V. Marolda February 2011

thinStates C Compiler for StateWORKS

Introduction

The thinStates C Compiler for StateWORKS offers a “thin” way to Finite State Machine software design with the StateWORKS SWStudio.

The generated C source code, models the State Machines' logic and interconnections with the other objects.

At the moment, there is no “standard library” for all the external objects (e.g. Counters, Timers etc), but an example is given for the very basic useful objects: Commands, Timers, Digital Inputs, Digital Outputs, Output Functions.

The lack of such a standard library also means there is no standard configuration support for the external objects: for example, in no way the “interval” setting of timers is reflected in the generated C code.

Generation Technique

The key concept in thinStates code generation, is the generation of a .c file for *each* VFSM instance.

For example, if we define just only one State Machine in SWStudio (e.g. *Button*) and then we instantiate it two times (e.g. *Button_1* and *Button_2*), we get two .c files: `swc_vfsm_Button_1.c` and `swc_vfsm_Button_2.c`.

In fact, the two .c files look very similar, but the object names they refer to are different: each one *statically* refers to the objects that *that* instance is connected to, in SWStudio.

In our experience, this approach produces the most human-friendly (and fast, as well) source code: it is simple to read, and simple to debug: we can put a breakpoint when a *particular instance* of state machine changes its state, and inspect the relevant variables if needed, to understand an issue that happens on a single instance.

Generated Modules

For each VFSM instance:

- `swc_vfsm_`*`name`*`.c`: contains the logic (the state functions) for the vfsm instance, *statically* linked to the referred objects, and instantiates the state index variable.

For each VFSM definition:

- `swc_vfsm_`*`name`*`.h`: where *name* is the name of the VFSM definition, contains:
  - the enumeration of all the states' number, starting from 1, numbered as they are in the SWStudio VFSM editor;
  - the externs for the logic invocation and state index variable of all the instances of this definition.

Always generated:

- `swc_config.h`: includes a custom header file `app_swc_config.h` and all the vfsm include (.h) files.
- `swc_vfsm.c`: provides two functions that invoke *all* the VFSM instances in the system:
  - `uint8_t swc_vfsm_Execute_All_I()`: executes all the *actions* for all the state machines, and always returns 0.
  - `uint8_t swc_vfsm_Execute_All_T()`: executes all the *transitions* for all the state machines, and returns 1 if one has changed state.

Example: lpc_p2148_example

The example runs on a development board from Olimex: the LPC-P2148, which is based on an NXP LPC2148 (arm7-based) microcontroller.

The board provides the following i/o that are used in our project:

- Two buttons

- One potentiometer
- Two leds
- One buzzer

The example project in SWStudio defines the following:

- Two Digital Inputs:
  - Di_button:001
  - Di_button:002
- Three Digital Outputs:
  - Do_buzzer
  - Do_led:001
  - Do_led:002
- Two Commands for the state machines:
  - Cmd__dummy (for the state machines that we don't need to command)
  - Cmd_Flasher_Buzzer (for the Flasher_Buzzer state machine instance)
- Two Output Functions:
  - OFun_readPotPosition (divides the pot area in three slots numbered 0,1,2)
  - OFun_updateBuzzerTimer (changes the buzzer on/off period when the buzzer is activated)
- Three Timers:
  - Ti_button:001 (debouncer for button 1)
  - Ti_button:002 (debouncer for button 2)
  - Ti_buzzer (defines the on/off period for buzzer activation)
- Three VFSM Definitions:
  - Button: changes its state from NotPressed to Pressed and vice-versa, reading the button's digital input and using the timer to perform debouncing. It does not need commands: it always runs.
  - Flasher: alternates on/off on a digital output, using a timer. It responds to two commands: start and stop.
  - Main: it is the main controller: in the *Always* state, it inspects the state of the two buttons' state machine instances, and turns leds 1 and 2 on if the state is “Pressed”, off is the state is “Not Pressed”.
    In the state logic, it Activates the buzzer if both buttons are “NotPressed”, after both buttons have been “Pressed” and the pot is in central position.
    During buzzer activation, it invokes the Ofun_updateBuzzerTimer to change the on/off period.
    Also, if the user presses both buttons and then releases them, the state machine returns to the initial state.

Now let's skip SWStudio usage explanation (there are other docs for this), and let's give a look at the generated code.

Generated Code for Button_1 instance

Let's immediately look at the state machine *Button_1*, which gets compiled in `swc_vfsm_button_1.c`.

We can divide the file in the following parts:

Includes

```
#include "swc_config.h"
```

Just one include here: `swc_config.h`, which contains the useful definitions of the whole StateWORKS system, plus – through the user-supplied `app_swc_config.h` - the needed realworld's definitions.

Instantiation of the state index variable, and new state index variable

```
swc_vfsm_stateidx_t swc_vfsm_Button_1_stateIdx = 1; //Init state
swc_vfsm_stateidx_t swc_vfsm_Button_1_stateIdx_new = 1; //Init state
```

The global state index variable is `swc_vfsm_Button_1_stateIdx`. This variable will always contain the state number in which the state machine is. It is defined as starting from 1, which is the number of the "Init" state. In fact, the number that will be here will always be the same number that SWStudio displays for the states.

The global state index variable is not modified during the transitions. Instead, `swc_vfsm_Button_1_stateIdx_new` is updated. The change will be reflected to `swc_vfsm_Button_1_stateIdx`, only after *all* state transitions will be calculated.

This will provide state index stability during the transitions of all the VFSMs, regardless of the order of execution of the transitions' code.

Both variables are of type `swc_vfsm_stateidx_t`, so that the user can choose the preferred type for it (`unsigned char` will be good if we will have a maximum of 255 states in our largest state machine, for example).

Definition of Virtual Inputs

```
#define VI_Ti_button_OVER() \
      SWC_TI_readVI(Ti_button_001,SWC_TI_VI_OVER)
#define VI_Di_button_notPressed() \
      SWC_DI_readVI(Di_button_001,SWC_DI_VI_LOW)
#define VI_Di_button_pressed() \
      SWC_DI_readVI(Di_button_001,SWC_DI_VI_HIGH)
```

In this section, each #define defines a name that is composed as VI_*name*, where *name* is the name of the Virtual Input, as it is defined in the Inputs table in SWStudio. This name, that is defined in the VFSM editor, will be the same for both the instances Button_1 and Button_2, but the two translations will be different.

In fact, it is in the translations that we perform the *static link* of this VFSM instance with the connected objects, which – for the vitrual inputs – are: `Ti_button_001,Di_button_001`. Please note that the : in object names, is always translated to an underscore.

The translations in turn invoke *object-aware* macros (`SWC_TI_readVI, SWC_DI_readVI`), which have the responsibility to return `TRUE (1)` if *the object instance* whose name is specified as first argument, is in the *state* that's specified as second argument.

Definition of Virtual Outputs

```
#define OA_Ti_button_Reset() \
      SWC_TI_performOA(Ti_button_001,SWC_TI_OA_Reset)
#define OA_Ti_button_Start() \
      SWC_TI_performOA(Ti_button_001,SWC_TI_OA_Start)
```

The concept here is very similar to the Virtual Input definition: let's just show the differences

- The names that are defined begin with OA (Output Action) instead of VI (Virtual Input).
- The macros that are invoked end with *performOA* instead of *readVI*.

The invoked *object-aware* macros have the responsibility to *perform*, on *the object instance* whose name is specified as first argument, the *action* that's specified as second argument.

The Always state

```
static void Always() {
}
```

In *Button* VFSM, there is nothing in the Always state. Since it is declared as *static,* this should easily optimized away by the compiler.

The state functions prototypes

```
static uint8_t State_Init_e();
static uint8_t State_Init_x();
static uint8_t State_Init_i();
static uint8_t State_Init_t();

static uint8_t State_NotPressed_e();
static uint8_t State_NotPressed_x();
static uint8_t State_NotPressed_i();
static uint8_t State_NotPressed_t();

static uint8_t State_Pressed_e();
static uint8_t State_Pressed_x();
static uint8_t State_Pressed_i();
static uint8_t State_Pressed_t();
```

In the *Button* VFSM, there are only 3 states: `Init, NotPressed, Pressed`.

For each, whe find here 4 functions, which are composed by `State_` prefix, then the state name, then one of the 4 suffixes in the following list:

- `_e`: contains the *entry* output actions of the state;
- `_x`: contains the *exit* output actions of the state;
- `_i`: contains the *conditional actions* that the state performs while it is active.
- `_t`: contains the *conditional transitions* that the state examines while it is active, to advance the state machine to one of the successor states.

Please note that all the state functions are declared as *static*, in order to have just a module-local scope, which avoids unwanted name collisions, and allows compiler optimizations (mainly automatic inclusion) also.

The state function implementations

```
static uint8_t State_Init_e() {
	swc_vfsm_Button_1_stateIdx_new = BUT_State_Init;
	return 0;
}
static uint8_t State_Init_x() {
	return 0;
}
static uint8_t State_Init_i() {
	Always();
	return 0;
}
static uint8_t State_Init_t() {
	if (1){
		State_Init_x();
		State_NotPressed_e();
		return 1;
	}
	return 0;
```

```
}

static uint8_t State_NotPressed_e() {
      OA_Ti_button_Reset();
      swc_vfsm_Button_1_stateIdx_new = BUT_State_NotPressed;
      return 0;
}
static uint8_t State_NotPressed_x() {
      return 0;
}
static uint8_t State_NotPressed_i() {
      Always();
      if (VI_Di_button_notPressed()){
            OA_Ti_button_Reset();
      }
      if (VI_Di_button_pressed()){
            OA_Ti_button_Start();
      }
      return 0;
}
static uint8_t State_NotPressed_t() {
      if (VI_Ti_button_OVER()){
            State_NotPressed_x();
            State_Pressed_e();
            return 1;
      }
      return 0;
}

static uint8_t State_Pressed_e() {
      OA_Ti_button_Reset();
      swc_vfsm_Button_1_stateIdx_new = BUT_State_Pressed;
      return 0;
}
static uint8_t State_Pressed_x() {
      return 0;
}
static uint8_t State_Pressed_i() {
      Always();
      if (VI_Di_button_pressed()){
            OA_Ti_button_Reset();
      }
      if (VI_Di_button_notPressed()){
            OA_Ti_button_Start();
      }
      return 0;
}
static uint8_t State_Pressed_t() {
      if (VI_Ti_button_OVER()){
            State_Pressed_x();
            State_NotPressed_e();
            return 1;
      }
      return 0;
}
```

We can note the following details:

- It is the *entry* function of a state that updates the state index variable, setting the new state number (which is *always* not-zero), in the variable `swc_vfsm_Button_1_stateIdx_new`.
- The *conditional actions* function of every state calls the *Always* function.
- The *conditional transitions* function returns 0 if no transition has occurred.

The state function vector and the execution functions

```
static swc_state_fn_t * const swc_sfv[][2] = {
      {State_Init_i,State_Init_t},
      {State_NotPressed_i,State_NotPressed_t},
      {State_Pressed_i,State_Pressed_t},
};
uint8_t swc_vfsm_Button_1_Execute_I(){
      return swc_sfv[swc_vfsm_Button_1_stateIdx-1][0]();
}
uint8_t swc_vfsm_Button_1_Execute_T(){
      return swc_sfv[swc_vfsm_Button_1_stateIdx-1][1]();
}
```

The state function vector directs, with a constant-time cost, the execution calls to the appropriate state function, with a bi-dimensional array: the first index is the state index, the second index is for *I* or *T* function address.

The function `swc_vfsm_Button_1_Execute_I()` will execute the actual state's *conditional actions* piece of logic, and will always return 0.

The function `swc_vfsm_Button_1_Execute_T()` will execute the actusl state' *conditional transition* logic, and – if a transition occurs – exit function of this state and entry function of next state will be executed, and 1 will be returned.

If no transitions will occur, the function will return zero.

Generated code for Button State Machine

The *Button* state machine, which provides the two instances *Button_1* and *Button_2*, also generates some code: it is a *header file*: `swc_vfsm_button.h`.

Let's look at the generated code:

```
#ifndef _SWC_VFSM_BUTTON_H_INCLUDED_
#define _SWC_VFSM_BUTTON_H_INCLUDED_

enum {
      BUT_State_Init = 1, //Init state has value 1
      BUT_State_NotPressed,
      BUT_State_Pressed,

      BUT_State__NUMBER
};


extern swc_vfsm_stateidx_t swc_vfsm_Button_1_stateIdx;
extern swc_vfsm_stateidx_t swc_vfsm_Button_1_stateIdx_new;
extern uint8_t swc_vfsm_Button_1_Execute_I();
extern uint8_t swc_vfsm_Button_1_Execute_T();
extern swc_vfsm_stateidx_t swc_vfsm_Button_2_stateIdx;
extern swc_vfsm_stateidx_t swc_vfsm_Button_2_stateIdx_new;
extern uint8_t swc_vfsm_Button_2_Execute_I();
extern uint8_t swc_vfsm_Button_2_Execute_T();

#endif //_SWC_VFSM_BUTTON_H_INCLUDED_
```

There are the single-inclusion safeguard macros, then:

- enumeration of all the states, with *BUT* prefix: this is the unique prefix that is specified in SWStudio, which belongs to the Button state machine, with the first being numbered as 1, and a final constant `BUT_State__NUMBER`, which can be useful if we would like to dimension an array, for example.
- externs for all the instances of Button (Button_1 and Button_2): state index variables (actual and new), and execution functions.

Generated code for the whole State Machine system

This is `swc_vfsm.c`:

```
#include "swc_config.h"

uint8_t swc_vfsm_Execute_All_I(){
      uint8_t y=0;
      y = y || swc_vfsm_Flasher_Buzzer_Execute_I();
      y = y || swc_vfsm_Button_1_Execute_I();
      y = y || swc_vfsm_Button_2_Execute_I();
      y = y || swc_vfsm_Main_001_Execute_I();
      return y;
}

uint8_t swc_vfsm_Execute_All_T(){
      uint8_t y=0;
      y = y || swc_vfsm_Flasher_Buzzer_Execute_T();
      y = y || swc_vfsm_Button_1_Execute_T();
      y = y || swc_vfsm_Button_2_Execute_T();
      y = y || swc_vfsm_Main_001_Execute_T();
      //now, if at least one has changed state, let's commit all
      if(y){
            swc_vfsm_Flasher_Buzzer_stateIdx =
swc_vfsm_Flasher_Buzzer_stateIdx_new;
            swc_vfsm_Button_1_stateIdx = swc_vfsm_Button_1_stateIdx_new;
            swc_vfsm_Button_2_stateIdx = swc_vfsm_Button_2_stateIdx_new;
            swc_vfsm_Main_001_stateIdx = swc_vfsm_Main_001_stateIdx_new;
      }
      return y;
}
```

There is the normal include for the whole system definitions, then the two execution functions that operate on *all* the state machine instances.

The function `swc_vfsm_Execute_All_I()` will always return zero.

The function `swc_vfsm_Execute_All_T()` will return one if at least one state machine instance has changed state, and – if this is the case – it will commit all the *_new* state index variables to the corresponding actual state index variables.

In order to make the whole state machine system work, these are the only functions that we need to call.

Generated code for the whole system configuration

This is a header file: `swc_config.h`:

```
#ifndef _SWC_CONFIG_H_INCLUDED_
#define _SWC_CONFIG_H_INCLUDED_

//put in app_swc_config.h all the system includes
#include "app_swc_config.h"

typedef uint8_t (swc_state_fn_t)();

//all VFSM's includes
#include "swc_vfsm_flasher.h"
#include "swc_vfsm_button.h"
#include "swc_vfsm_main.h"

//the global invocation points for all the VFSM's executors
extern uint8_t swc_vfsm_Execute_All_I();
extern uint8_t swc_vfsm_Execute_All_T();
```

```
#endif //_SWC_CONFIG_H_INCLUDED_
```

It immediately includes `app_swc_config.h`, which must contain the user's system definitions, then defines the state function type, then includes all the vfsm's include files, then it declares as `extern`'s the global invocation points (implemented in `swc_vfsm.c`).

User code

Now that we have seen what the thoinStates compiler produces, it's time to look at the code that we supplied in the example, to complete the system.

We supplied the following:

- `app_swc_config.h`: the definitions that the generated code needs to get compiled
- `app_swc_cmd.c/.h`: our implementation of CMD (state machine command) objects
- `app_swc_di.c/.h`: our implementation of DI (digital input) objects
- `app_swc_do.c/.h`: our implementation of DO (digital output) objects
- `app_swc_ofun.c/.h`: our implementation of OFUN (output function) objects
- `app_swc_timer.c/.h`: our implementation of TI (timer) objects
- `main.c`: application entry point, and execution loop.

Let's see the most important aspects, starting from the main:

```
int main(void){
  lowLevelInit();

  swc_configuration();

  for(;;){
    //housekeeping
    advanceTimer(); //let's matain milliseconds variable

    //StateWORKS!!!
    swc_timer_advanceTimers(milliseconds); //timers
    updateDis(); //reading of REAL world's input to supply to VFSMs
    swc_vfsm_Execute_All_I(); //execution of ALWAYS and actual states
conditional ACTIONS
    commitCmds(); //they could have issued some COMMANDS: let's now
reflect all of them in the inputs
    swc_vfsm_Execute_All_T(); //execution of actual states conditional
TRANSITION, and, if transition occurs,
                              //Exit Actions of actual state and Entry
Actions of new state
    commitCmds(); //through exit/entry actions, they could have issued
some COMMANDS: let's now reflect all of them in the inputs
    updateDos(); //writing of VFSMs calculated Do's to the REAL world
  }
}
```

Initialization

Before entering the main loop, it initializes the board, then configures the objects: the timers are the only objects that need to be initialized.

The implementation we provided for timers exports a function: `swc_timer_setMatchValue(int idx, unsigned int value)`, which serves this purpose, and it is called in the main's `swc_configuration()` function:

```
static void swc_configuration(){
  swc_timer_setMatchValue(Ti_button_001,500);
  swc_timer_setMatchValue(Ti_button_002,500);
  swc_timer_setMatchValue(Ti_buzzer,250); //this will anyway be changed
by the ofun, when the buzzer will be activated
}
```

Our timer implementation works in milliseconds.

Main loop

Here the system runs.

In the main module, a rolling milliseconds time-base is kept updated by the function `advanceTimer()`.

Then, the StateWORKS timers (implementation provided in `app_swc_timer.c/.h`) are updated by the function `swc_timer_advanceTimers(milliseconds)`.

Then, a copy of the system's digital inputs is taken in memory by the function `updateDis()`.

Now, the memory contains the fresh information that the state machines need, in order to perform their tasks.

The actual states are now invoked to perform their conditional actions (and *Always* state also). No transitions for now. We will let them examine the system and perform their actions, calling the function `swc_vfsm_Execute_All_I()`.

During the call, *all* the state machine instances will perform some conditional output actions.

In particular, they could have performed some actions on the COMMAND objects, to give commands to the other state machines.

The COMMAND objects are implemented in a way that separates output variables from input variables, so that actions on command are *not* immediately visible, but a *commit* must be called to the command implementation module, which performs the copy.

So, after giving the chance for action to all the state machines, it is time to commit their changes in a single phase with the function `commitCmds()`.

Then, the state machines are invoked to perform their transitions, if any, with the function `swc_vfsm_Execute_All_T()`.

We already know that this function automatically commit state changes after all the transitions.

But since during transitions also output actions happen (through exit and entry commands), maybe some command has been changed also.

So let's commit these changes again, calling `commitCmds()`, in order to have these changes visible at the next loop interaction.

At the end, calling `updateDos()`, transfers the digital outputs memory image to the hardware.

Note on commits

We will further explain why there is the need to *commit* for commands and state index.

The basic concept is to keep invariant the behavior of the state machines, regardless of the execution order.

Let's see for example what happens when we invoke the global transition function:

```
uint8_t swc_vfsm_Execute_All_T(){
      uint8_t y=0;
      y = y || swc_vfsm_Flasher_Buzzer_Execute_T();
      y = y || swc_vfsm_Button_1_Execute_T();
      y = y || swc_vfsm_Button_2_Execute_T();
      y = y || swc_vfsm_Main_001_Execute_T();
      //now, if at least one has changed state, let's commit all
      if(y){
            swc_vfsm_Flasher_Buzzer_stateIdx =
```

```
swc_vfsm_Flasher_Buzzer_stateIdx_new;
            swc_vfsm_Button_1_stateIdx = swc_vfsm_Button_1_stateIdx_new;
            swc_vfsm_Button_2_stateIdx = swc_vfsm_Button_2_stateIdx_new;
            swc_vfsm_Main_001_stateIdx = swc_vfsm_Main_001_stateIdx_new;
      }
      return y;
}
```

The vfsms are called in a particular sequence. So, if *any* output action operation from a state machine was made *immediately* visible as input to all the other state machines, changing the execution order would change the behavior of the system.

Instead, keeping the *important* inputs constant during the round, the execution order will not change the system behavior.

In SWStudio, there is no way to control the execution order: all should behave as if it was running simultaneously.

Separating the outputs and the inputs by a *commit* phase, we can guarantee that the system will always run in a consistent manner, regardless of the execution order.

But which objects should be considered *important*? They are the objects that are *shared* across vfsms, so primarily *state indexes* and *commands*.

As a general rule, other objects should not be shared.

But if they need to be shared for a particular purpose, care should be taken to analyze if the separation-commit algorithm should be implemented.

Conclusions

The thinStates C Compiler for StateWORKS brings the StateWORKS' idea of an executable specification, to the microcontroller level.

After properly modeling the “abstraction layer” between the microcontroller board resources and the used StateWORKS objects, the user will be able to execute the specification with a minimal amount of memory.

The generated code is very human-readable, and it is easy to debug, because of the instance-to-c-file translation strategy.