State Machine Generator User Guide

2020-02-10

# Background

The remote-control application has a lot of moving parts. We need a formal way to describe what should happen (state machine) when various events occur that is easy to understand, can be automatically converted into the code that drives the S3, and can be simulated to get an understanding of behavior prior to implementation.

# Hierarchical State Machine

### Processes

The remote-control application consists of several processes. Ideally these processes should be independent in order to make the system adaptable. Each process will have various states and actions. To illustrate, the basic Remote Control has 11 processes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Type | Purpose | Has State | Has Actions |
| DFS | S/W | Handle power states | Yes | Yes |
| PTT\_Switch | H/W | Handle the PTT switch | Yes | No |
| MuteSwitch | H/W | Handle the Mute switch | Yes | No |
| VM1010 | H/W | Handle the WOS state of Vesper mic | Yes | Yes |
| WOS\_Timer | S/W | Hysteresis to put Vesper in WOS state | Yes | Yes |
| AudioHW | H/W | Handle the audio subsystem | Yes | Yes |
| LPSD | H/W | Handle the LPSD hardware | Yes | Yes |
| CircularBuffer | S/W | Provides a look-back for VR events | Yes | Yes |
| VR | S/W | Scans audio looking for keyword | Yes | Yes |
| HIF | S/W | Tracks interaction with host | Yes | Yes |
| D2H | S/W | Handles data transmission to host | Yes | Yes |

### Process Actions

In order to facilitate automation as well as understanding, every process should have the following function and data structure:

|  |  |
| --- | --- |
| Function Name | Purpose |
| <ProcessName>\_FSMAction(  enum process\_action pa,  void \*parg) | Called by the FSM to perform an action, for example: CONFIG, START, STOP.  The requested action is defined by an enum, and any required data is supplied by the argument. |
| <ProcessName>\_FSMConfigData | Definition of the data element that will be passed to FSMAction() |

The FSM generator assumes these are the names and generates the apaction array with pointers to the appropriate functions, as well as pointers to the arguments.

# Defining Process Characteristics

Each process has its own small state machine. A typical simple process will have the following state machine:

PACTION\_CONFIG

PACTION\_START

PACTION\_STOP

PACTION\_POWER\_DOWN

The ‘ProcessCharacteristics’ file is a text file that is used to describe the high-level characteristics of all the processes to fsmgen. For each process the developer needs to define:

* The name of the process
* The actions that the process recognizes, and what process state results from the actions
* What control events the process can generate

The format for these elements is:

|  |  |
| --- | --- |
| Element |  |
| Process name | Process: <name> |
| Actions and states | Action: <action\_name> => <resulting\_state> |
| Control events | Event: <event\_name> |

Fsmgen expectes to see the names exactly as entered in the state table description.

Fsmgen will use the data in the characteristics file to create three enums in fsm.h:

Enum process\_action, enum process\_state, and enum control event.

The elements of:

* Process\_action are obtained by concatenating “PACTION\_” with the action\_name
* Process\_state are obtained by concatenating “PSTATE\_” with the resulting\_state
* Control\_event are obtained by concatenating “CEVENT\_” with the event\_name

Fsmgen automatically adds process states PSTATE\_UNCONFIG, PSTATE\_DONT\_CARE and PSTATE\_NULL if they are not present; and actions PACTION\_NULL, PACTION\_START and PACTION\_CONFIG if they are not present.

## Example

A portion of the RemoteControl process characteristic file is:

Process: VM1010

Action: CONFIG=>STOPPED

Action: STOP => STOPPED

Action: START => STARTED

Event: VM1010

Process: WOS\_Timer

Action: CONFIG => STOPPED,

Action: STOP => STOPPED

Action: START => STARTED

Event: WOS\_TIMER

Process: AudioHW

Action: CONFIG => STOPPED,

Action: STOP => STOPPED

Action: START\_SAVING => SAVING

Action: START\_STREAMING => STREAMING

This portion defines three processes:

1. VM1010
2. WOS\_Timer
3. AudioHW

### VM1010

The VM1010 has three actions:

1. CONFIG – which is used to setup the GPIO that is used for the wait on sound interrupt
2. START – which is used to start the wait on sound function of the mic
3. STOP – which is used to stop the wait on sound function of the mic

And the VM1010 produces one event:

1. VM1010 – this event is produced whenever the VM1010 triggers a wake on sound interrupt

### WOS\_Timer

The WOS\_Timer has three actions:

1. CONFIG – which is used to create the FreeRTOS timer and set the duration of the timer
2. START – which is used to start the timer
3. STOP – which is used to stop the timer

And the WOS\_Timer produces one event

1. WOS\_Timer – this event is produced whenever the WOS\_Timer expires

### AudioHW

The AudioHW process has four actions:

1. CONFIG – used to configure many parameters (such as mono/stereo, sinc rate)
2. START\_SAVING – clears the 8KB circular buffer and starts the audio writing PCM data to the 8KB circular buffer
3. START\_STREAMING – if the audio was in SAVING mode, this starts the DMA which drains the 8KB circular buffer and then stream incoming data. If the audio was in the STOPPED state this starts the DMA immediately streaming the incoming data
4. STOP – stops the audio subsystem

The AudioHW process produces no events (just data blocks).

The developer is encouraged to look at vm1010.c to see a typical implementation of <process>\_FSMAction().

### State Table

While each process maintains its local state, the state table represents the global state of the system. Thus, the state table has columns for processes and rows for states, event and guards. The semantics are loosely modeled on UML. The actual state table is presented to fsmgen as a CSV file, however, it is convenient to use a full XLS file as the master because the XLS file can use color and cell merging to make the intent clear. When ready, a CSV file can be exported from the XLS file. The example XLS files supplied by QuickLogic use green rows to represent system states, and the white rows immediately below a state represent arcs leaving that state. The event rows are organized into the UML form of event[guards]/actions.

This section of a state table illustrates the features:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | System State | Events | Expected Next State | GUARDS | | | ACTIONS | | |
|  | VM1010 | WOS\_Timer | AudioHW | VM1010 | WOS\_Timer | AudioHW |
| 0 | WAIT\_ON\_KWD |  |  | STOPPED | STOPPED | STREAMING | STOPPED | STOPPED | STREAMING |
| 1 |  | VR\_TRIGGER | WAKING\_HOST1 |  |  |  |  | STOP | START\_STREAMING |
| 2 |  | LPSD\_OFF | WAIT\_ON\_LPSD |  |  |  |  | START | START\_SAVING |
| 3 |  | LPSD\_ON | -- |  |  |  |  |  |  |
| 4 | WAIT\_ON\_LPSD |  |  | STOPPED | STARTED | SAVING | STOPPED | STARTED | SAVING |
| 5 |  | LPSD\_ON | WAIT\_ON\_KWD |  |  |  |  | STOP | START\_STREAMING |
| 6 |  | WOS\_TIMER | WAIT\_ON\_SOUND |  |  |  | START | STOP | STOP |
| 7 |  | LPSD\_OFF | -- |  |  |  |  |  |  |
| 8 | WAIT\_ON\_SOUND |  |  | STARTED | STOPPED | STOPPED | STARTED | STOPPED | STOPPED |
| 9 |  | VM1010 | WAIT\_ON\_LPSD |  |  |  | STOP | START | START\_SAVING |
| 10 | CONFIGURED |  |  | STOPPED | STOPPED | STOPPED | STOPPED | STOPPED | STOPPED |
| 11 |  | START |  |  |  |  | START |  |  |
| 12 | INITIAL |  |  | UNCONFIG | UNCONFIG | UNCONFIG | UNCONFIG | UNCONFIG | UNCONFIG |
| 13 |  | CONFIG |  |  |  |  | CONFIG | CONFIG | CONFIG |
| 14 | //END OF TABLE |  |  |  |  |  |  |  |  |

Starting from the bottom, row 12 is a green row (state row) labeled INITIAL which is the initial state: all three processes are in state UNCONFIG.

The FSM automatically triggers a CONFIG event when it is started. Row 13 controls what happens when a CONFIG event occurs while the FSM is in the INITIAL state. Row 13 specifies CONFIG actions from all three processes, the result of which will be that all three processes enter their STOPPED state. Row 10 matches the condition where all 3 are stopped, so the FSM will advance to the CONFIGURED state. The FSM also automatically triggers a START event immediately after the CONFIG event. Row 11 controls what happens when a START event occurs while the FSM is in the CONFIGURED state. Row 11 specifies that a START action will be sent to the VM1010 process. This action causes the VM1010 to start monitoring for sound. It also changes the VM1010 to the STARTED state. Row 8 matches the condition where VM1010 is STARTED and the other 2 are STOPPED, so the FSM will advance to the WAIT\_ON\_SOUND state.

When the VM1010 detects sound, it will trigger an interrupt that generates a VM1010 event. Row 9 specifies that a VM1010 event in the WAIT\_ON\_SOUND state results in a STOP action sent to VM1010, START action sent to WOS\_Timer and START\_SAVING action sent to the AudioHW. This will move the FSM to Row 4, WAIT\_ON\_LPSD. If the WOS\_Timer expires it will send a WOS\_Timer event, and if the FSM is still in WAIT\_ON\_LPSD (row 4), the WOS\_Timer event (Row 6) will send a START to the VM1010 telling it start looking for sound, a STOP to WOS\_Timer telling it to stop the timer, and a STOP to the AudioHW telling it to stop listening.

# Simulation Script

The simulation script is a text file that lists a series of events. Fsmgen will apply these events to the FSM and report the consequences. The file format is simply one event per line, each line has the syntax:

EVENT,<event\_name>

Example:

EVENT,CONFIG

EVENT,START

EVENT,VR\_TRIGGER

Please be aware that the parser is brutally simple, so it won’t tolerate extra white space.

# Executing fsmgen

Usage: fsmgen --state-machine file.csv

--device-description file.txt

--simulation-script file.txt

--target-dir dir

The only non-obvious argument is ‘--target-dir dir’. Without this parameter, fsmgen writes the three output files, fsm.c, fsm.h and fsm\_cli\_table.c in the execution directory. Target dir allows you to specify the path to where the three files should be written.

# Observations

The idea behind the design is that all global interactions are captured in the control process, so that individual processes do not directly depend on the state of other processes.

Careful use of states can maintain some history without making the state table too complicated. For instance, in the RCStates FSM, the D2H process has a notion of BEGIN\_STOP that puts D2H in the STOPPING state, and when D2H has transmitted the last packet, the FINISH\_STOP action changes the state to STOPPED.

|  |  |  |
| --- | --- | --- |
| Version | Date | Comments |
| V1 | 2020-02-10 | Initial version |
| V2 | 2020-03-01 | Major revisions to ProcessDescription |