20100907	#3
00.04	20101014	Fourth revision, for pre-release
1.01	20101025	Release version

Contact information

For additional information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

The State Configurable Timer is a powerful 32-bit Timer/Counter combined with a configurable state-machine. Inputs, outputs and timer values can be evaluated and trigger events, which in turn can influence outputs or modify an internal state value. Events can also stop and start a timer, reset its counter value to zero, halt a timer, capture its current value, generate an interrupt to the processor or a dma request.

With an increasing number of I/O’s, events, and states, the task of setting up the configuration registers can become quite daunting. Programming at register and bit level alone becomes a challenge, and making quick modifications to an existing design can get tedious and error-prone.

This is where the FSM Designer tool simplifies the programmer´s task. It allows drawing a graphical description of a state machine, and provides a tool to translate it into C code in a transparent way to the user.

A seamless integration into standard coding tools like Keil uVision is also provided with the examples.

1.1 Design Flow

Instead of directly programming the SCT configuration as C code, its function is defined by a graphical diagram including the state machine and its associated events.

The states are represented by bubbles, and the transitions between the bubbles represent events. Under each event (arrow transition) effects on the outputs, the timers themselves and the rest of the system can be specified. The diagram includes an attribute table to provide the parameters for the events, describe a list of inputs and outputs, specify the trigger of a DMA or IRQ request, and so on.

The graphical drawing gets added as a custom file to a programmer´s project, just like any other source file.

A custom pre-build step is used to translate the diagram directly into a C source code, which gets then compiled and linked with the rest of the user´s application. No manual action is necessary for this extra step as it is integrated into the project build rules.

1.2 Advantages of an automated process

Using a graphical tool to draw the state machine diagram allows the programmer to rapidly prototype and test an application concept, skipping the step of mapping the idea into the peripheral´s register implementation by translate a configuration directly.

This can be done in a very natural and intuitive way, without having to get familiar with all the details of the peripheral setup, since the tool automatically generates the complete configuration. This hugely reduces the debug efforts and saves you from time consuming manual checks, especially during the initial phases where the design might be subject to a lot of changes.

1.3 Advanced features

Within the tool it is possible to specify an order of precedence (priority) in case there are multiple events which might happen within the same state, in order to transition to the desired destination state. The selection of the event numbering assignment is done automatically by the tool (since events with higher hardware index number take precedence over events with lower index).

Generally speaking, events are the most valuable resource within the SCT.
An optimal usage of them is a key aspect of the synthesis process, as minimizing the number of used events allows adding more functionality within an SCT configuration.
At the same time, this is strongly linked with determining a clever numbering scheme for the states, which can allow reusing one same event in multiples states of the diagram.

Besides from very simple configurations, this is not easy to determine manually. For this reason an optimizing algorithm is integrated in the tool which tries to synthesize the state machine configuration using the minimal amount of resources possible.
A side effect of this is that you do not have to take care of optimizing the state machine diagram from the very beginning, in case you need to use lot of resources (define a lot of events).

2. Installation

2.1 Installation steps

For the installation of the required sw, please refer to the installation guide fsmdesigner-install-guide.pdf provided with the software package

2.2 Integration in a Keil uVision project

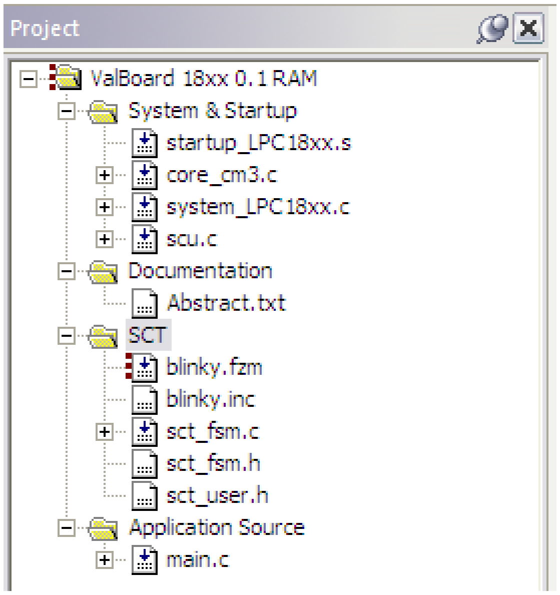
Take a note of the directory where the tools were installed as for section 2.1

You will need to add three files to the project: *blinky.fzm*, *blinky.inc*, and *sct_user.h*. A walkthrough for each of them follows.

First add your state machine drawing (FZM file, here: *blinky.fzm*) to your project.

The proposal shown in the next figure uses a group SCT which is placed before all other application files.

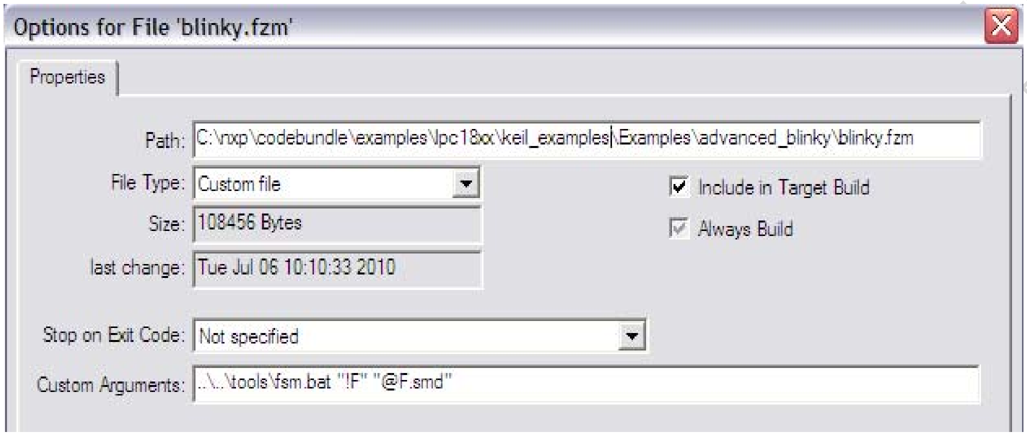
This ensures that the automatic build process can do everything in one pass only as the state machine configuration files will be created before the compile and link steps are performed.



```
ValBoard 18xx 0.1 RAM
    System & Startup
        startup_LPC18xx.s
        core_cm3.c
        system_LPC18xx.c
        scu.c
    Documentation
        Abstract.txt
    SCT
        blinky.fzm
        blinky.inc
        sct_fsm.c
        sct_fsm.h
        sct_user.h
    Application Source
        main.c
```

Fig 1. SCT project list in Blinky example project

In the local options for the .fzm file, select file type “Custom file”, and for the build process enter a command line like: `..\fsm_tools\fsm.bat "!F" "@F.smd"`



Options for File 'blinky.fzm'

Path: C:\nxp\codebundle\examples\lpc18xx\keil_examples\Examples\advanced_blinky\blinky.fzm

File Type: Custom file

Size: 108456 Bytes

last change: Tue Jul 06 10:10:33 2010

☑ Include in Target Build

☑ Always Build

Stop on Exit Code: Not specified

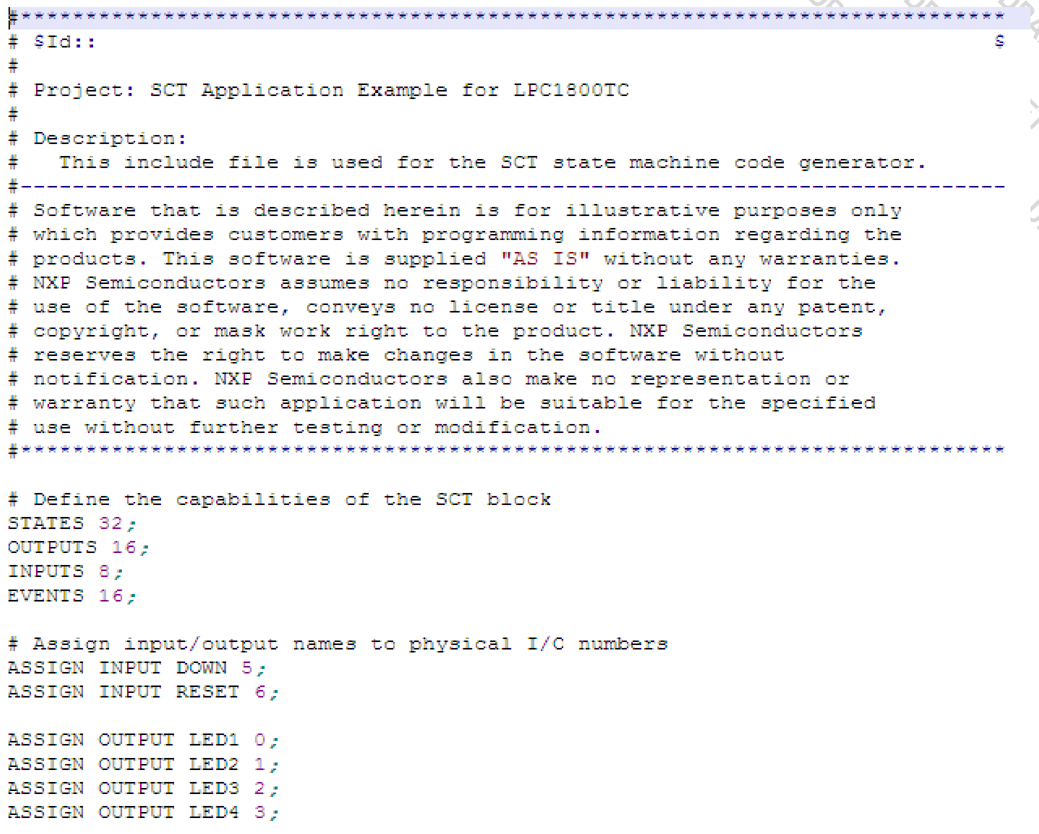
Custom Arguments: ..\..\tools\fsm.bat "!F" "@F.smd"

Fig 2. Build settings for FZM files in Keil uVision

The fsm.bat path could need to be updated for the current location of your tools install you have performed in step 2.1. Note that within the delivered example projects, the build tools are also available within the \tools subdirectory - for convenience, so that you should always be able to successfully rebuild the examples.

Now add the *blinky.inc* file as well. Make sure to select “Text Document file” for the file type within the file properties.

An example file is shown in Fig 3 below. This file holds the assignment (mapping) of the symbolic I/O pin names used in the state machine graphical tool to the physical input or output pin number of the UT.

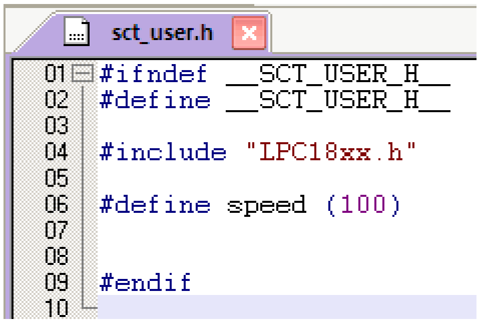


```
#*****************************************************************************
# $Id::                                                                      $
#
# Project: SCT Application Example for LPC1800TC
#
# Description:
#   This include file is used for the SCT state machine code generator.
#-----------------------------------------------------------------------------
# Software that is described herein is for illustrative purposes only
# which provides customers with programming information regarding the
# products. This software is supplied "AS IS" without any warranties.
# NXP Semiconductors assumes no responsibility or liability for the
# use of the software, conveys no license or title under any patent,
# copyright, or mask work right to the product. NXP Semiconductors
# reserves the right to make changes in the software without
# notification. NXP Semiconductors also make no representation or
# warranty that such application will be suitable for the specified
# use without further testing or modification.
#*****************************************************************************

# Define the capabilities of the SCT block
STATES 32;
OUTPUTS 16;
INPUTS 8;
EVENTS 16;

# Assign input/output names to physical I/C numbers
ASSIGN INPUT DOWN 5;
ASSIGN INPUT RESET 6;

ASSIGN OUTPUT LED1 0;
ASSIGN OUTPUT LED2 1;
ASSIGN OUTPUT LED3 2;
ASSIGN OUTPUT LED4 3;
```

Fig 3. I/O assignment file blinky.inc

As last step, you must supply a header file *sct_user.h*. This header file is required to:

- Define all symbolic constant values that have been used as match values definitions within the FZM file, like the match value called “speed” which defines the “maxcount” event in the blinky example.
- Include the appropriate header file including the definition of the peripheral registers (for instance LPC18xx.h).



```
#ifndef __SCT_USER_H__
#define __SCT_USER_H__

#include "LPC18xx.h"

#define speed (100)


#endif

```

Fig 4. sct_user.h file for blinky example

When building the project the included state machine drawing (like ***blinky.fzm***) will be converted into a file called *sct_fsm.c* that is automatically generated within the project directory. Note that the *sct_fsm.c* file name is fixed in the current tool implementation and cannot be user specified.

When building a project from scratch, the state machine configuration file *sct_fsm.c* needs to be generated once (the very first time) in order to be able to include it in the project source list.

For this you can simply hit the “build” button within uVision; it is expected that the whole project build will likely fail at this first run, as we are still missing the initializations and definitions found in *sct_fsm.c* and *sct_fsm.h.* Alternatively, to build only the *sct_fsm.c* manually, you can also right-click on the file within the uVision project list, and choose “translate blinky.fzm”.

After having built it once **-** in either way **-** you can now add *sct_fsm.c* file to your project, and rebuild. After this first time, the whole project will generate automatically the fsm configuration when needed.

Important note: to make sure the latest modifications are always included in the project build, make sure the file *sct_fsm.c* file is listed in the same source group and **after** the FZM file in the project file list (so that the FZM file gets translated first during the build, and the most up-to-date version of sct_fsm.c file gets compiled).

3. Creating and Editing State Machines - tutorial

This chapter gives a short introduction into the usage of the Fizzim state machine editor. It’s particularly important to understand how the SCT features are related to the Fizzim capabilities. All features will be explained with the extensive use of examples.

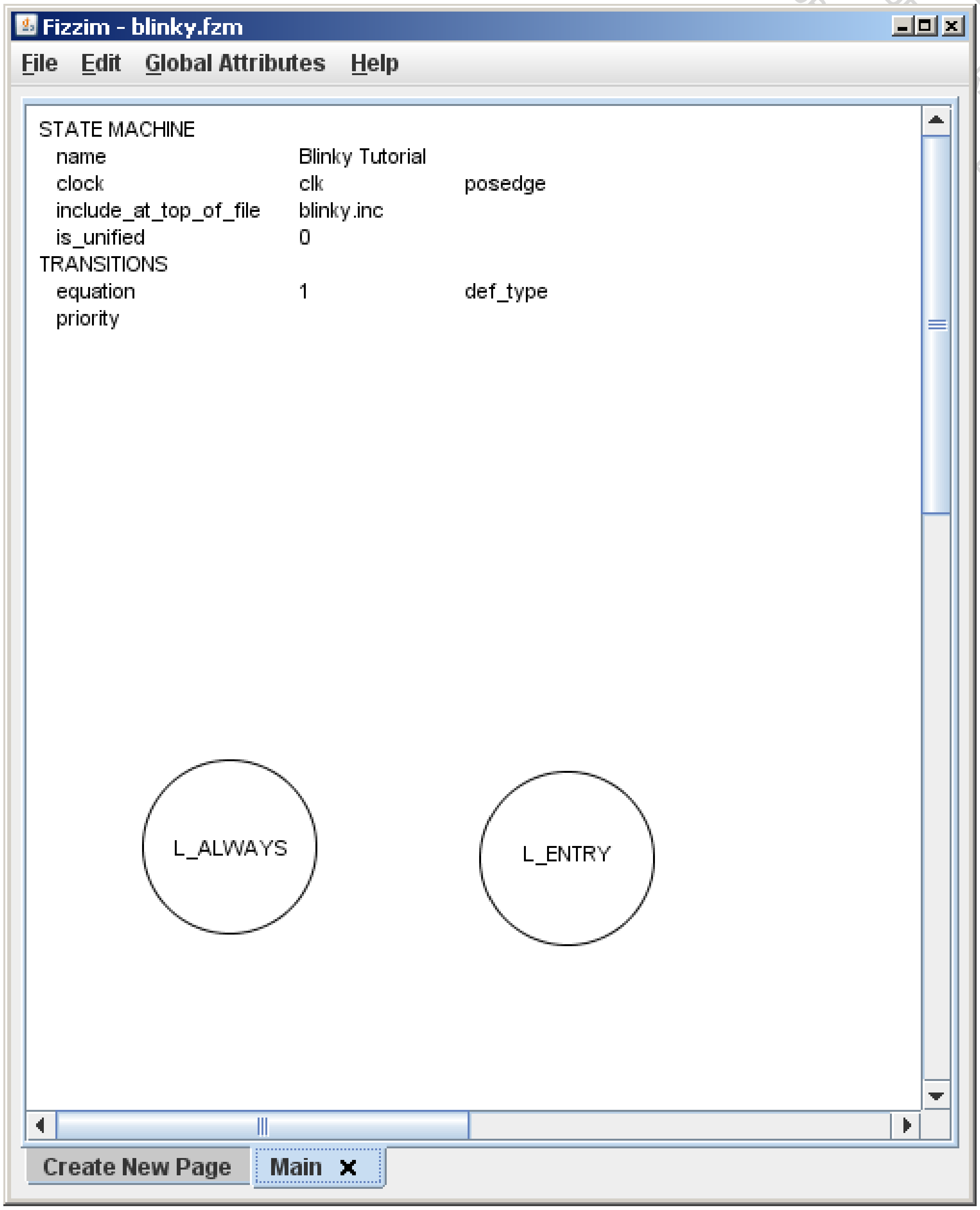
Note that some of the options included in the GUI are not used, but must not be removed for proper functionality of the graphical tool itself, so edit only the described sections.

3.1 The Blinky Project

As you may have guessed, the *blinky* project will demonstrate how you can have a few SCT outputs blink (toggle) under the state machine control.

3.1.1 Start

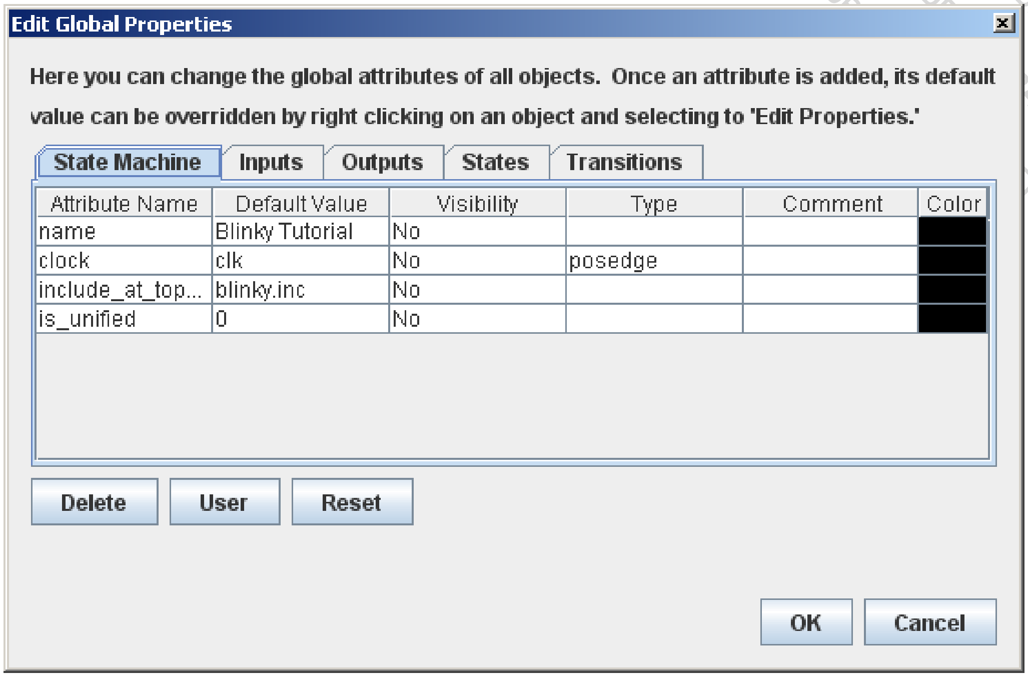
Start by opening the included *blinky* µVision project. Now open the *blinky.fzm* file in Fizzim by double-clicking on it in the Windows file explorer (a copy of this same file can be found as *blinky_start.fzm* within the \solutions folder). This is what you should see:



```
STATE MACHINE
   name                    Blinky Tutorial
   clock                   clk              posedge
   include_at_top_of_file  blinky.inc
   is_unified              0
TRANSITIONS
   equation                1                def_type
   priority
```

Fig 5. Empty blinky.fzm

You can make changes to the project name (just an info text, currently set to Blinky Tutorial) and the name of the included file in the **Global Attributes/State Machine** dialog:



Edit Global Properties

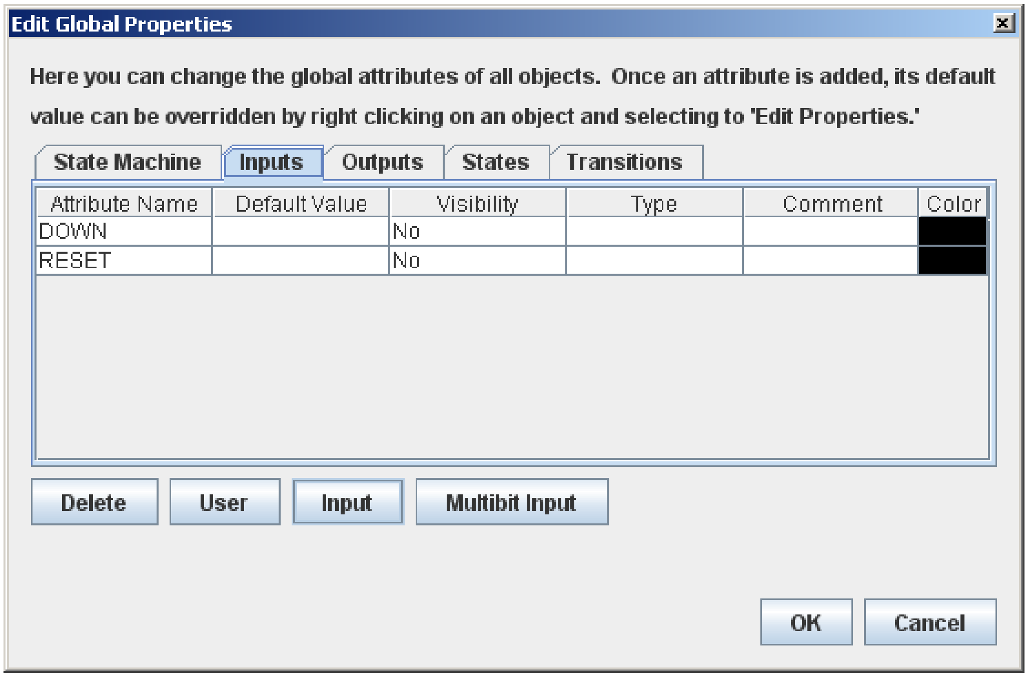
Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
name	Blinky Tutorial	No			■
clock	clk	No	posedge		■
include_at_top...	blinky.inc	No			■
is_unified	0	No			■

Fig 6. Global Attributes/State Machine dialog

Let’s now define the inputs and outputs of the project. We want to have four outputs called LED1, LED2, LED3, and LED4, plus two inputs DOWN and RESET.

Use the **Global Attributes/Inputs** dialog to enter the input names:



Edit Global Properties

Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
DOWN		No			■
RESET		No			■

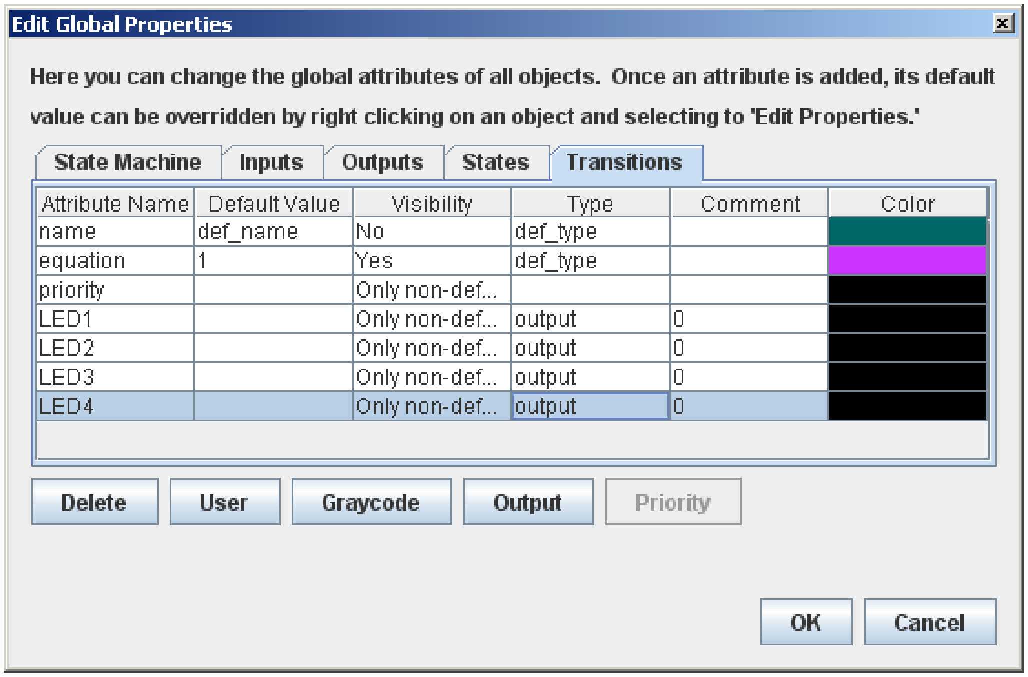
Fig 7. Inputs for Blinky project

Outputs can be defined in the **Global Attributes/Transitions** dialog. Outputs are those entries in this dialog which have the *Type* field set to *output*. Enter the four outputs LED1 to LED4 by clicking on the *Output* button. Make sure the *Visibility* field is set to *Only non-default,* this ensures the outputs are shown in the diagram only when you want to explicitly drive them to a certain value (otherwise they would appear in every transition even when not needed)

Note: do **not** delete the other predefined rows *name* and *equation*, even if those are unused

Every output can be preconfigured to have an initial state which will be pre-set during initialization, before the timer starts. This is useful if you want to have some of the output pins to a specific logic level when your system starts. For this you have to set the *Comment* field of an output to either 1 or 0 if you want it to be initialized, or leave this field blank if you don't want an initialization.

In our example we want all outputs to be explicitly cleared before we start.



Edit Global Properties

Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
name	def_name	No	def_type		🟩
equation	1	Yes	def_type		🟪
priority		Only non-def...			⬛
LED1		Only non-def...	output	0	⬛
LED2		Only non-def...	output	0	⬛
LED3		Only non-def...	output	0	⬛
LED4		Only non-def...	output	0	⬛

Fig 8. Outputs for Blinky project

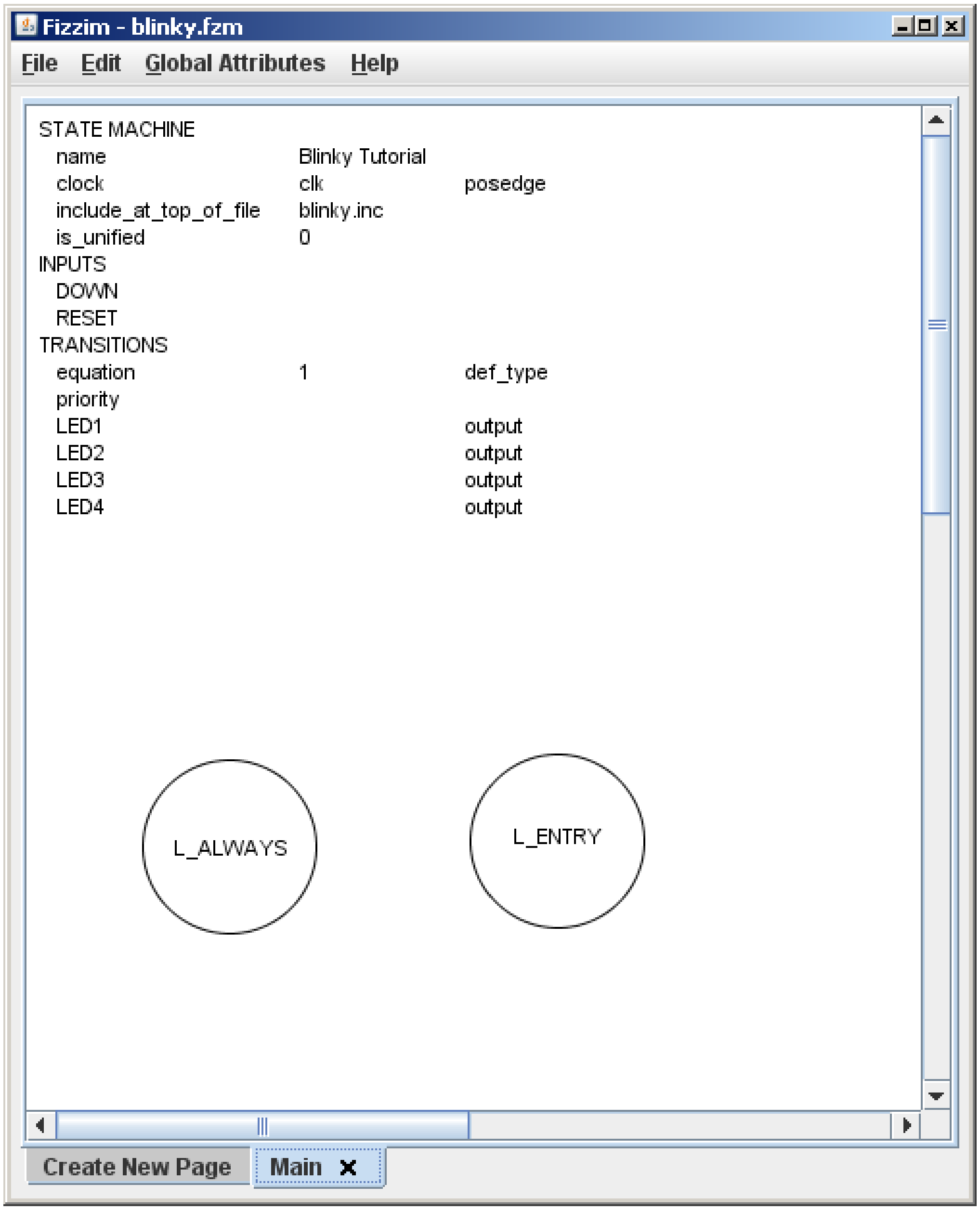
3.1.2 Step 1

Edit the *blinky.inc* file, and make sure it contains I/O pin assignments for all our inputs and outputs (some entries have been commented by purpose, so the project will not build without editing the file – you have to uncomment them).

The number at the end of each assign statement is the physical index of an SCT input or output.

```
# Assign input/output names to physical I/O numbers
ASSIGN INPUT DOWN 5;
ASSIGN INPUT RESET 6;
ASSIGN OUTPUT LED1 0;
ASSIGN OUTPUT LED2 1;
ASSIGN OUTPUT LED3 2;
ASSIGN OUTPUT LED4 3;
```

If everything went right so far, and the FZM file looks similar to what is shown in the next figure, you should be able to build the *Blinky* project in µVision without errors or warnings. A copy of the files is provided as *blinky_step1.fzm* and *blinky_step1.inc* within the \solutions folder



```
STATE MACHINE
  name                      Blinky Tutorial
  clock                     clk               posedge
  include_at_top_of_file    blinky.inc
  is_unified                0
INPUTS
  DOWN
  RESET
TRANSITIONS
  equation                  1                 def_type
  priority
  LED1                                        output
  LED2                                        output
  LED3                                        output
  LED4                                        output
```

Fig 9. blinky.fzm with I/O definitions included

3.1.3 Full version

Before we start with our state machine drawing, let’s describe the expected function:

- Only the L counter shall be used.
- Only one output shall be active at any time.
- Every time there is a timer match, the active output shall advance in a round-robin fashion, switching from one led to the other:

  LED1 --> LED2 --> LED3 --> LED4 --> OFF --> LED1 --> LED2 --> ...

- If the DOWN input is 1, the direction of the output change shall be reversed:

  LED1 --> OFF--> LED4 --> LED3 --> LED2 --> LED1 --> OFF--> LED4 --> ...

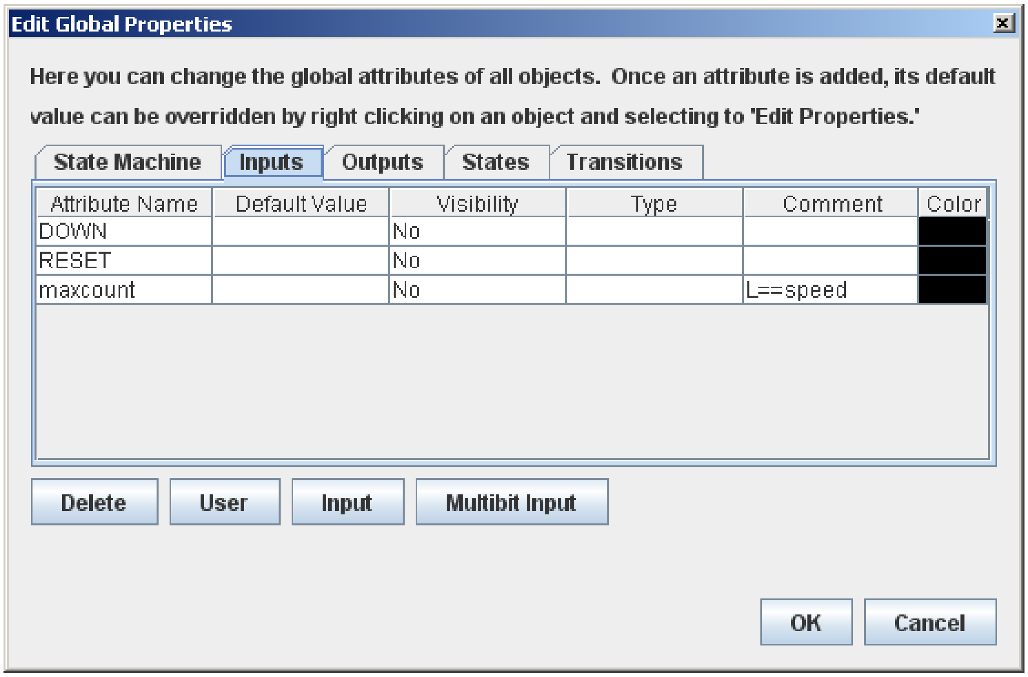
- If the RESET input is high, all outputs shall be off. When it goes low again, the sequence shall start with LED1 (if DOWN=0) or LED4 (if DOWN=1).

Now add a named match definition for the L counter, so that it can be used as a LIMIT event for the counter. A match gets defined in the **Global Attributes/Inputs** dialog, with the match condition indicated in the *Comment* field. Such a condition takes the form *L==value* or *H==value*, depending on which counter you use it.

The *value* field can be any custom text which is copied “as is” to the generated C output file. Therefore be aware that it must be a valid C expression.

The name you choose for *value* needs to be defined in the **sct_user.h** file.

In this case, define a match condition named *maxcount* which takes the value *speed*:



Edit Global Properties

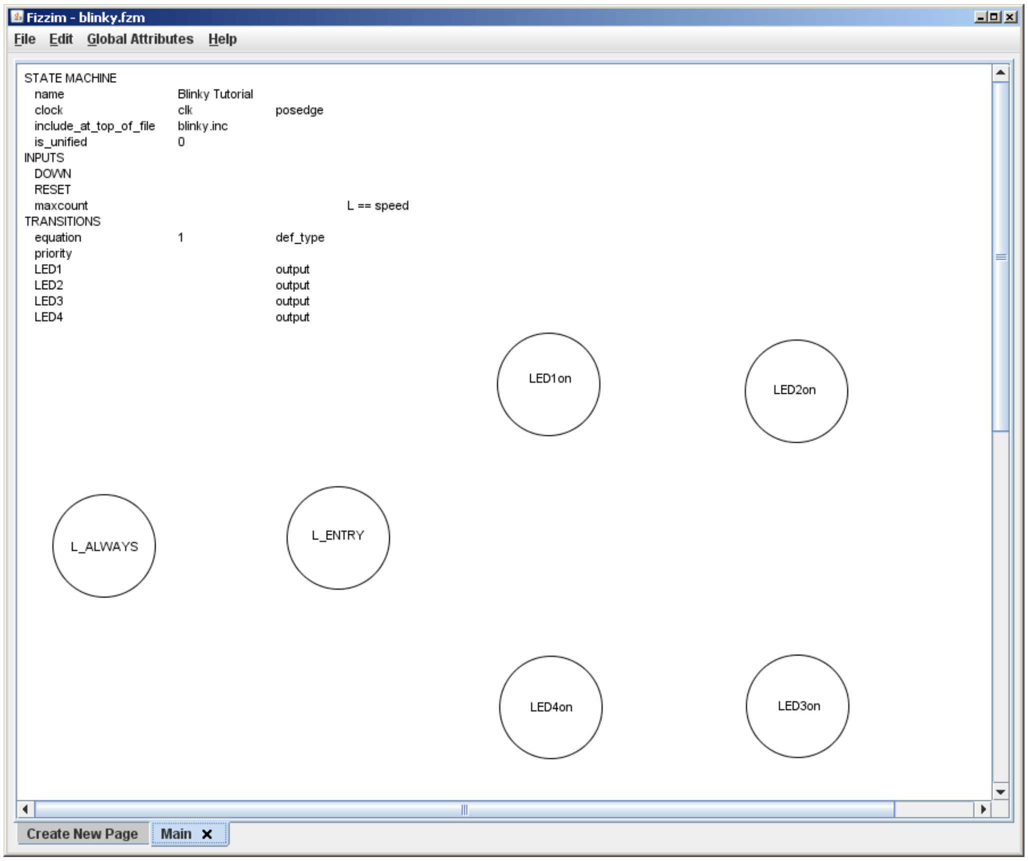
Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
DOWN		No			■
RESET		No			■
maxcount		No		L==speed	■

Fig 10. Match condition for Blinky project

The state machine will always start at node **L_ENTRY,** this is the first (entry) state for each configuration.

Now add four more states (right mouse click on the diagram -> **Quick New State**). The new states get default names, which you can change if you like to make the function of the state machine clearer. Let’s call them **LED1on, LED2on, etc**. The drawing would now look like this:



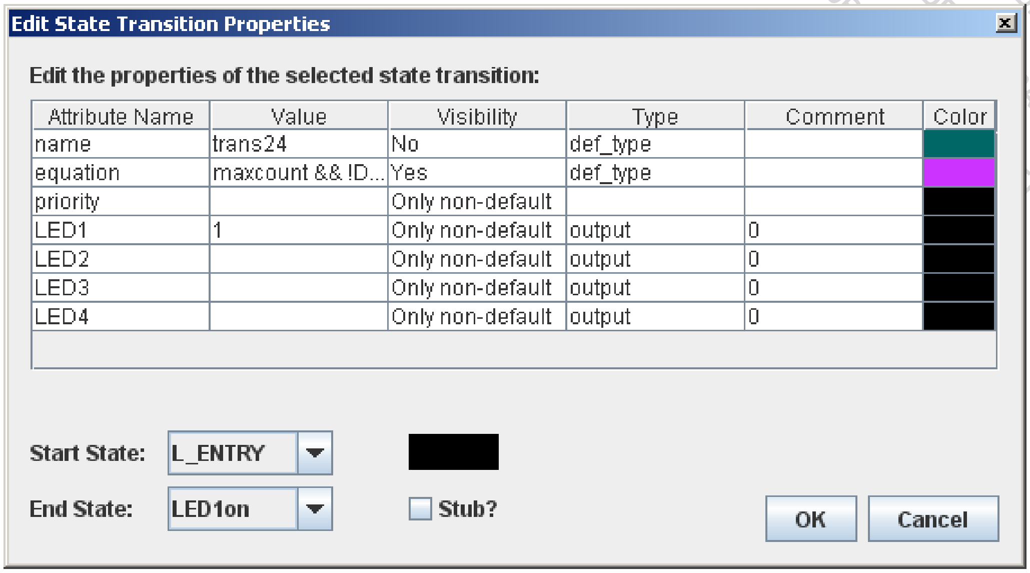
```
STATE MACHINE
  name                   Blinky Tutorial
  clock                  clk             posedge
  include_at_top_of_file blinky.inc
  is_unified             0
INPUTS
  DOWN
  RESET
  maxcount                                         L == speed
TRANSITIONS
  equation               1               def_type
  priority
  LED1                                   output
  LED2                                   output
  LED3                                   output
  LED4                                   output
```

Fig 11. blinky.fzm with all states

Right-click on the **L_ENTRY** state, and select **Add State Transition to...** to draw a transition to the **LED1on** state. Double-click on this new connection, and modify the equation from the default (shown as “1”) to **maxcount && !DOWN**.

This defines the condition for the state transition which appears in the *Value* column of the *equation* row.

Now change the value of the *LED1* row to **1**. The effect of a transition on a defined output is configured by specifying a non-empty *Value* field in one of the outputs listed.



Edit State Transition Properties

Edit the properties of the selected state transition:

Attribute Name	Value	Visibility	Type	Comment	Color
name	trans24	No	def_type		🟦
equation	maxcount && !D...	Yes	def_type		🟪
priority		Only non-default			⬛
LED1	1	Only non-default	output	0	⬛
LED2		Only non-default	output	0	⬛
LED3		Only non-default	output	0	⬛
LED4		Only non-default	output	0	⬛

Start State: L_ENTRY

End State: LED1on

□ Stub?

Fig 12. Transition dialog for Blinky project

As you can see, the idea is for this transition (= event) to set the output LED1=1, and leave all other outputs untouched, as we expect them to be all zero in state **L_ENTRY** (because we pre-configured them so, refer back to Fig 8).

The event definition is a bit more complicated: the equation *maxcount && !DOWN* means an event gets triggered when:

- the counter reaches the value *speed* (which is what *maxcount* has defined as a match value in the global table, see Fig 10)

AND

- the *DOWN* input is zero

at the same time.

This logical association is specified by the *&&* and the *!* operators, which are on purpose chosen to have the same meaning as in C language, which you should already be familiar with.

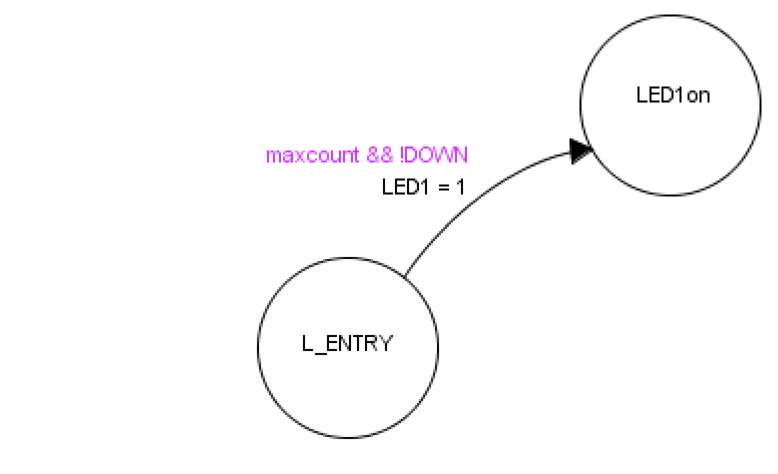


Fig 13. First transition in Blinky project

Repeat this for all other transitions, then add also the transitions for the other direction (DOWN=1). Do not forget to take care of the desired output action for each event!

This is what you should get at this point:

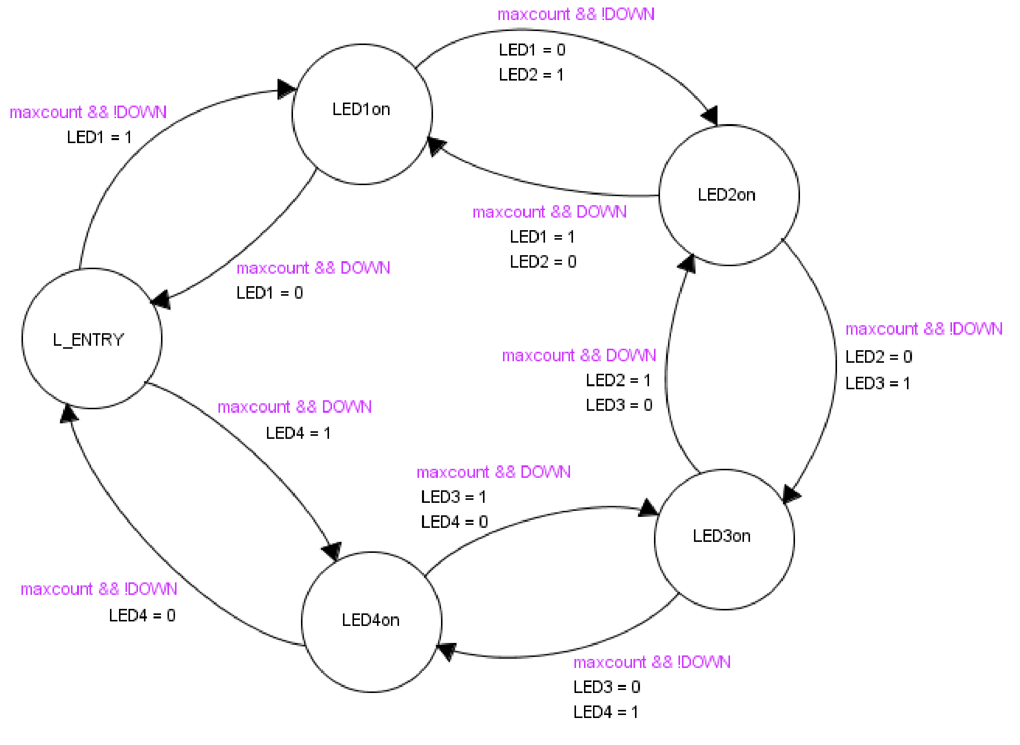


Fig 14. State machine for Blinky project

Only the RESET input is still missing. As we want the RESET input to be a global event, considered in each state, (so independently of the current state) we use the special state **L_ALWAYS**. This is a pseudo state which is gets not actually mapped to a specific physical state.

It is just a graphical convenience which allows specifying an event which has to be considered in all states of the state machine, but avoids having to replicate and draw the same event in every defined state. As a consequence, any event starting from **L_ALWAYS** will have its state mask set to all ones, so that it will be considered in any defined state.

When the RESET condition is verified (the RESET signal is at logic 1) we want the state machine to jump to the entry state L_ENTRY. To achieve this, draw a transition from **L_ALWAYS** to **L_ENTRY**. We do not know the state of the outputs when we receive a RESET signal, since it can happen asynchronously at any time, so we force all outputs low when this event happens.

Also, we want the state machine to stay in reset (in state **L_ENTRY)**, even if one of the events occurs that would make it evolve to state **LED1on** or **LED4on,** as long as the RESET signal is high. So the event associated with the RESET signal must have precedence.

For this we specify a numeric value for the *priority* field of the reset event. The code generator will automatically map the transition with highest priority to the highest event number. Per SCT design a higher event number (higher priority) always takes precedence over a lower event number (lower priority), when multiple events happen at the same time, but only one resulting state jump can be performed. So in this case the occurring event with the higher number (the higher priority value) will decide where the state machine will jump to.

However please note that outputs are not “filtered” or restricted by the priority settings; the priority order will determine the landing state, but each event which is defined and occurs while in a certain state will influence the outputs

In this sense, if your state machine can present such a race condition, care must be taken to ensure that a suitable conflict resolution is set for any affected outputs. See the section at the end of the document for the current limitations on this.

The next step towards our final state machine should look like this:

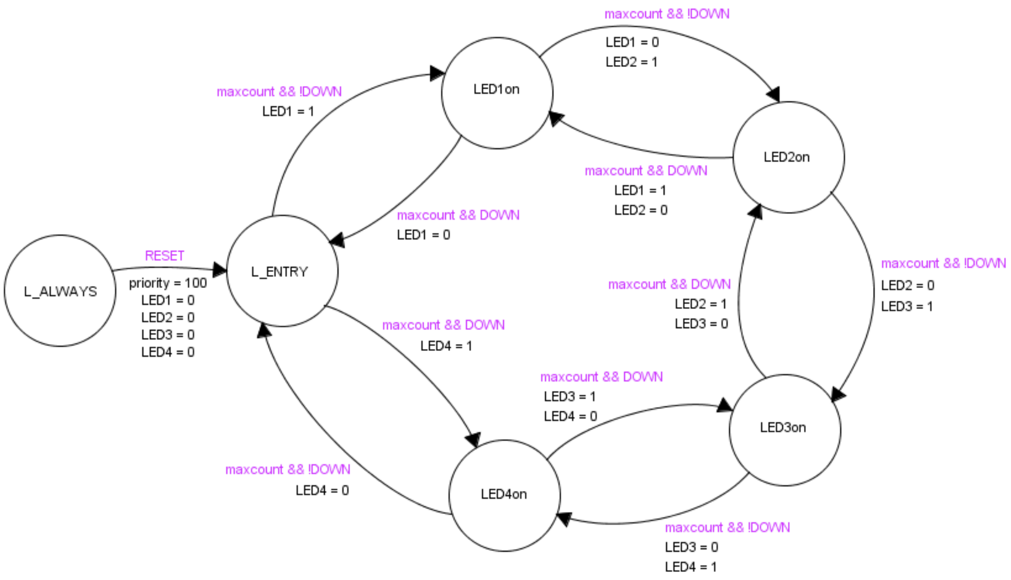
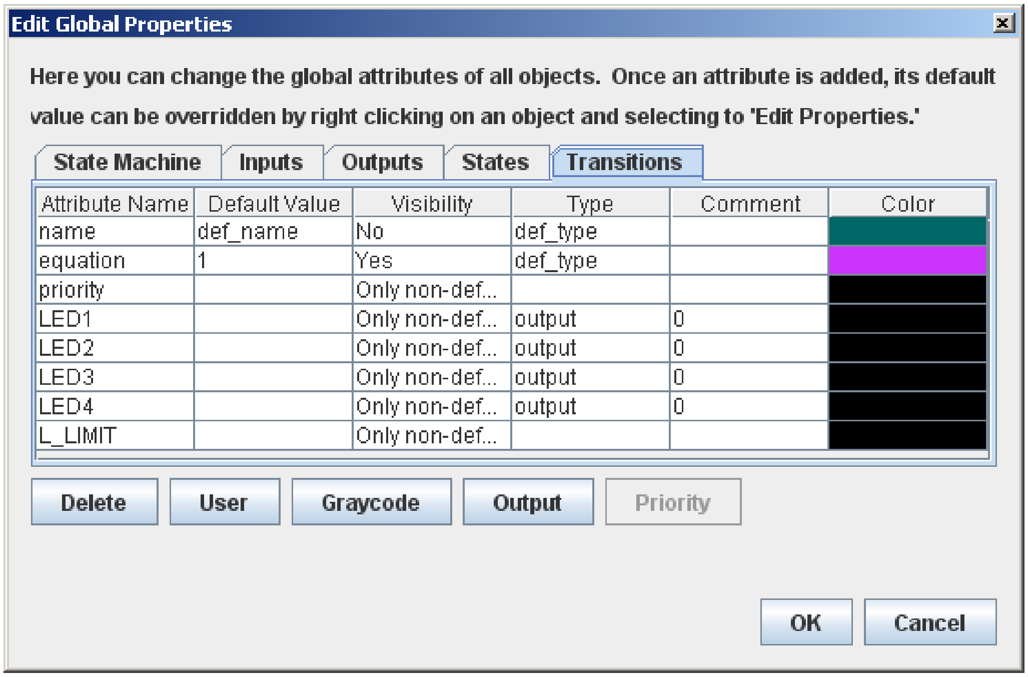


Fig 15. Almost final state machine of Blinky project

Before you compile the project again, add a definition for the *speed* value to the *sct_fsm.h* header file:

```
#define speed (10000-1)
```

The last step is to tell the counter to restart at 0 when he reaches *maxcount*. This is done by setting the signal **L_LIMIT** to 1 within an event. Add this signal to the global transitions table, and make sure its *Type* field is empty:



Edit Global Properties

Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
name	def_name	No	def_type		🟩
equation	1	Yes	def_type		🟪
priority		Only non-def...			⬛
LED1		Only non-def...	output	0	⬛
LED2		Only non-def...	output	0	⬛
LED3		Only non-def...	output	0	⬛
LED4		Only non-def...	output	0	⬛
L_LIMIT		Only non-def...			⬛

Fig 16. Add the L_LIMIT signal

Limiting the counter is a “global” (pseudo) event as well, and it shall not change the state machine. Therefore it is added as a loopback transition to the **L_ALWAYS** state.

Now we have completed the state machine:

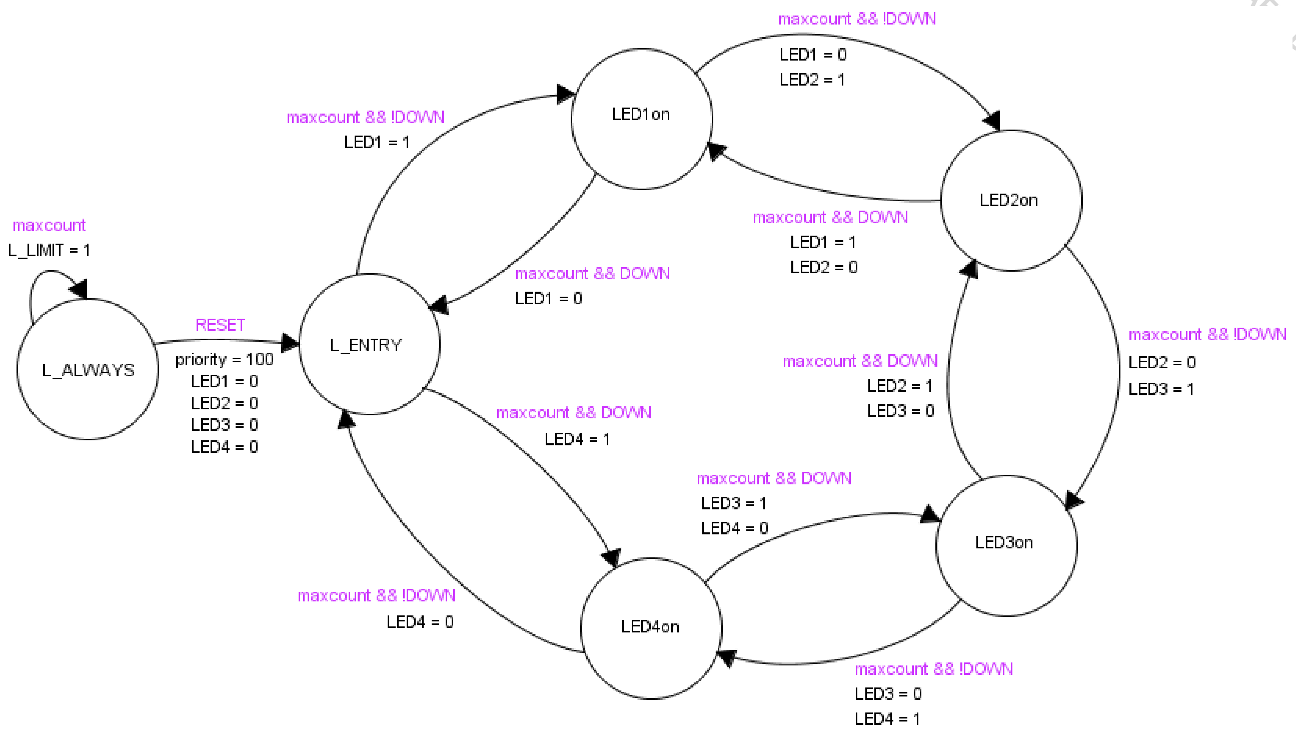
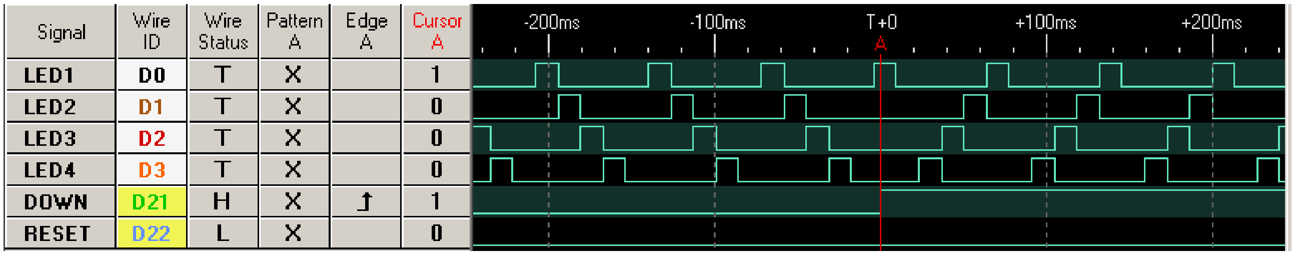


Fig 17. Final state machine of Blinky project

You can run the program on the target, and you should get the outputs changing as in the following logic analyzer screen shot:



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
LED1	D0	T	X		1
LED2	D1	T	X		0
LED3	D2	T	X		0
LED4	D3	T	X		0
DOWN	D21	H	X	↥	1
RESET	D22	L	X		0

Fig 18. Output of Blinky project

A copy of the solution is provided as *blinky_full.fzm* within the solutions folder

4. Examples quick reference

4.1 Basic examples

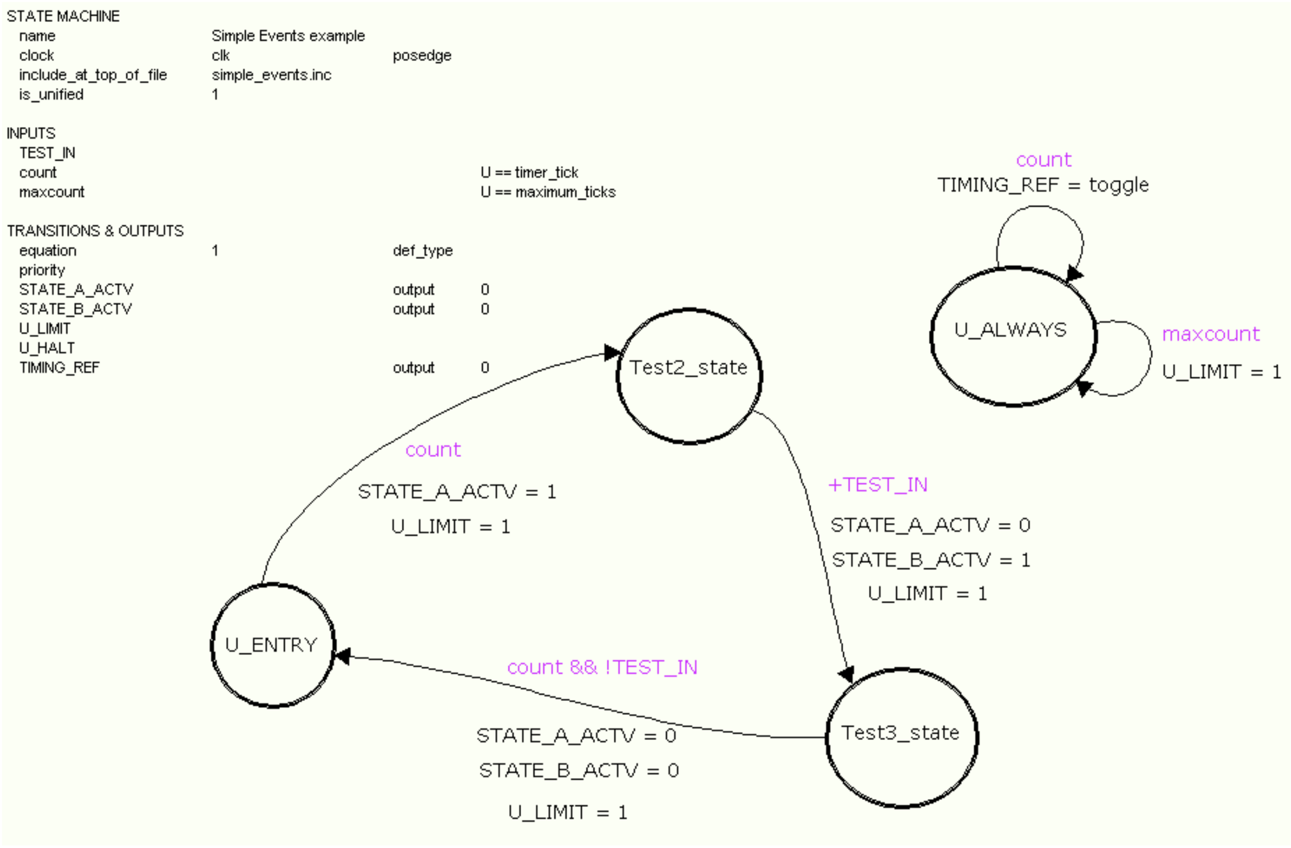
4.1.1 Blinky

Please see the tutorial at section 3.1

4.1.2 Simple_events

The example shows some combinations of event types. The following are provided:

- time based match (first event)
- input rising edge (second event)
- match and input low (third event)



```
STATE MACHINE
  name                     Simple Events example
  clock                    clk                      posedge
  include_at_top_of_file   simple_events.inc
  is_unified               1

INPUTS
  TEST_IN
  count                                                        U == timer_tick
  maxcount                                                     U == maximum_ticks

TRANSITIONS & OUTPUTS
  equation                 1                        def_type
  priority
  STATE_A_ACTV                                      output     0
  STATE_B_ACTV                                      output     0
  U_LIMIT
  U_HALT
  TIMING_REF                                        output     0
```

Fig 19. State machine for the simple_events example

The SCT is configured in unified mode (1 32 bit timer)

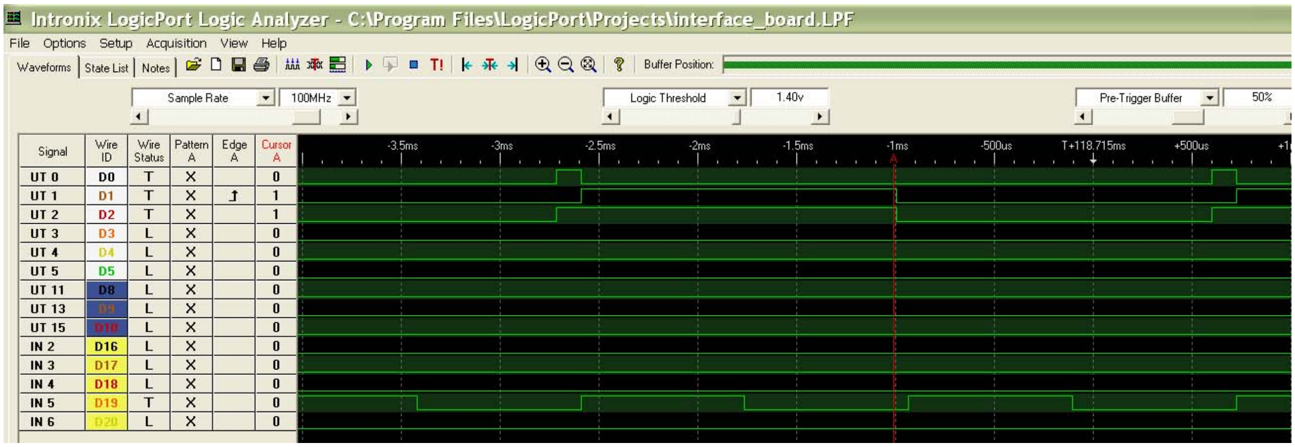
Notes:

- the third event will happen only when both conditions count AND !TEST_IN are satisfied, which means the last transition (back to state U_ENTRY) will depend on the relative timing condition between the CPU generated signal (TEST_IN) and the counter match 'count' condition. Modifying the CPU loop will change the amount of time for which the SCT will be kept in the Test3_state (until both sides of the 'AND' condition are satisfied)

- for convenience, the same match value (count) is also used to generate a reference signal (TIMING_REF) which always toggles, and is used to graphically visualize when the counter reaches the 'count' value, in order to analyze the diagrams more easily
- There is also another event which is in the ALWAYS pseudo state and used to limit the timer to count up to a maximum value 'maxcount'. This is done to avoid having to wait for the counter to roll-over in case we are in Test3_state and the TEST_IN input is not low at the moment the timer match 'count' happens

Following are some screenshots of the expected outputs. The first trace represents the STATE_A_ACTV signal, the second the STATE_B_ACTV signal, the third trace is the TIMING_REF signal, the bottom trace is the TEST_IN signal

:


Intronix LogicPort Logic Analyzer - C:\Program Files\LogicPort\Projects\interface_board.LPF

Sample Rate: 100MHz

Logic Threshold: 1.40v

Pre-Trigger Buffer: 50%

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
UT 0	D0	T	X		0
UT 1	D1	T	X	↑	1
UT 2	D2	T	X		1
UT 3	D3	L	X		0
UT 4	D4	L	X		0
UT 5	D5	L	X		0
UT 11	D8	L	X		0
UT 13	D9	L	X		0
UT 15	D10	L	X		0
IN 2	D16	L	X		0
IN 3	D17	L	X		0
IN 4	D18	L	X		0
IN 5	D19	T	X		0
IN 6	D20	L	X		0

Fig 20. simple_events example: entering the entry state (red cursor)

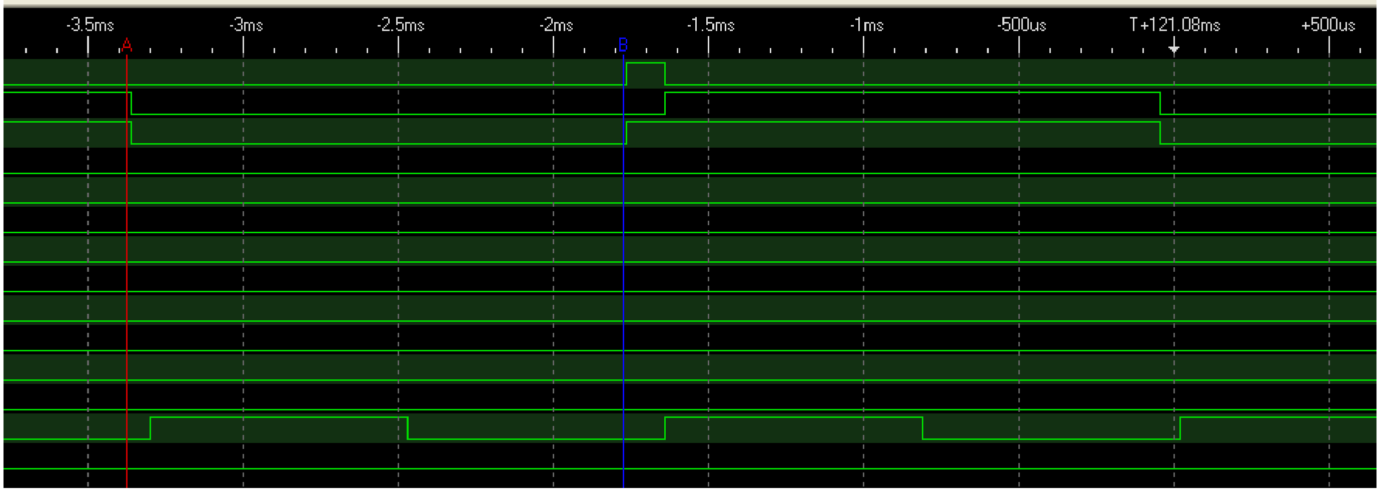


Fig 21. simple_events example: transition to test2_state (blue cursor)

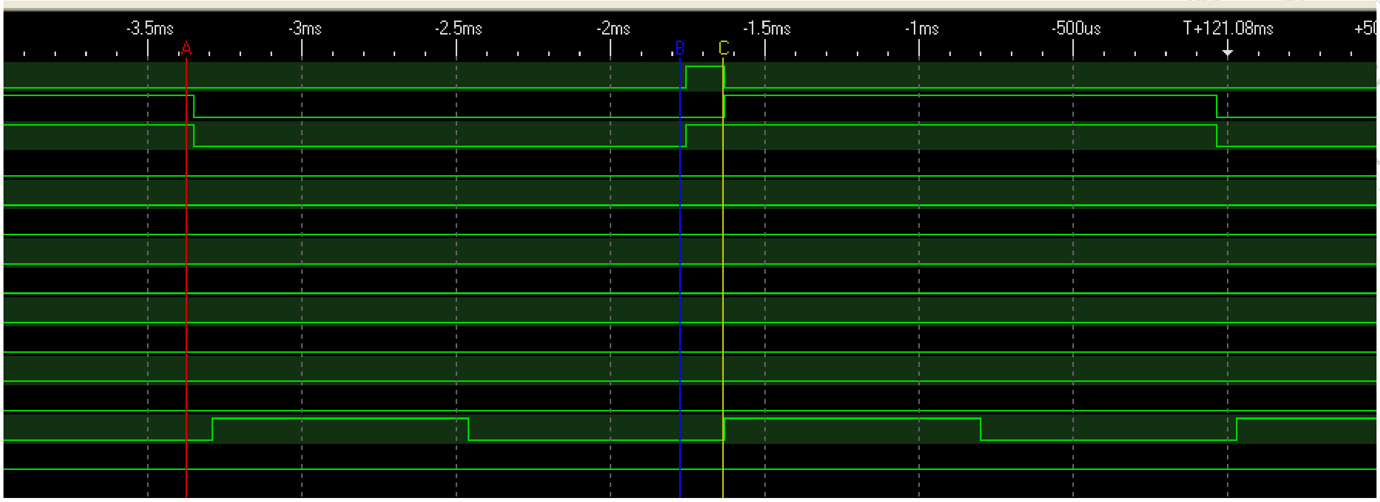


Fig 22. simple_events example: transition to test3_state (yellow cursor)

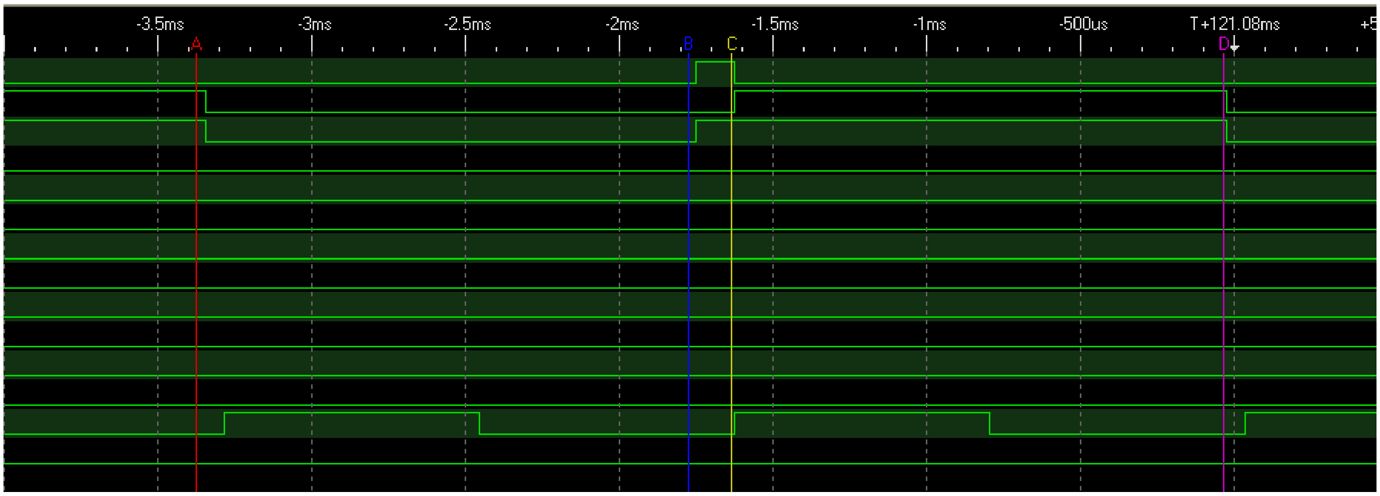
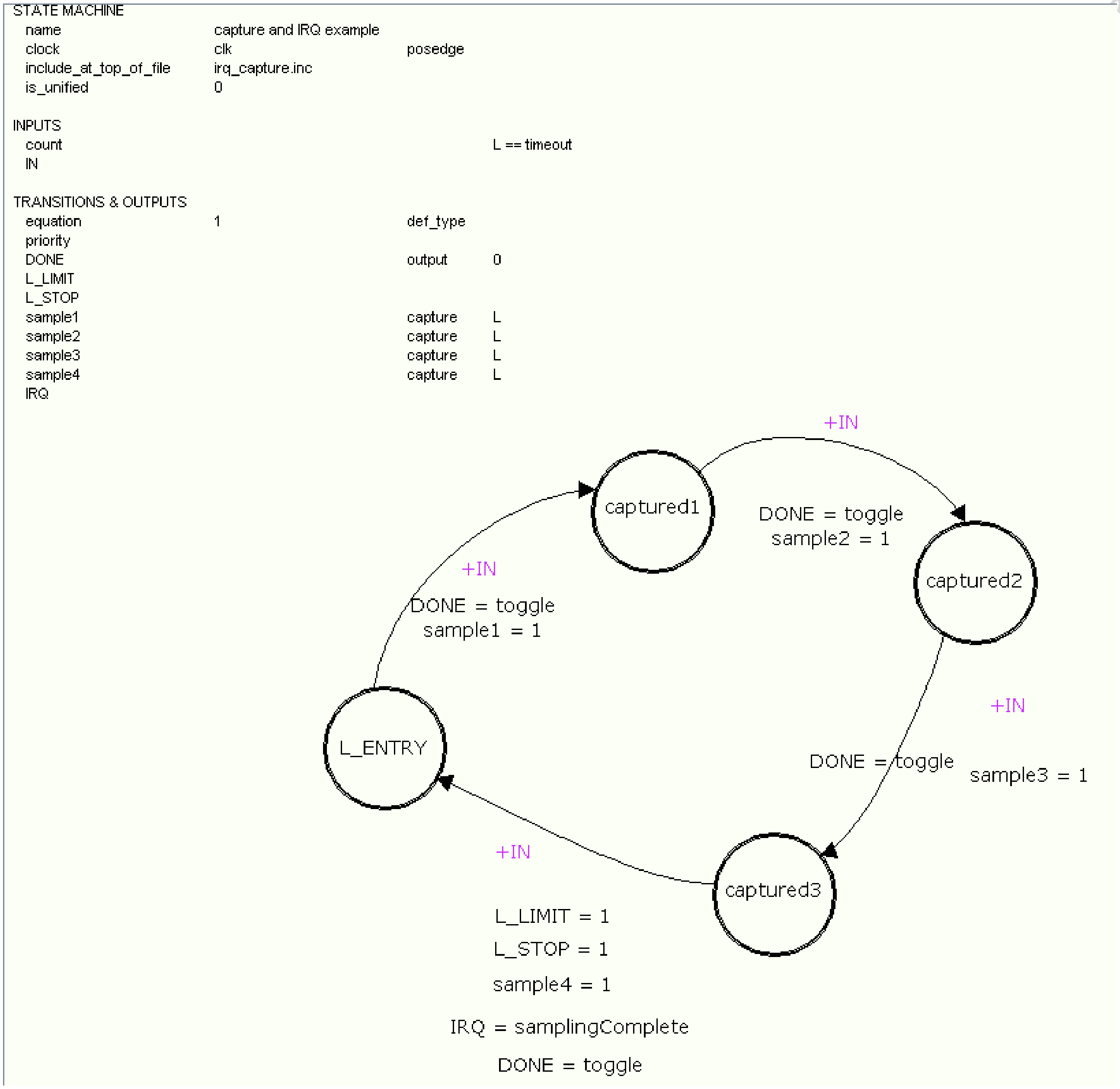


Fig 23. simple_events example: transition back to the entry state (purple cursor)

4.1.3 Simple_irq_capture

This example shows how to configure capture events for sampling the counter value at specific input signal conditions.



STATE MACHINE			
name	capture and IRQ example		
clock	clk	posedge	
include_at_top_of_file	irq_capture.inc		
is_unified	0		
INPUTS			
count			L == timeout
IN			
TRANSITIONS & OUTPUTS			
equation	1	def_type	
priority			
DONE		output	0
L_LIMIT			
L_STOP			
sample1		capture	L
sample2		capture	L
sample3		capture	L
sample4		capture	L
IRQ			

Fig 24. State machine for the simple_irq_capture example

In this case the sampling signal called IN is set as the rising edge of an external GPIO input, toggled in sw by the CPU.

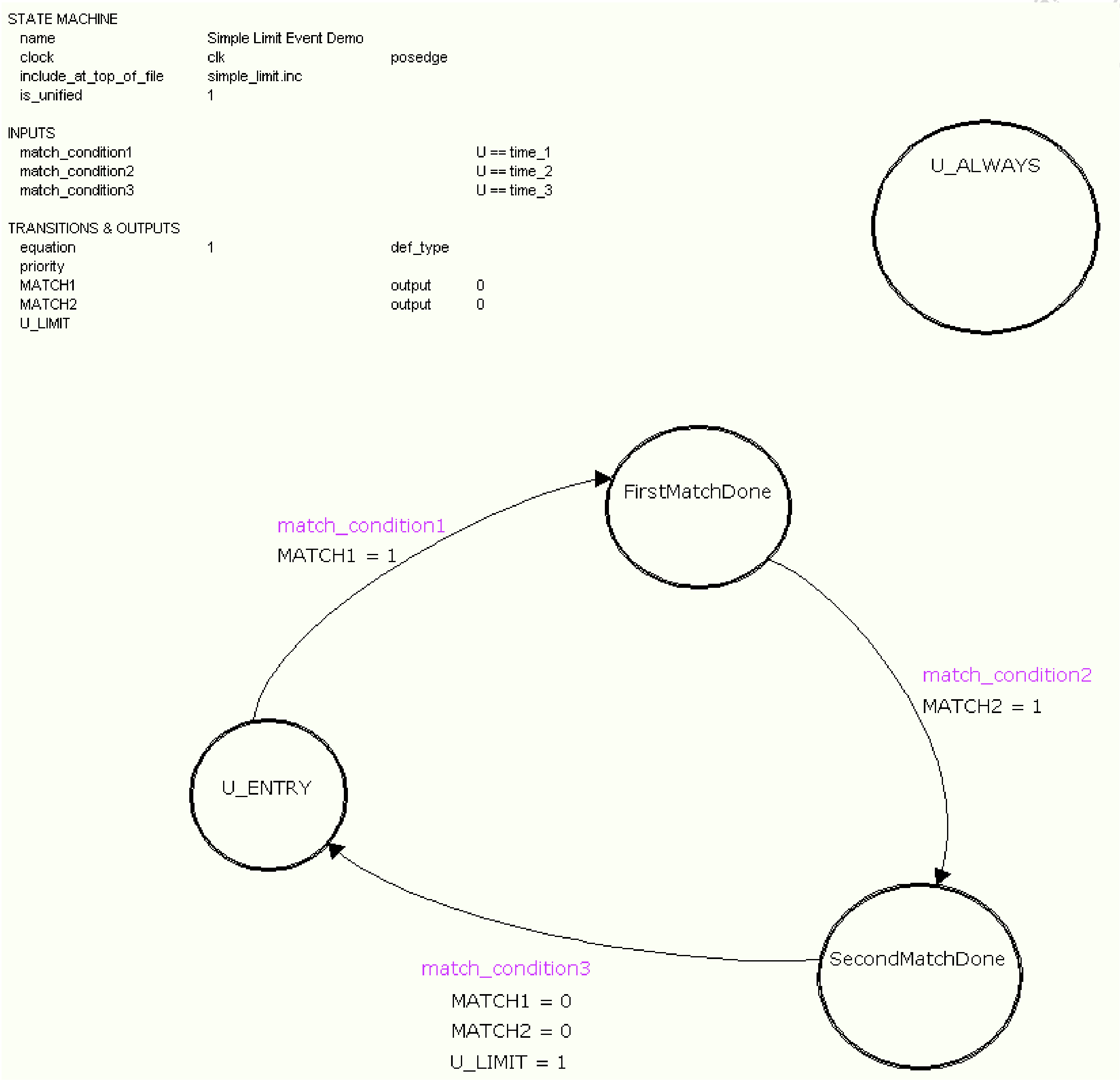
Every time a sample is acquired, the DONE output signal is toggled as a convenience signal. During the same event which triggers the capture of sample4 (sample4 = 1), and the SCT jumps from state captured3 back to the entry state, the counter is also set back to zero (L_LIMIT = 1), is stopped (L_STOP = 1) and an IRQ called *samplingComplete* is generated.

The application can acknowledge the interrupt and then retrieve the data and by using the tool generated macros, for example:

```
LPC_SCT->EVFLAG = (1 << SCT_IRQ_EVENT_samplingComplete);
sample_buffer[3] = SCT_CAPTURE_sample4;
```

4.1.4 Simple_limit

This example shows three different match conditions scheduled to happen at successive instants in time, chosen at 100,200,300.



STATE MACHINE			
name	Simple Limit Event Demo		
clock	clk	posedge	
include_at_top_of_file	simple_limit.inc		
is_unified	1		
INPUTS			
match_condition1			U == time_1
match_condition2			U == time_2
match_condition3			U == time_3
TRANSITIONS & OUTPUTS			
equation	1	def_type	
priority			
MATCH1		output	0
MATCH2		output	0
U_LIMIT			

Fig 25. State machine for the simple_limit example

The first two transitions (events) match_condition1, match_condition2 set a different output pin (MATCH1, MATCH2)

The third event (match_contition3) clears both outputs and also limits the timer, i.e. it sets its value back to zero.

The effect of limiting the timer is proven to happen since without it, the counter would have to reach its maximum value and then overflow/roll over to zero, before the first event (configured at timer tick 100) would happen again. Instead it happens right after 100 timer ticks as desired

The SCT is configured in unified mode (one single 32 bit counter)

Below a screenshot of the expected outputs; the first two traces represent signals MATCH1 and MATCH2

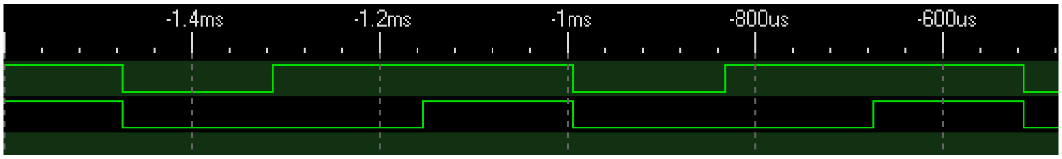
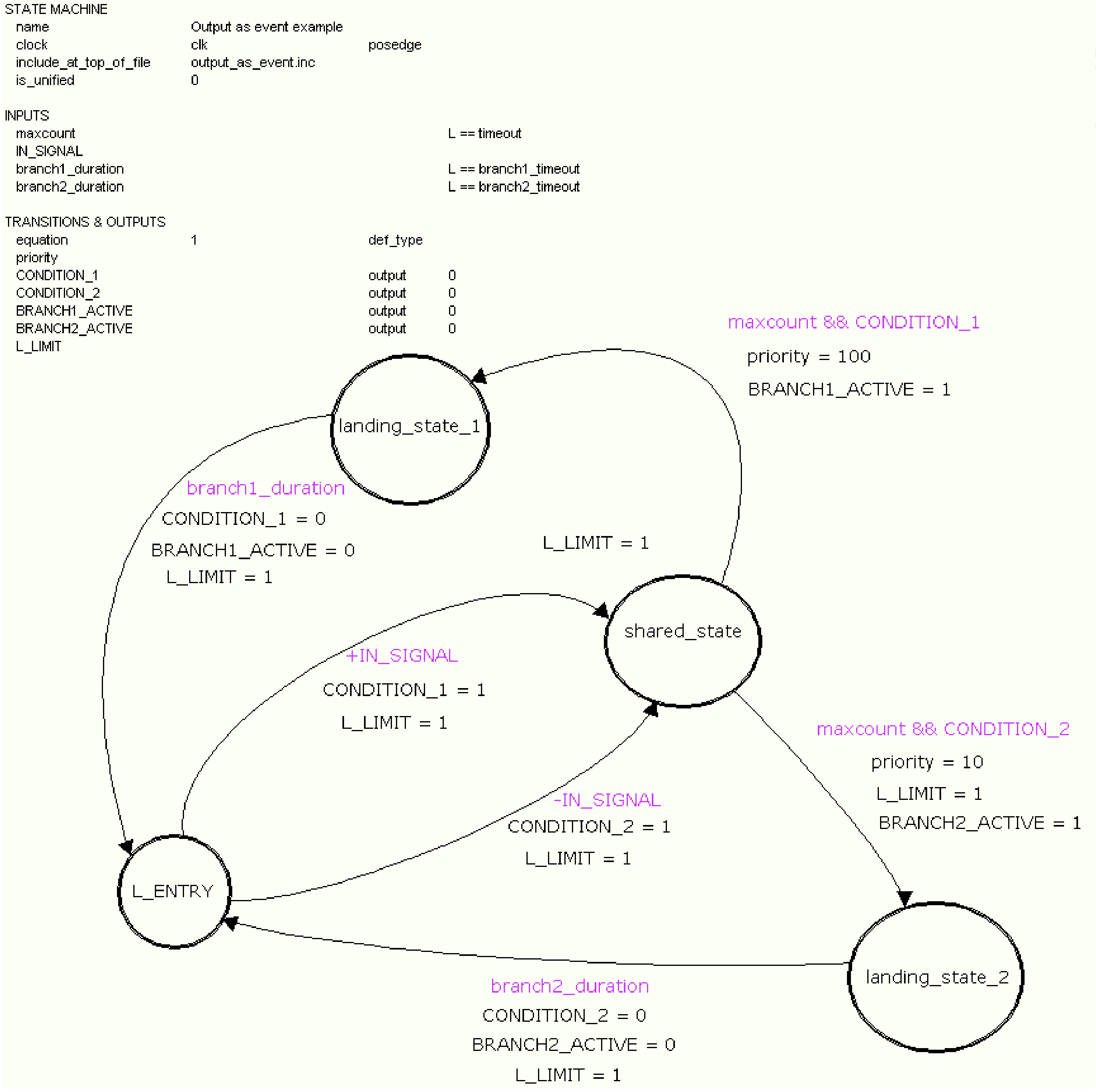


Fig 26. State machine for the simple_limit example

4.1.5 Simple_output_as_event



```
STATE MACHINE
  name                     Output as event example
  clock                    clk                         posedge
  include_at_top_of_file   output_as_event.inc
  is_unified               0

INPUTS
  maxcount                                                           L == timeout
  IN_SIGNAL
  branch1_duration                                                   L == branch1_timeout
  branch2_duration                                                   L == branch2_timeout

TRANSITIONS & OUTPUTS
  equation                 1                           def_type
  priority
  CONDITION_1                                          output        0
  CONDITION_2                                          output        0
  BRANCH1_ACTIVE                                       output        0
  BRANCH2_ACTIVE                                       output        0
  L_LIMIT
```

Fig 27. State machine for the simple_output_as_event example

This example shows how to use an output as an internal signal to condition the state machine evolution

The input signal called IN_SIGNAL is generated in sw by the CPU, by toggling a GPIO pin.

The signals *condition1* and *condition2* generated by the SCT are used only internally and depend if a rising or falling edge of signal IN_SIGNAL comes first.

When in the *shared_state*, the previously set condition will determine the landing state of the state machine, at the time *maxcount* match happens.

If a rising edge was seen first, the state machine will evolve in landing_state_1, else if a falling edge was seen first, the state machine will evolve in landing_state_2.

Note: it could have been possible to use the negated version of signal condition1 instead of having a separate signal called condition2, specifying the event going to landing_state_2 as *maxcount && !CONDITION_1 (*but that´s just for example purposes)

Since the duration (timeout condition) of each branch is different, the BRANCH1_ACTIVE and BRANCH2_ACTIVE debug outputs will alternate with each other in time, depending on the relative timing of the time constants used, and the IN_SIGNAL pattern.

Below an example of a possible output sequence

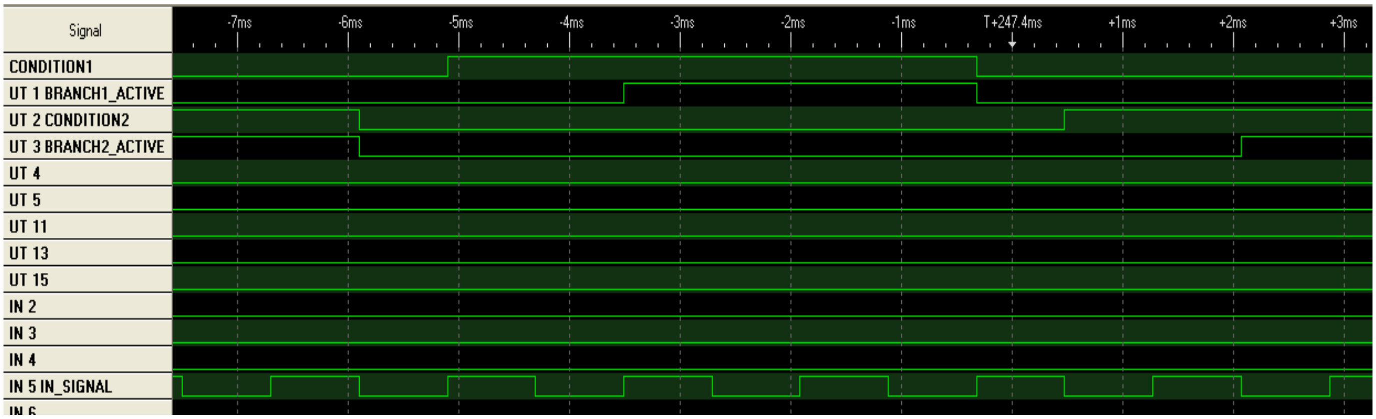
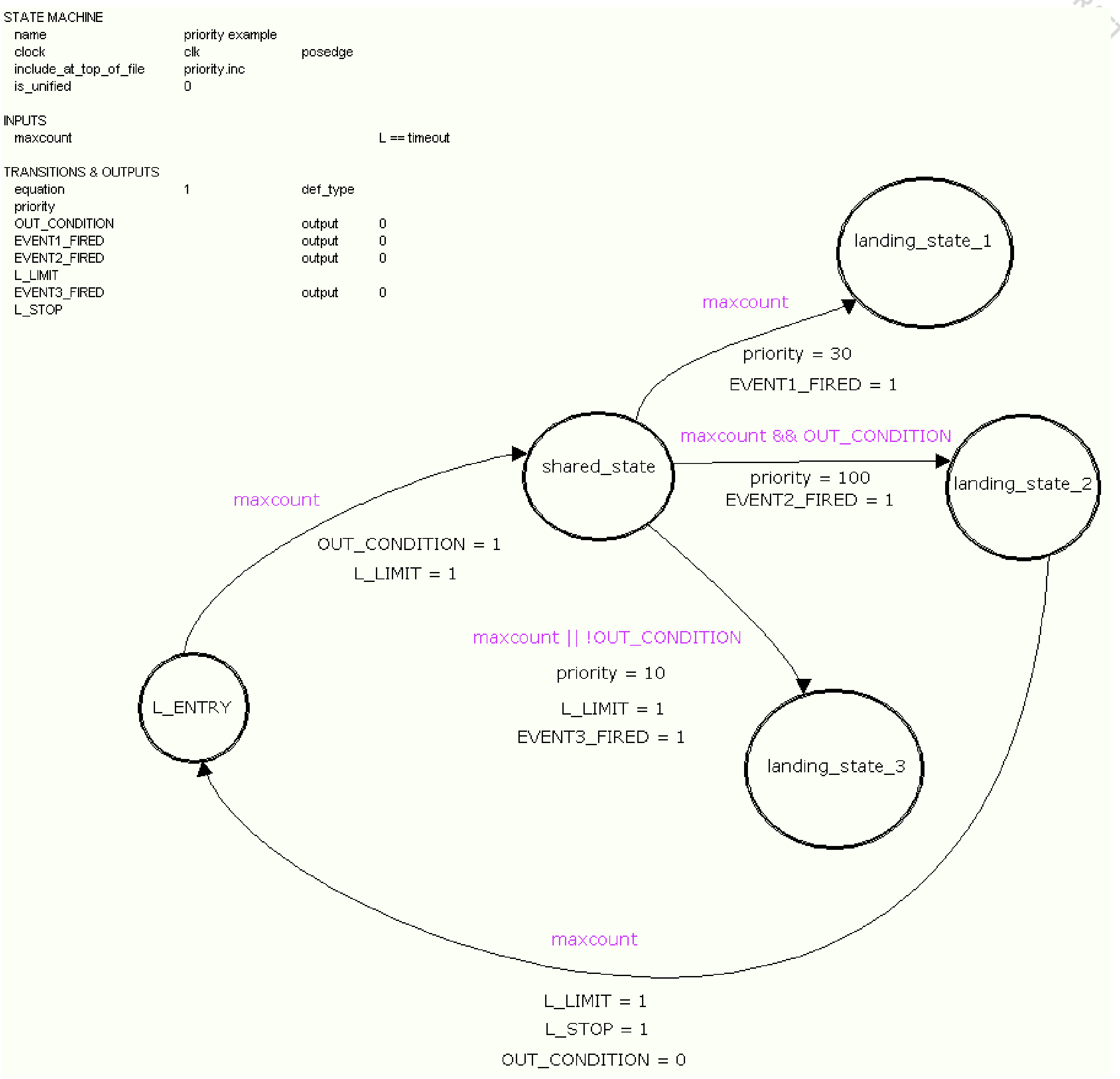


Fig 28. Simple_output_as_event example outputs

4.1.6 Simple_priority

This example shows the effect of the priority feature on the state machine evolution



```
STATE MACHINE
  name                     priority example
  clock                    clk                posedge
  include_at_top_of_file   priority.inc
  is_unified               0

INPUTS
  maxcount                                              L == timeout

TRANSITIONS & OUTPUTS
  equation                 1                  def_type
  priority
  OUT_CONDITION                               output    0
  EVENT1_FIRED                                output    0
  EVENT2_FIRED                                output    0
  L_LIMIT
  EVENT3_FIRED                                output    0
  L_STOP
```

Fig 29. State machine of simple_priority example

The state machine evolves first onto the *shared_state* when *maxcount* expires.

After the countdown *maxcount* expires the second time, all of the condition for the events:

- *maxcount*
- *maxcount && OUT_CONDITION*
- *maxcount || !OUT_CONDITION*

are satisfied, hence all three events happen within the shared state. This can be seen since all of the three associated outputs EVENT1_FIRED, EVENT2_FIRED, EVENT3_FIRED get activated. However since the priority of the second event:

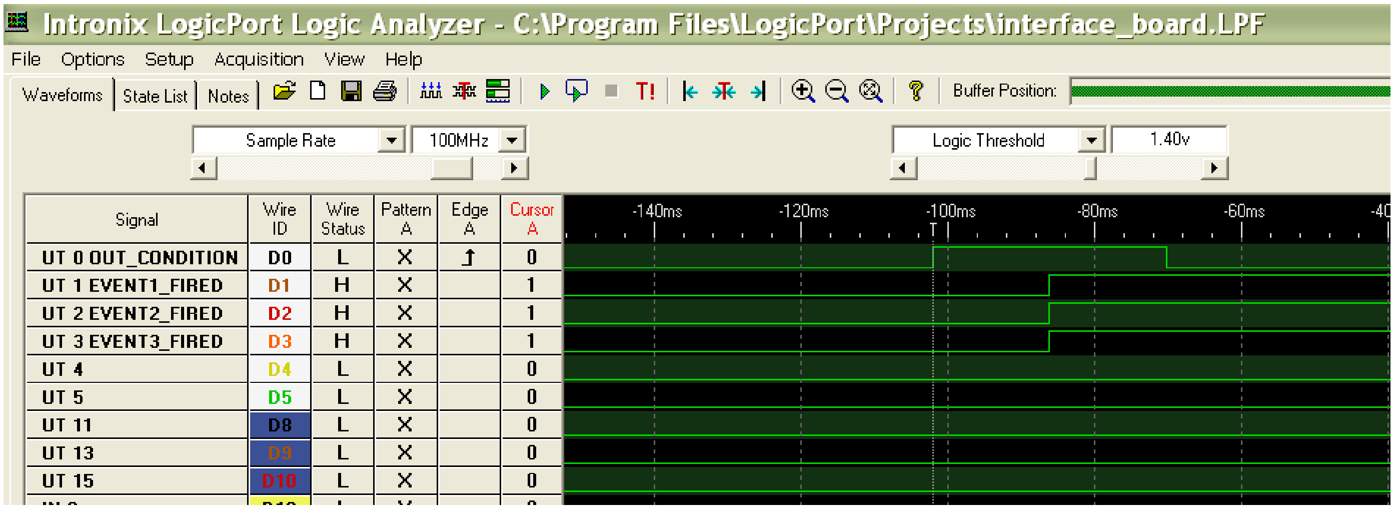
maxcount && OUT_CONDITION

is configured greater (100) than the priority of the other two events (10, 30) the state machine will evolve to the *landing_state_2*. This is confirmed by the fact that after the next countdown expires, the SCT evolves back to the entry state and the associated transition (event) clears the OUT_CONDITION output.

If any of the other two events would have been configured to determine the landing state of the SCT, the system would have been kept "trapped" forever into either *landing_state_1* or *landing_state_3* (from which there is no exit).

Specifying a priority field in the graphical tool allows a user to take advantage of an SCT design feature for which events mapped to a higher index number will have higher priority over other concurrent events which might happen in the same state. So the tool automatically maps the higher priority transitions to higher event numbers according to the user settings.

Below the expected outputs:



Sample Rate 100MHz

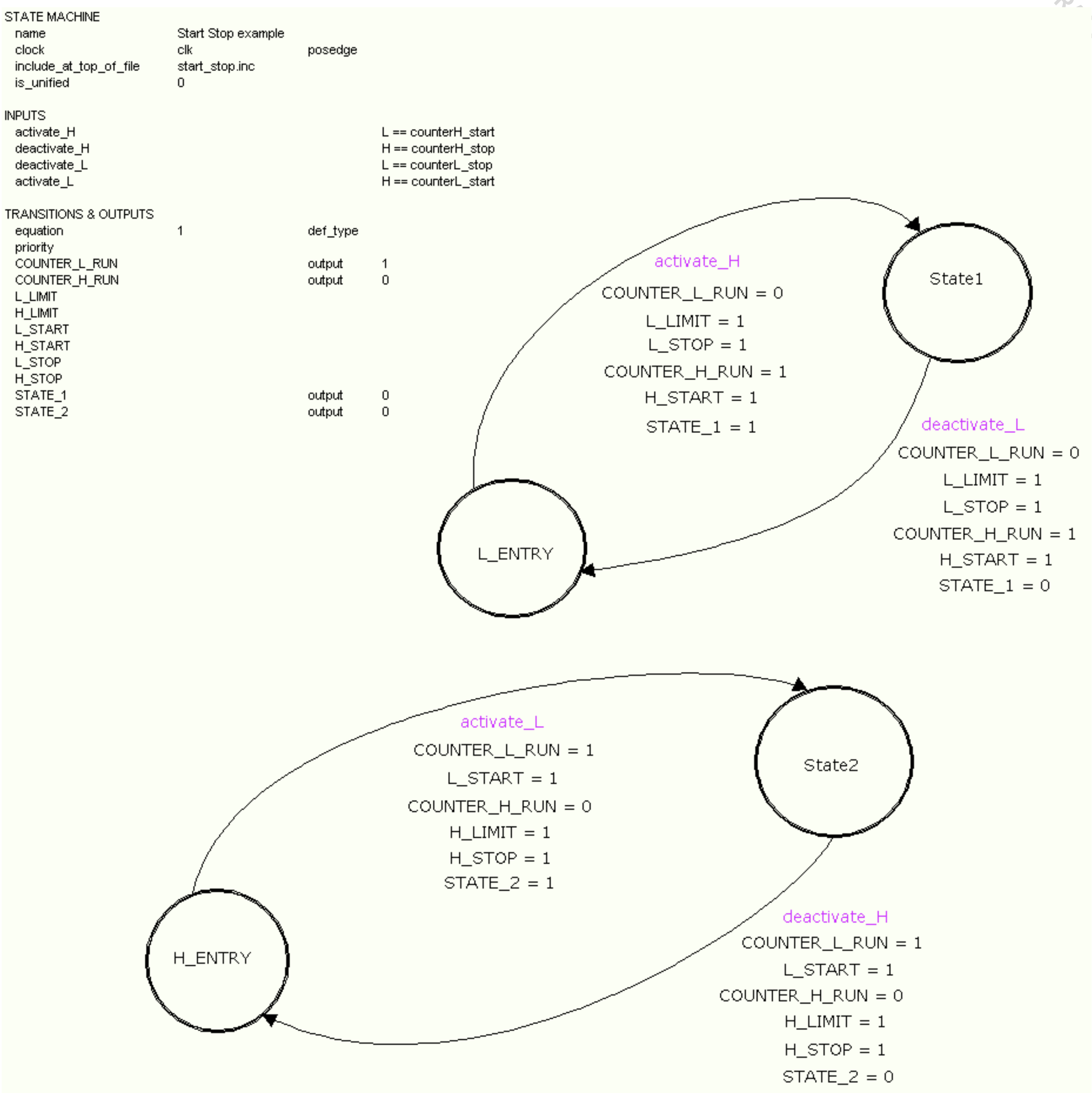
Logic Threshold 1.40v

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
UT 0 OUT_CONDITION	D0	L	X	↥	0
UT 1 EVENT1_FIRED	D1	H	X		1
UT 2 EVENT2_FIRED	D2	H	X		1
UT 3 EVENT3_FIRED	D3	H	X		1
UT 4	D4	L	X		0
UT 5	D5	L	X		0
UT 11	D8	L	X		0
UT 13	D9	L	X		0
UT 15	D10	L	X		0

Fig 30. Simple_priority example outputs

4.1.7 Start_stop

This example shows the usage of start and stop events.



```
STATE MACHINE
  name                    Start Stop example
  clock                   clk                 posedge
  include_at_top_of_file  start_stop.inc
  is_unified              0

INPUTS
  activate_H                                          L == counterH_start
  deactivate_H                                        H == counterH_stop
  deactivate_L                                        L == counterL_stop
  activate_L                                          H == counterL_start

TRANSITIONS & OUTPUTS
  equation                1                   def_type
  priority
  COUNTER_L_RUN                               output    1
  COUNTER_H_RUN                               output    0
  L_LIMIT
  H_LIMIT
  L_START
  H_START
  L_STOP
  H_STOP
  STATE_1                                     output    0
  STATE_2                                     output    0
```

Fig 31. State machine of start_stop example project

The SCT is configured in split mode (2 separate 16 bit timers)

Each half of the timer generates start and stop events for itself and the other side of the timer. These alternate with each other in a ping-pong like fashion.

The L counter is started first, while the H counter stays stopped. Within the activate_H match condition the L counter is limited and stopped, and the H counter is started.

The H counter (the state machine at the bottom) generates the activate_L condition, for which the L counter is started again. The same event stops and limits the H counter

A similar pattern repeats itself making the two counters evolve back to their Entry states (first L, then H).

4.2 More advanced examples

4.2.1 Traffic light control

This example shows a more advanced demo project for a control of a traffic light

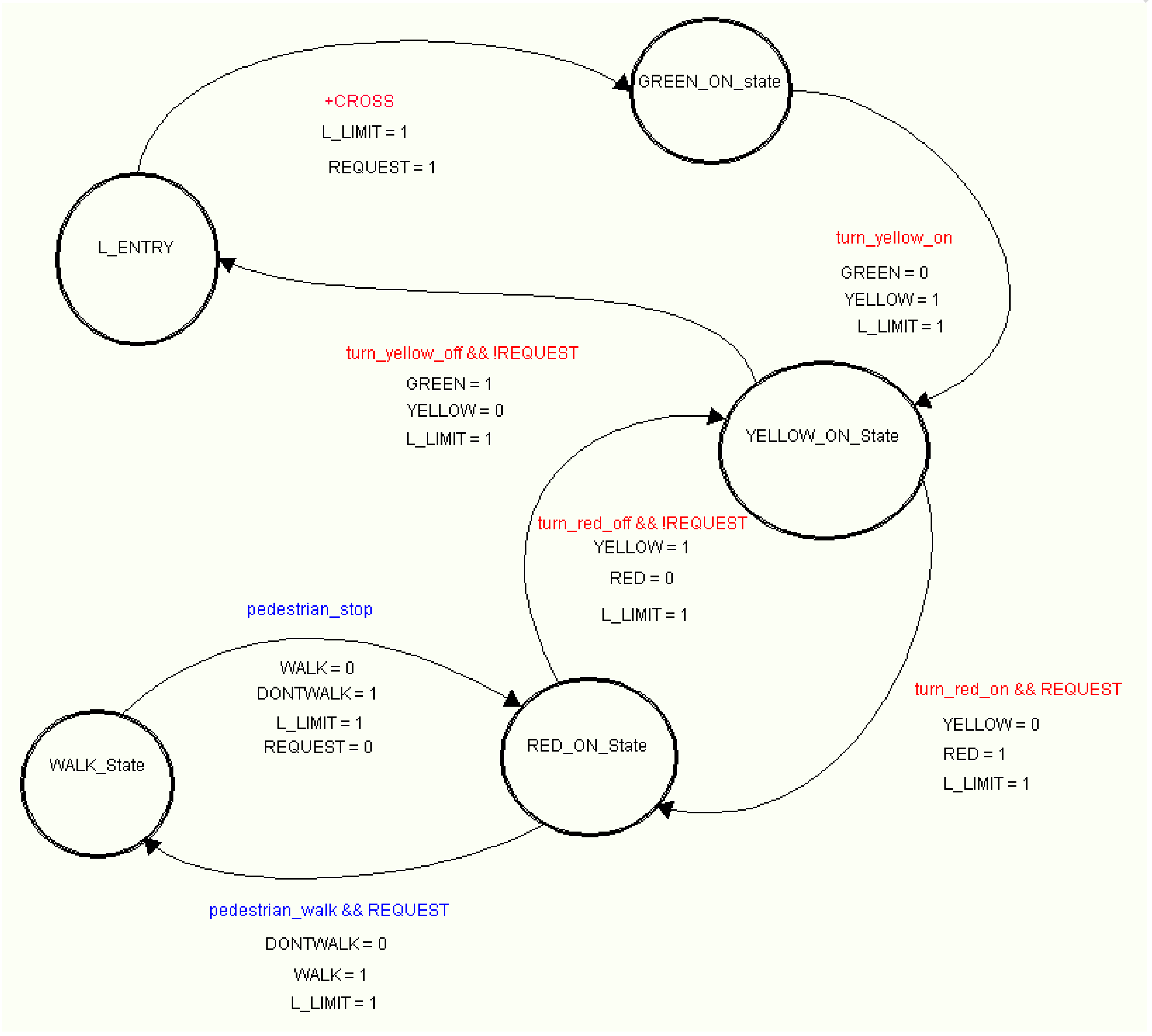


Fig 32. State machine of the traffic light demo

The state machine enters in a state where it waits for a rising edge on the input signal called *CROSS*.

When this happens, the pedestrian triggered request gets logged with the *REQUEST* output, which is used for internal purposes only.

After a small delay when the *turn_yellow_on* match happens, the traffic light is changed to yellow. In the *yellow_on_state*, when the *turn_red_on* match happens, the SCT will jump into the *red_on_state*, since this transaction is the only one which is conditioned to the *REQUEST* signal being high (which the SCT has already set at the beginning).

In the *red_on_state*, when the *pedestrian_walk && REQUEST* match happens the pedestrian green light (*WALK*) will be turned on, then turned off when *pedestrian_stop* match value has elapsed, returning to the *red_on_state*.

Then the sequence for the traffic lights is repeated in the opposite sense (turn to yellow, then back to green).

Since the *pedestrian_stop* event has cleared the *REQUEST* signal, the state machine will take the *turn_red_off && ! REQUEST*, *turn_yellow_off && !REQUEST* path when rolling back.

The lights are driven by a series of different match values which control the phases for the switching in an independent way.

4.2.2 Digital PLL

The SCT generates a clock which is synchronous to an external clock input.

The DPLL will lock if the input frequency is the same or an integer fraction of the center frequency of the DPLL. The center frequency is fixed to SCT_clock/32.

You must supply an external clock input to the SCT if you need a center frequency which cannot be provided directly by the peripheral clock divider.

This design follows an appnote about the 74LS297 DPLL device.

The selected settings are: SCT clock = 72 MHz --> Fc = 2.25 MHz

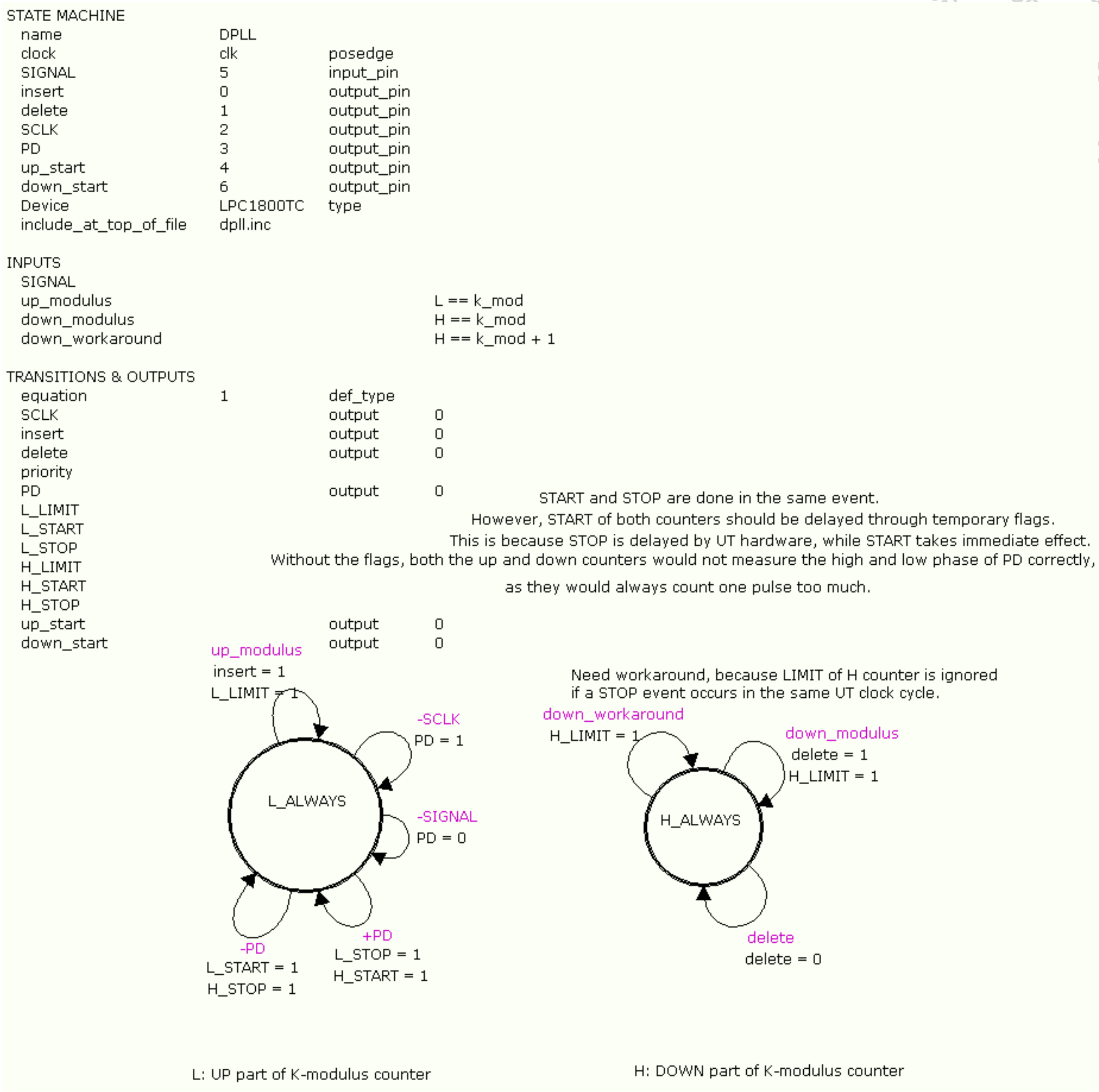
(Please note that the software uses the IRC, and the actual frequency may not be exact. Use a crystal if you need higher accuracy).

The lock range (Fc/K) is fairly small: With K=64, the range is +/- 35 kHz.

This design uses an edge-triggered phase detector (kd=2).

Clock dividers are: N=32, 2M=N=32.

Following some screenshots of the SCT configuration



```
STATE MACHINE
   name                       DPLL
   clock                      clk          posedge
   SIGNAL                     5            input_pin
   insert                     0            output_pin
   delete                     1            output_pin
   SCLK                       2            output_pin
   PD                         3            output_pin
   up_start                   4            output_pin
   down_start                 6            output_pin
   Device                     LPC1800TC    type
   include_at_top_of_file     dpll.inc

INPUTS
   SIGNAL
   up_modulus                                         L == k_mod
   down_modulus                                       H == k_mod
   down_workaround                                    H == k_mod + 1

TRANSITIONS & OUTPUTS
   equation                   1            def_type
   SCLK                                    output     0
   insert                                  output     0
   delete                                  output     0
   priority
   PD                                      output     0
   L_LIMIT
   L_START
   L_STOP
   H_LIMIT
   H_START
   H_STOP
   up_start                                output     0
   down_start                              output     0
```

START and STOP are done in the same event.
However, START of both counters should be delayed through temporary flags.
This is because STOP is delayed by UT hardware, while START takes immediate effect.
Without the flags, both the up and down counters would not measure the high and low phase of PD correctly,
as they would always count one pulse too much.

Need workaround, because LIMIT of H counter is ignored
if a STOP event occurs in the same UT clock cycle.

L: UP part of K-modulus counter

H: DOWN part of K-modulus counter

Fig 33. State machine of digital pll example – part1

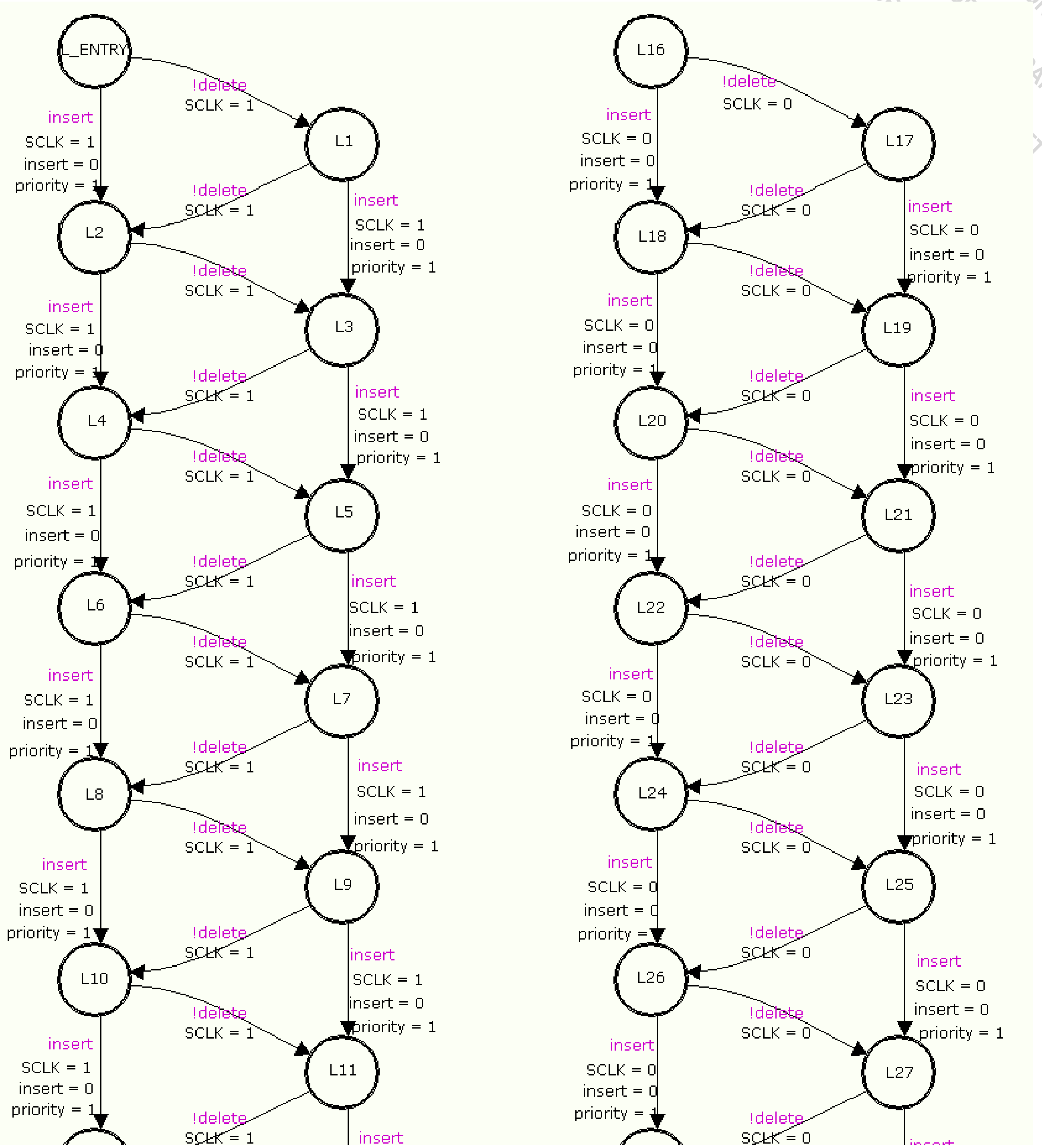


Fig 34. State machine of digital pll example – part2

4.2.3 Frequency measurement

This example shows how to configure capture events for sampling an external frequency signal

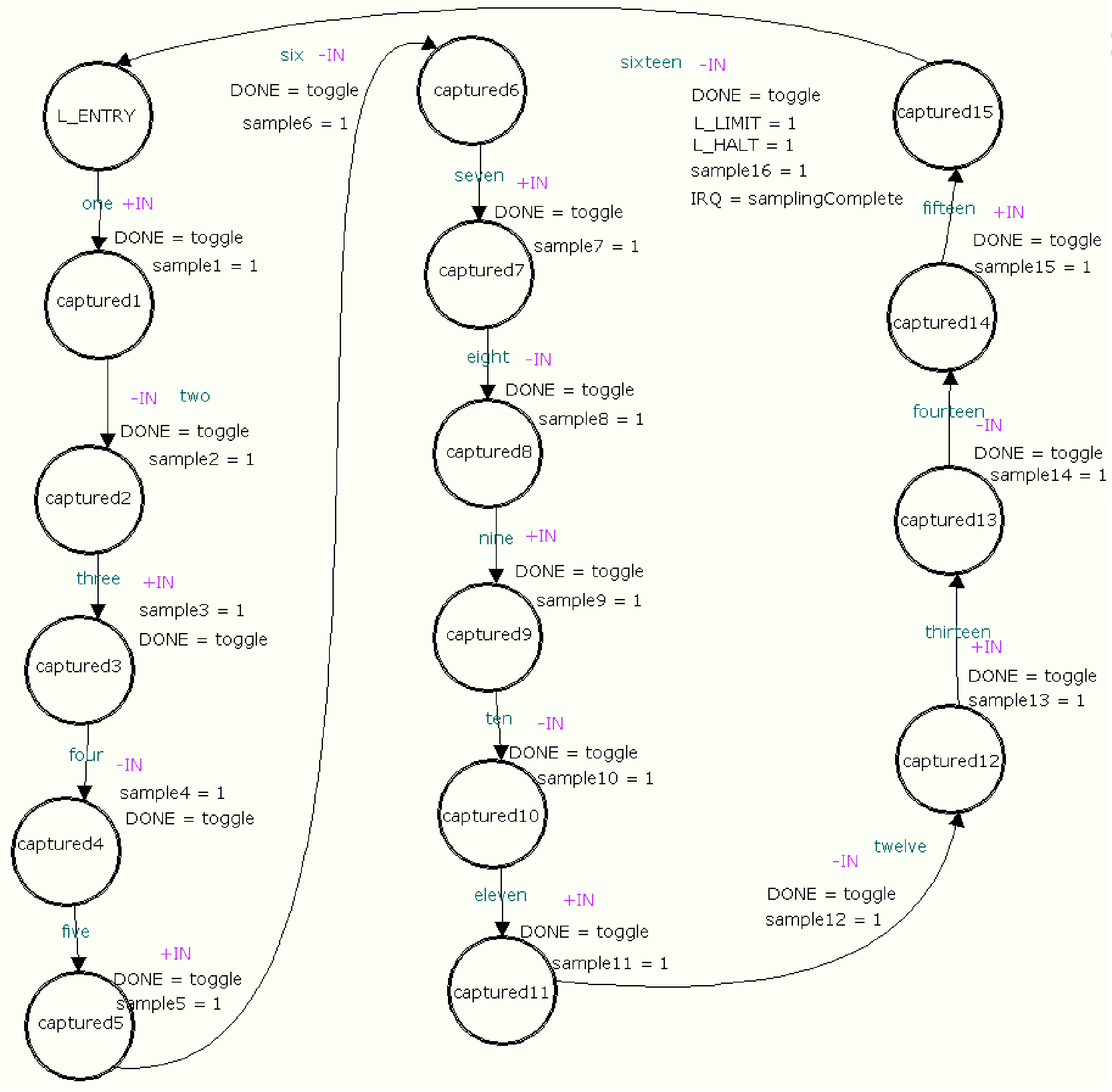


Fig 35. State machine of frequency measurement example

The sampling signal is set as the rising edge or falling edge of an external input.

At the end an IRQ is generated, the counter limited, halted, and the application retrieves the data which is put in two buffers.

The entry state is on the top left of the diagram.

The measured frequency values are then calculated within the application and put into the buffers called freq_rising, freq_falling

The signal output DONE is used as a convenience output to visually check on a scope when the sampling has occurred

Below an example of the generated output:

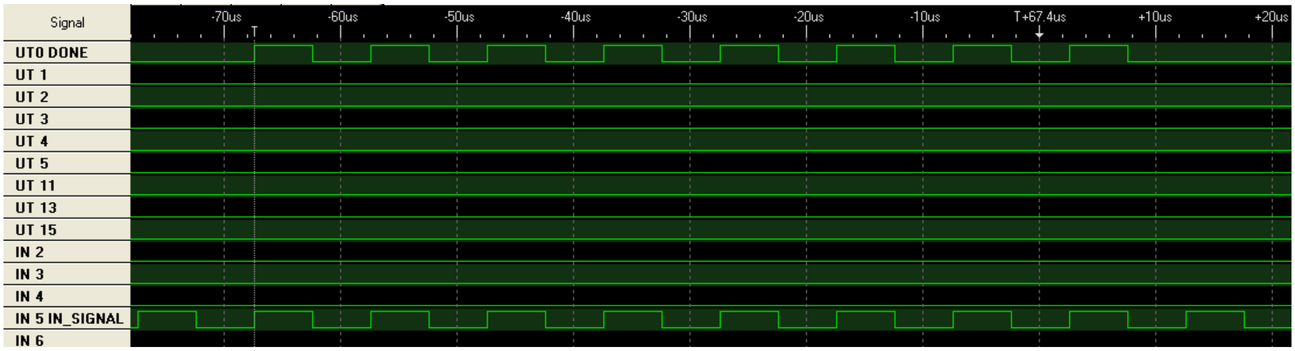


Fig 36. Output of frequency measurement example

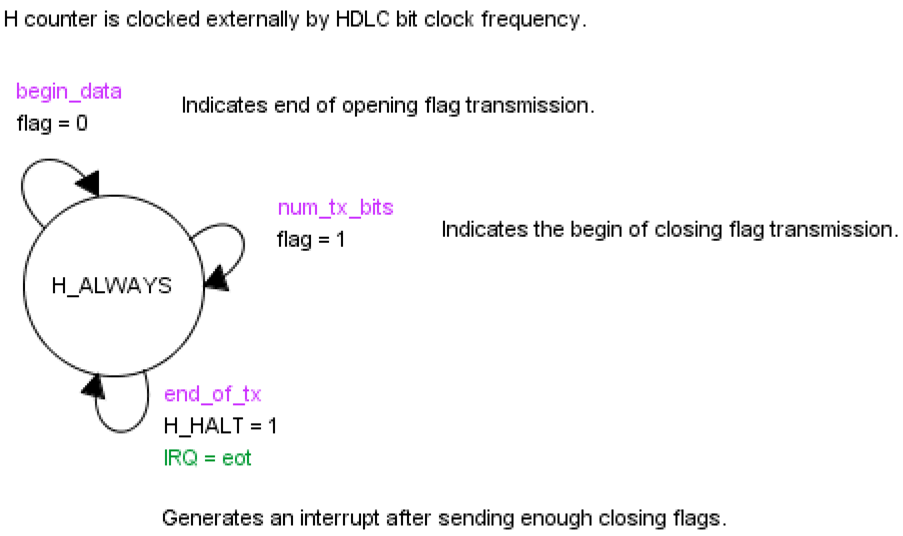
At each edge of the DONE signal a capture value has been taken, so 8 rising + 8 falling period measurements are done in total

After the SCT has gone though all states, at the last sample the counter is halted.

4.3 Application examples

4.3.1 HDLC Transmitter

……



H counter is clocked externally by HDLC bit clock frequency.

Indicates end of opening flag transmission.

Indicates the begin of closing flag transmission.

Generates an interrupt after sending enough closing flags.

Fig 37. State machine of HDLC-TX project

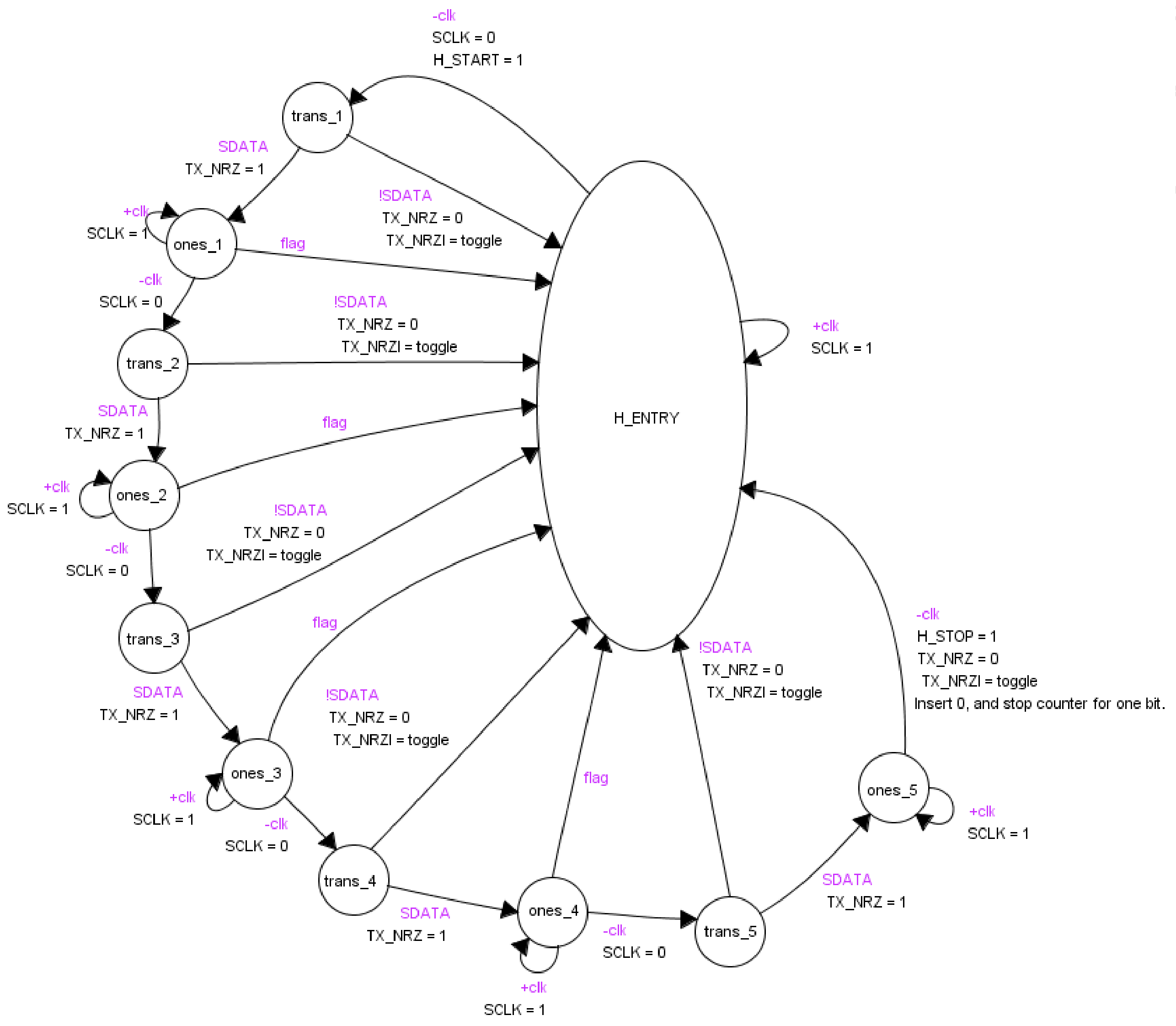
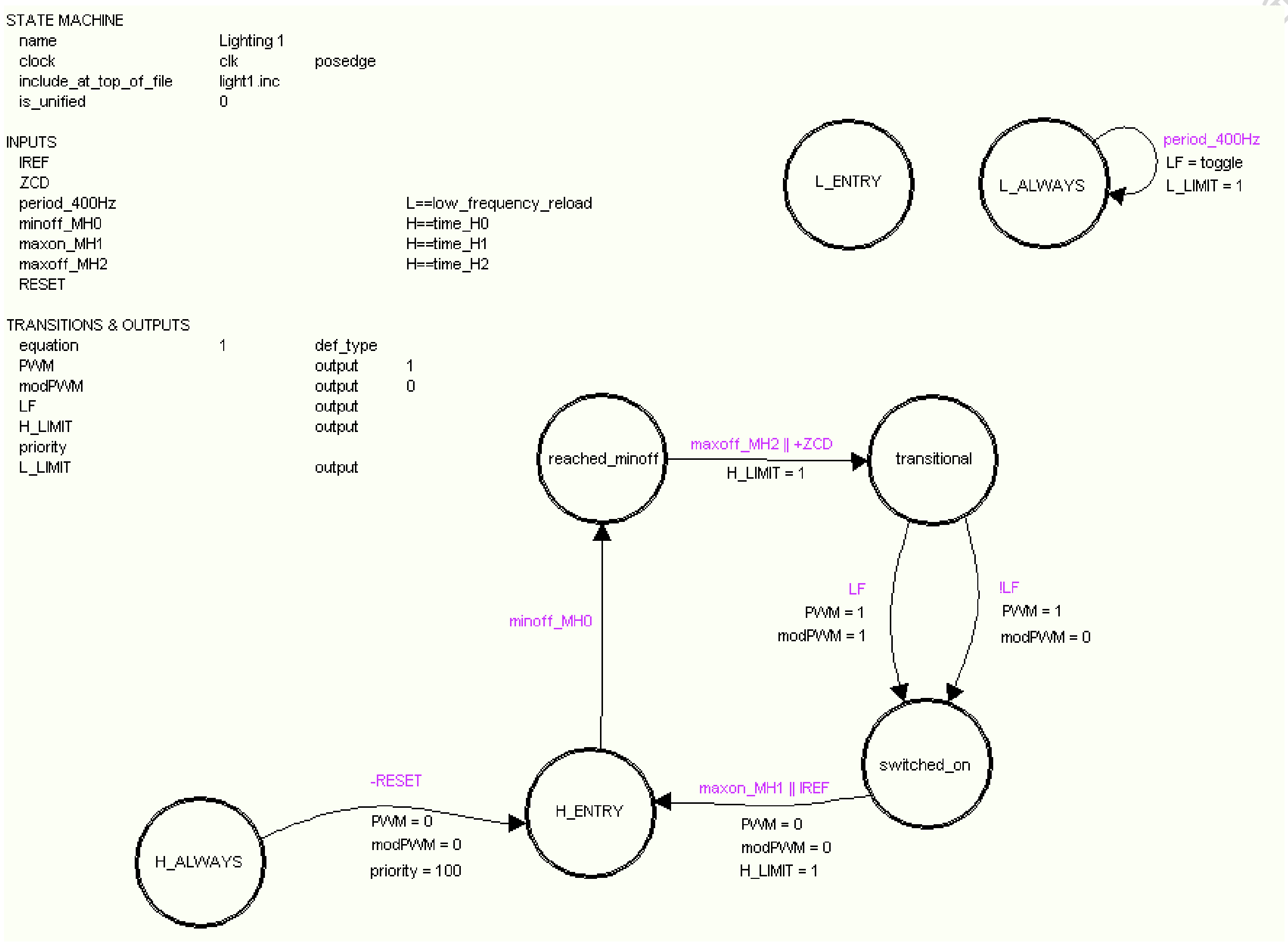


Fig 38. State machine of HDLC-TX project

4.3.2 Lighting control

This example generates a high frequency PWM signal, modulated by a low frequency signal, which can be used for lighting control.



```
STATE MACHINE
  name                  Lighting 1
  clock                 clk          posedge
  include_at_top_of_file light1.inc
  is_unified            0

INPUTS
  IREF
  ZCD
  period_400Hz                                    L==low_frequency_reload
  minoff_MH0                                      H==time_H0
  maxon_MH1                                       H==time_H1
  maxoff_MH2                                      H==time_H2
  RESET

TRANSITIONS & OUTPUTS
  equation              1            def_type
  PWM                                output     1
  modPWM                             output     0
  LF                                 output
  H_LIMIT                            output
  priority
  L_LIMIT                            output
```

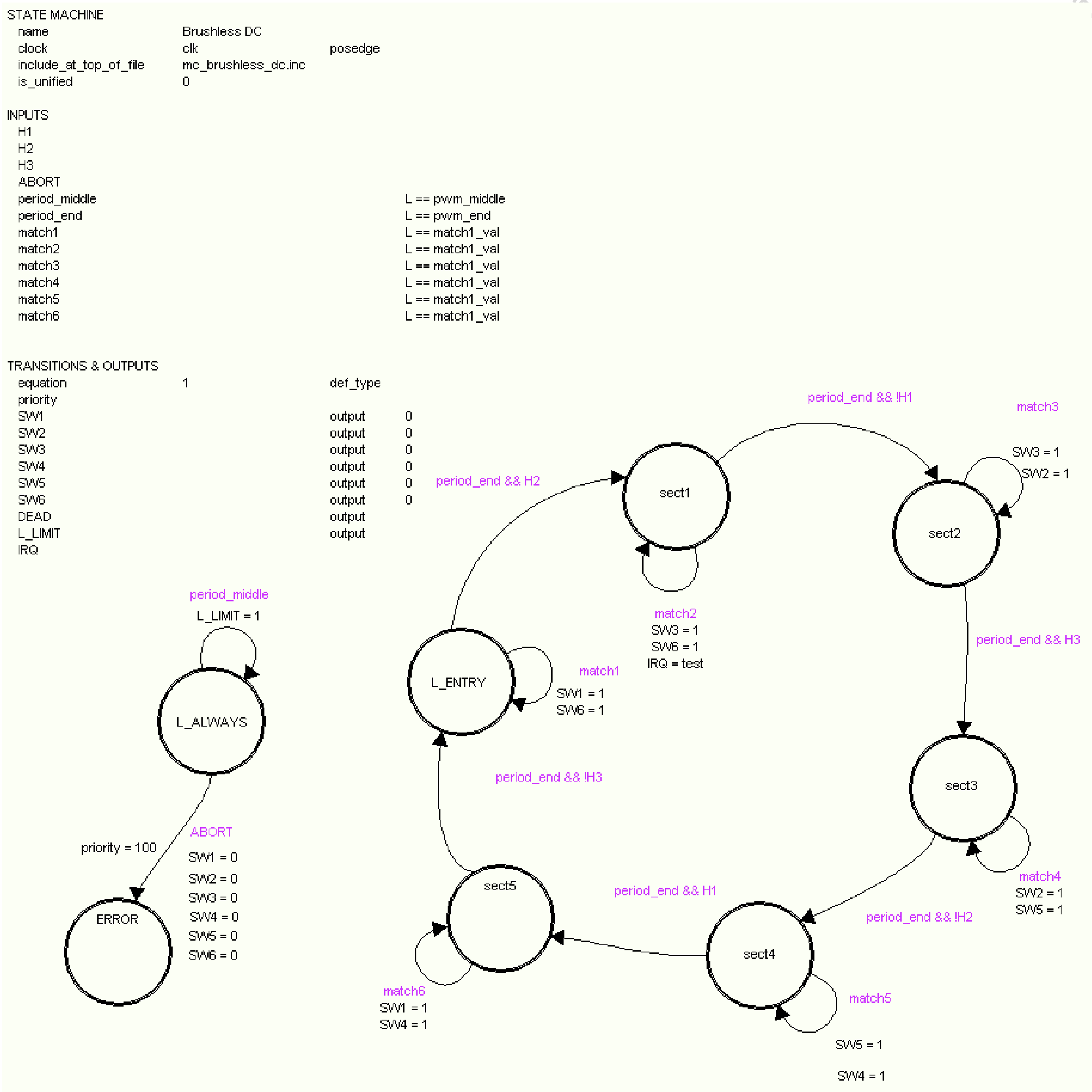
Fig 39. State machine of the light example project

The low frequency signal is generated by the L side of the SCT, and the control of the high frequency PWM signal is done on the H side.

Two external inputs can terminate the high or low phase of the PWM, which are a zero crossing detection and a maximum value for the measured current, see the events *maxoff_MH2* || *+ZCD* and *maxon_MH1* || *IREF*.

4.3.3 Brushless DC Motor control

This example shows a possible implementation of a motor control PWM setup for brushless DC motors



```
STATE MACHINE
  name                  Brushless DC
  clock                 clk              posedge
  include_at_top_of_file mc_brushless_dc.inc
  is_unified            0

INPUTS
  H1
  H2
  H3
  ABORT
  period_middle                                   L == pwm_middle
  period_end                                      L == pwm_end
  match1                                          L == match1_val
  match2                                          L == match1_val
  match3                                          L == match1_val
  match4                                          L == match1_val
  match5                                          L == match1_val
  match6                                          L == match1_val

TRANSITIONS & OUTPUTS
  equation              1                def_type
  priority
  SW1                                    output    0
  SW2                                    output    0
  SW3                                    output    0
  SW4                                    output    0
  SW5                                    output    0
  SW6                                    output    0
  DEAD                                   output
  L_LIMIT                                output
  IRQ
```

Fig 40. State machine of the brushless dc motor control example project

Note: it is outside of the scope of this manual to describe the theory of operation of the brushless dc motor, for more details you might refer to application note AN10898 - BLDC motor control with LPC1700

The location of the motor is assumed to be determined by 3 Hall sensors, the inputs of which are called H1, H2, H3. These sensor inputs are used to make the SCT state machine evolve through the different states being defined, which are tightly related to a specific sector of operation (L_ENTRY, sect1 to sect5)

Each sector (position) of the motor is related to a specific configuration of the hall sensor values. For 3 sensors, there are six possible sectors the motor can be located into, each defining an angle of 60 degrees (where of course a full rotation is done at every 360 degrees).

The PWM signal for a specific set of phases / switches, indicated by SW1 to SW6, is repeatedly generated within each state (sector), when the respective match signal happens (match1 to match6). Each phase is potentially configurable with a different PWM period, although they are all by default pre-initialized to use the same value (match1_val).

When the hall configuration changes, meaning a different sector (position) of the motor has been reached while rotating, the SCT will jump to the next state.

Note the SCT is configured in bidirectional mode in this example, which means the events which set a specific output during the up-counting phase will clear the same output when the same match value is reached during the down-counting phase.

The duration of each PWM half period is determined by the *period_middle* match value, which limits the timer and makes it start counting down.

There is also an ABORT input which will trap the SCT in an error state, keeping all outputs (switches) deactivated. This event might happen at any time, thus is configured as being in the L_ALWAYS pseudo state.

The state machine in question will make the motor turn in a specific direction (let´s say clockwise). For the opposite direction, a complementary setup is provided in the project and can be used to replace the state machine shown in Fig 40

The different hall sensor inputs have been simulated within the application by means of GPIO signals, which follow the same signal sequence as real world sensors. This is the current configuration provided in the main application, although it has been tested also successfully with a real motor.

5. FSM Designer Reference

5.1.1 States

The states which you draw in FSM Designer will be mapped by the tool into a specific state number within the SCT configuration (the optimizing routine built into the tool does not attempt to reduce the number of states in the diagram)

The only attribute of a state is its *name*. The user can choose whatever name for a state, with following exceptions:

- **H_ALWAYS**, **L_ALWAYS** (split counter mode), **U_ALWAYS** (unified counter mode):

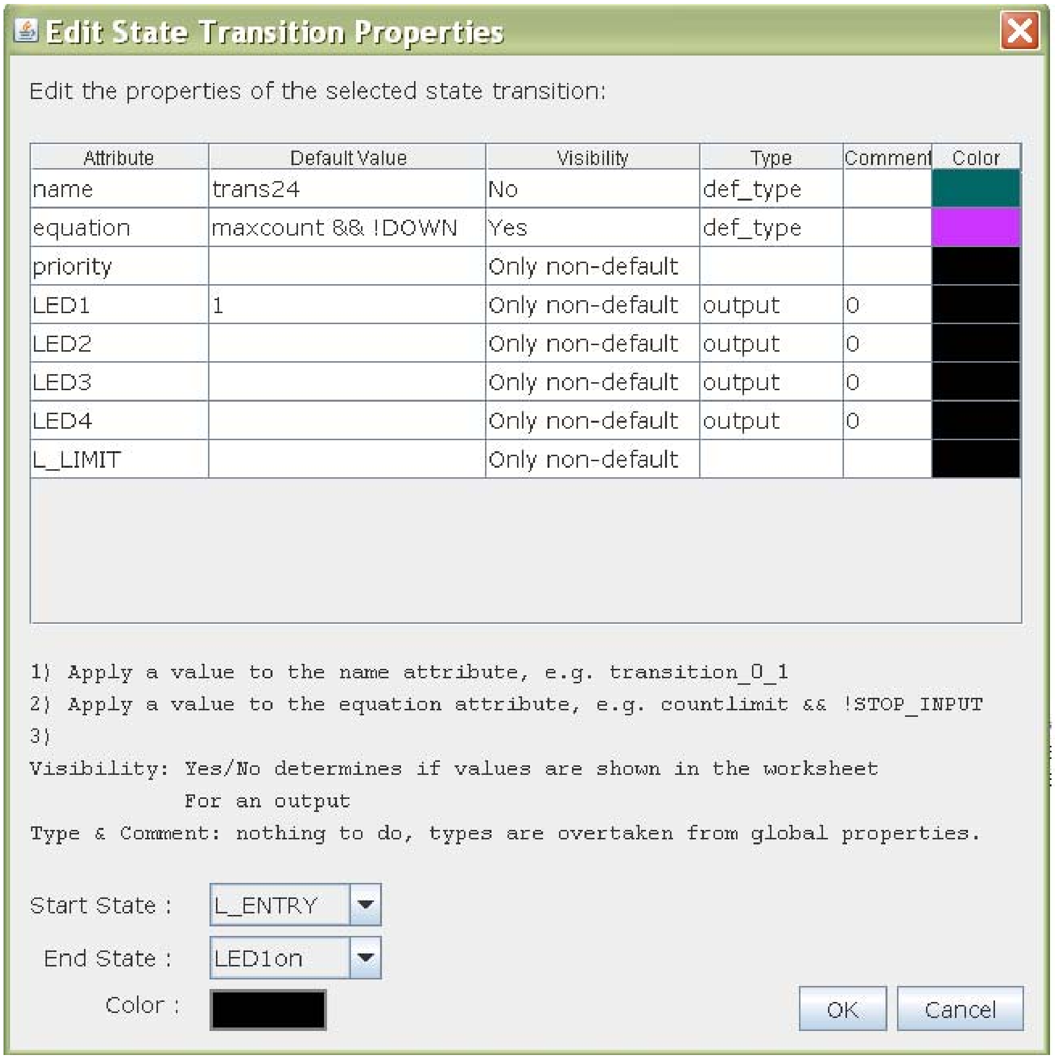
These are pseudo (or “virtual”) states which do not get mapped into a state register value for the SCT state machine. It is just a graphical convenience to represent events which are state independent, or in other words are considered to be valid in all defined states

- **H_ENTRY**, **L_ENTRY** (counter split mode), **U_ENTRY** (unified counter mode)

These represent the initial value of the state register the SCT will have after configuration. It is a useful feature as you might want the state machine to start from a user defined condition. If not specified, the SCT will be left in the default configuration after reset, that is, start from state zero. Note that the tool will map the state numbering at convenience, so use the ENTRY feature if the starting state is of relevance for your application

5.1.2 Events

SCT events are represented by *Transitions* in the Fizzim tool. A *Transition* can be drawn to either connect two different states or to start and end in the same state (this is referenced as a *Loopback Transition)*.



Edit State Transition Properties

Edit the properties of the selected state transition:

Attribute	Default Value	Visibility	Type	Comment	Color
name	trans24	No	def_type		🟦
equation	maxcount && !DOWN	Yes	def_type		🟪
priority		Only non-default			⬛
LED1	1	Only non-default	output	0	⬛
LED2		Only non-default	output	0	⬛
LED3		Only non-default	output	0	⬛
LED4		Only non-default	output	0	⬛
L_LIMIT		Only non-default			⬛

```
1) Apply a value to the name attribute, e.g. transition_0_1
2) Apply a value to the equation attribute, e.g. countlimit && !STOP_INPUT
3)
Visibility: Yes/No determines if values are shown in the worksheet
            For an output
Type & Comment: nothing to do, types are overtaken from global properties.
```

Start State : L_ENTRY

End State : LED1on

Color : ⬛

Fig 41. Event definition

The condition for an event is specified as the *equation* field of the transition properties.

Any actions (to outputs or to the SCT internal control logic) are specified by assigning a value into the *DefaultValue* column of the available output fields, which are listed in the *Attribute* list in the transition's context menu. In the example above, LED1 gets driven to logic 1.

An equation can be a combination of a match condition and I/O condition.

Assuming we have defined a time based match condition *match_name*, an input signal called *input_name,* an output signal called *output_name*, the syntax for an equation can take one of the following formats:

Condition Syntax	Meaning
match_name	Time based match
match_name && input_name	Time based match AND input_name high
match_name && !input_name	Time based match AND input_name low
match_name && +input_name	Time based match AND input_name rising edge
match_name && -input_name	Time based match AND input_name falling edge
match_name || input_name	Time based match OR input_name high
match_name ||!input_name	Time based match OR input_name low
match_name || +input_name	Time based match OR input_name rising edge
match_name || -input_name	Time based match OR input_name falling edge
match_name && output_name	Time based match AND output_name high
match_name && !output_name	Time based match AND output_name low
match_name && +output_name	Time based match AND output_name rising edge
match_name && -output_name	Time based match AND output_name falling edge
match_name || output_name	Time based match OR output_name high
match_name ||!output_name	Time based match OR output_name low
match_name || +output_name	Time based match OR output_name rising edge
match_name || -output_name	Time based match OR output_name falling edge
input_name	Input_name high
!input_name	Input_name low
+input_name	Input_name rising edge
-input_name	Input_name falling edge
output_name	Output_name high
!output_name	Output_name low
+output_name	Output_name rising edge
-output_name	Output_name falling edge

Examples:

Counter reaches **maxvalue** while the input **BUTTON** is low:

```
maxvalue && !BUTTON
```

Either counter reaches **tick** or output **LED** is high:

```
tick || LED
```

Rising edge on input **contact**:

```
+contact
```

Counter reaches value **highend**:

```
Highend
```

One important hint: if you notice the screenshot in Fig 41 the name of the transaction is “trans24”. This is a default name which the graphical tool assigns automatically.

The only restriction when naming transactions is that every name must be unique in the diagram (that´s why fizzim assigns a progressive number by default, to ensure there are no conflicts). However the user can also modify this and give the transaction a custom name.

When changing it from trans24, make sure to choose “Yes” within the *Visibility* column, so that your custom name is displayed onto the diagram.

This will have also another useful effect: the name you have chosen will be visible within the comments included in the generated *sct_fsm.c* file, in the event registers section.

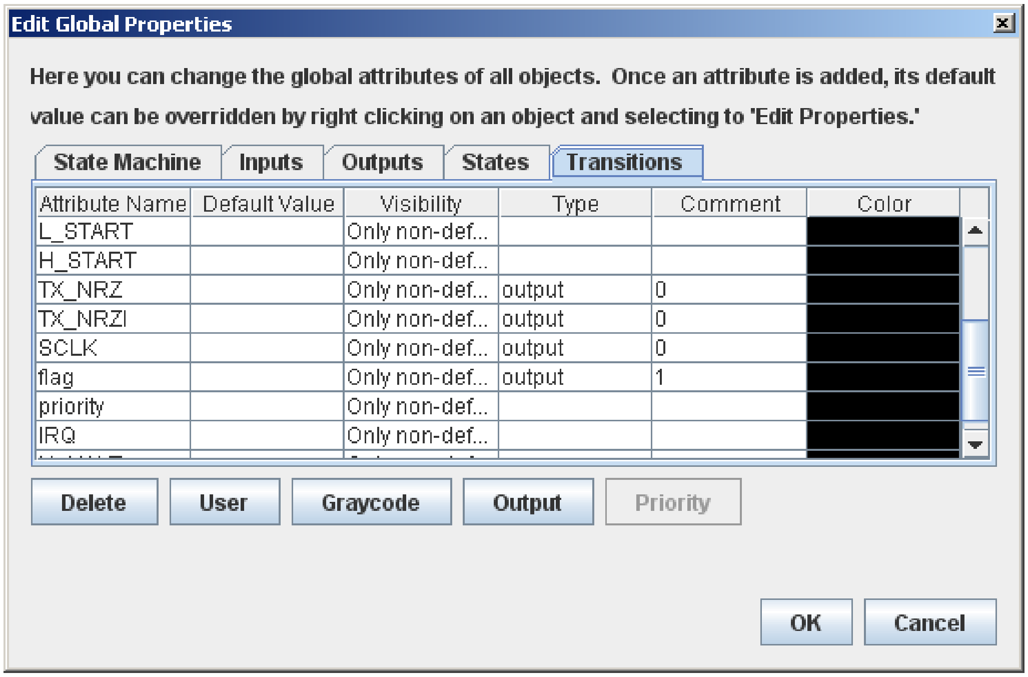
This will make easier to identify to which physical event number your transaction was mapped into by the tool, for example to cross check the generated output, if desired.

5.1.3 Predefined names for pseudo-outputs

In order to have a consistent way to specify interrupts, HALT, STOP and START events, some pseudo signals have been pre-defined within the tool by using reserved keywords.

These signals are specified (together with the physical outputs) within the *Transitions* tab for the “*Global Attributes*” of the current diagram.

As an example, see below a snapshot from the HDLC example project, which uses some of the predefined signals to start and stop the timer, and to generate an interrupt.



Edit Global Properties

Here you can change the global attributes of all objects. Once an attribute is added, its default value can be overridden by right clicking on an object and selecting to 'Edit Properties.'

Attribute Name	Default Value	Visibility	Type	Comment	Color
L_START		Only non-def...			■
H_START		Only non-def...			■
TX_NRZ		Only non-def...	output	0	■
TX_NRZI		Only non-def...	output	0	■
SCLK		Only non-def...	output	0	■
flag		Only non-def...	output	1	■
priority		Only non-def...			■
IRQ		Only non-def...			■

Fig 42. Example of pseudo-outputs

Note: those pseudo signals are used internally by the SCT logic and are not to be confused with the SCT block output pins. The following predefined pseudo signals are defined:

- `L_LIMIT, H_LIMIT (for split mode), U_LIMIT (for unified mode)`

If set to 1, these configure the transition (event) generate a LIMIT condition for the corresponding counter

- `L_START, H_START (for split mode), U_START (for unified mode)`

If set to 1, these configure the transition (event) generate a START condition for the corresponding counter

- `L_STOP, H_STOP (for split mode), U_STOP (for unified mode)`

If set to 1, these configure the transition (event) generate a STOP condition for the corresponding counter

- `IRQ`

If you assign a name to this signal, the corresponding event will trigger an interrupt. As the event number is determined automatically by the tool, the user needs to know this number in order to correctly process the event in the SCT interrupt handler. For example, if the assigned name is “process_done”, the tool will add this definition to the sct_fsm.h header file:

```
#define UT_IRQ_EVENT_process_done (3)
```

In this way the user gets the association between the defined IRQ event and the event index the tool automatically assigned it to (in this case index 3)

- `DMA0, DMA1`

For these pseudo events, setting the value to ‘1’ within the *default value* field of the state transitions properties will generate a DMA request for the channel (DMA0 or 1)

Important note: within the *State Transition* properties, all these pseudo-signals like (**IRQ**, **H_START**, etc.) have an empty *Type* field. This is required by the tools to distinguish them from any other signal defined as a SCT block output, which must have the *Type* field set to *output*.

6. Command line build

The SCT FSM designer tool is the most convenient way for the developer to test specific SCT configurations in a rapid and intuitive manner. Still, it is possible to rebuild an SCT configuration by using a command line environment on both Windows and Linux machines.

The SCT configuration generation process consists of two steps; first the output of the graphical tool is converted from the fizzim format (which is essentially based on XML syntax) to an intermediate text file, which includes all the SCT relevant information for the desired state machine configuration. This is performed by the Perl script fzm2script.pl.

You might have noticed the intermediate file, left in the same folder where the fizzim state machine is located, has by default the extension .smd and the same name as your state machine diagram (without the fizzim extension). It is generated so that you can also double check the configuration in case error messages are output to the screen by the parser during the build process.

This file is given as input to the C code generator, which is responsible for synthesizing the correspondent SCT setup (sct_fsm.c, sct_fsm.h).

A user can also directly generate an SCT configuration by passing this .smd file directly to the fzmparser.exe. The following command lines can be used on Windows and Linux machines:

- `fzmparser.exe < my_sct_config.smd       (on Win)`
- `cat my_sct_config.smd | fzmparser       (on Linux)`

Note: the Linux build is not provided by default, but can be generated from the sources by modifying and using the shell script called *build.sh* which is provided in the src directory

For rebuilding the sources, the *make.bat* dos batch file can also be used, but the following packages need to be installed before on your Windows based computer:

- Complete package of the Windows versions of flex + bison:
  http://gnuwin32.sourceforge.net/packages/bison.htm
  http://gnuwin32.sourceforge.net/packages/flex.htm
- MinGW (includes the GCC compiler):
  http://sourceforge.net/projects/mingw/

Make sure the installation goes into a path that does not contain any SPACE characters! For instance: D:\GnuWin32, D:\MinGW. Also do not forget to add the path to the binaries, like D:\GnuWin32\bin, D:\MinGW\bin to your Windows PATH environment**,** in Windows XP you can find the setting here:

```
Control Panel/System/Advanced/Environment Variables
```

There are also Linux versions available of them. You likely would have to update your user shell environment variables, in case the installation process does not update the user path information automatically (please check the respective sites for availability and installation instructions)

7. Current known limitations

The ultimate goal of the tool is to allow specifying every detail of a complex SCT design.

For the time being, the following limitations are present:

- The SCT global configuration (like the operating mode, the clocks configuration) has still to be specified manually within the application code. All specified match and acquired capture values are relative to those global clock settings
- The conflict resolution register setup is not fully included in the state machine configuration. The default hardware setup after reset is to take no action in case of conflicts for events trying to drive the same output at the same time.

  In the current implementation, it is possible to choose the "toggle" action for those outputs which are set and cleared simultaneously at a specific point in time. This is done by specifying the "pin=toggle" attribute in the FZM drawing.

  In case of multiple events driving the same outputs, the programmer needs to manually modify the generated code to override the current settings and program the conflict resolution register as desired, as this feature is not included in the current tool version (1.x)
- The conflict enable register is also not programmed and needs to be written by the programmer if it is required to trigger a “no change conflict” interrupt
- DMA0 and DMA1 support is potentially included in the tool and the parser, but it has not been extensively tested

.

8. Licenses

8.1 Fizzim

This is free software. Visit http://fizzim.com for details.

8.2 Perl

Currently we use a Perl package from ActiveState (http://www.activestate.com). The “community” version of this software can be used for free for commercial purposes. However, the install file may only be distributed inside your own organization (i.e. NXP). If ever this package will be required by someone outside NXP, please refer them to the download page of ActiveState.

9. Legal information

9.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is for the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

9.3 Licenses

Purchase of NXP <xxx> components
<License statement text>

9.4 Patents

Notice is herewith given that the subject device uses one or more of the following patents and that each of these patents may have corresponding patents in other jurisdictions.

<Patent ID> — owned by <Company name>

9.5 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

<Name> — is a trademark of NXP B.V.

10. Index

11. Contents

Please be aware that important notices concerning this document and the product(s) described herein, have been included in the section 'Legal information'.

For more information, please visit: http://www.nxp.com
For sales office addresses, email to: salesaddresses@nxp.com

Date of release: 25 October 2010

**Document identifier: <DOC_ID>