Contents

1 Introduction

The creation of state machines is shown in the Yakindu documents in detail. However, what is left is how these state machine models can be used to create source code. The purpose of this document is, to give an overview of how the code is created and how the created code can be integrated into existing projects.

The C source code generator, shipped with the yakindu release, is optimized for small embedded systems with certain restrictions, like small RAM/ROM, ANSI-C restictions and MISRA rules. These restictions are mendatory for many tasks e.g. in the automotive area.

Actually the **Yakindu C source code generator** is under heavy development as the other Yakindu features, too. So the interfaces are not fixed yet and will probably change in the near future.

This document guides through an example scenario on an **Display3000** development board. It uses an Actmel ATMega128 CPU, 128 kByte Flash and 4 kByte RAM.

1.1 Extracting the Example Scenario

Please use the Eclipse import functionality to extract the example scanario into your workspace. The Scenario is ship in the ZIP container **Traffic_Light.zip**.

When the files are extracted, your directory looks like follows:

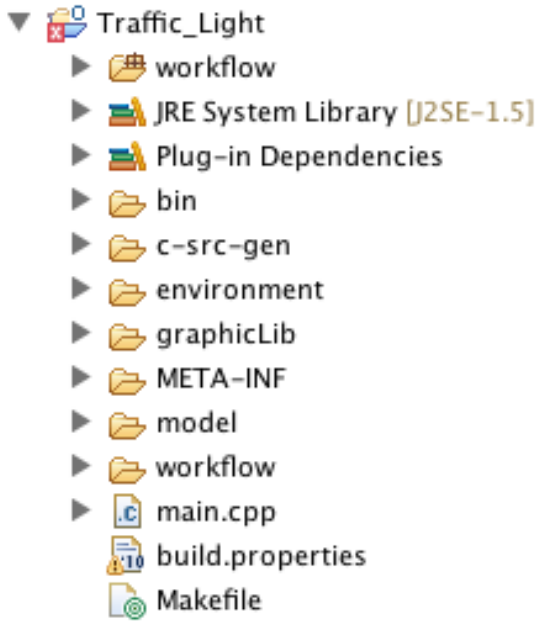

Figure 1: Workflow Tree of the Traffic Light Example

2 Example Scenario

The example scenario is a simple pedestrian traffic light. A pedestrian can press a button to indicate, she/he wants to cross the street. Then a blinking white light indicates, that the traffic light has recognized the request. After a few seconds, the traffic light for the street turns to red and the pedestrian traffic light turns to green. Then the pedestrian traffic light turns to red and the street traffic light changes to green again.

The state machine to model this behaviour is shown on figure 2. This state machine must not be created but is shipped with the exmple project and can be found in the `workspace/Traffic_Light/model` directory.

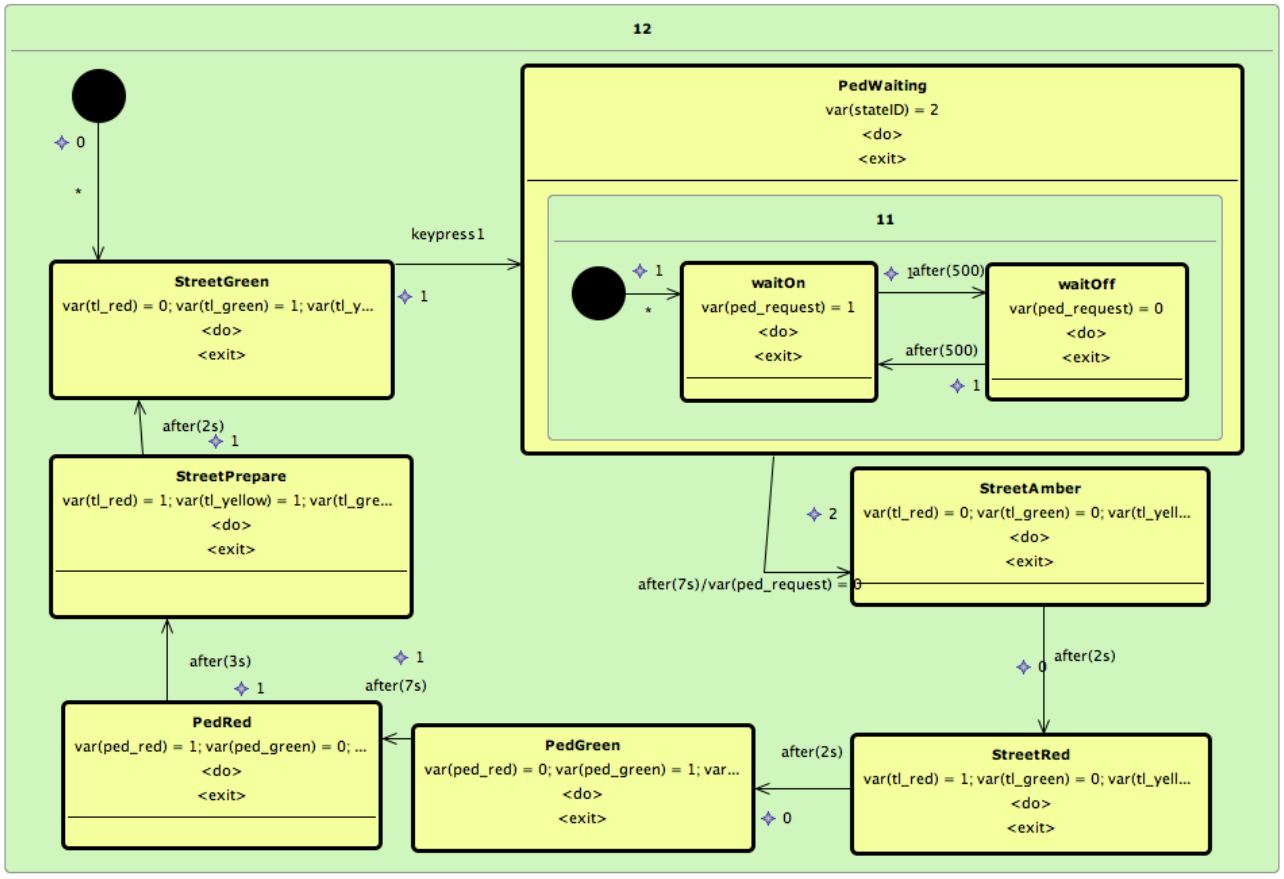

Figure 2: Statemachine for the Traffic Light Example

The embedded system that is used with this scenario is a ATMega128 Controller on a Display3000 board. This system was extended by an additional board representing a traffic light controlled crossroad. This board is connected to the Display3000 board.

3 Running the Workflow and Compilation

Hier fehlt noch was: wie wird der Workflow gestartet, und was muss alles vorhanden sein, damit der workflow tatsaechlich starten kann.

The system is designed for the Display3000 board, so the **Makefile** was designed to create a binary for this hardware. Therefor it requires the AVR-gcc toolchain to be able to compile. The toolchain contains a compiler, a linker and some other usefull tools e.g. to upload the data.

So go to your workspace directory and from her into the base directory of your project and call `make`:

```
workspace$ cd Traffic_Light
Traffic_Light$ make
[...] compiling [...]
avr-gcc -Wall -Os -DF_CPU=7456000 -mmcu=atmega128 -I. -I./graphicLib [...]
rm -f main.hex
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

If the compile was successfull, you have a working binary `main.hex` that could be deployed on the target.

To transfer the binary, you can use `make flash`.

4 State Machine Interfaces

The code for the state machine can be found in the folder **c-src-gen**. After running the workflow the following files should be available in this directory:

```
c-src-gen$ ls
make.include  region12.h        statesDefinition.c
region11.c    sm_interface.c    statesDefinition.h
region11.h    sm_interface.h
region12.c    startdef.h
```

The files completely defines the state machine. To integrate the source into another project, the `make.include` file needs to be included into the original makefile. It contains the files, that needs to be compiled.

The interface functions can be found in `sm_interface.h`. It enables the communication between the status machine and the surrounding environment. Here you can find the Trigger types and the variable types, which are used within the status machine. To access the triggers and variable, the following functions are defined:

```
extern void smi_raiseTrigger(SMInterfaceHandle* handle, TriggerType type);
extern BOOL smi_isTriggerRaised(SMInterfaceHandle* handle, TriggerType type);
extern void smi_resetTrigger(SMInterfaceHandle* handle, TriggerType type);
extern void smi_cleanTriggers(SMInterfaceHandle* handle);
```

```
extern void smi_setVariable(SMInterfaceHandle* handle, VariableType type,
                            VARIABLE value);
extern VARIABLE smi_getVariable(SMInterfaceHandle* handle, VariableType type);
extern void smi_cleanVariables(SMInterfaceHandle* handle);
extern VARIABLE* smi_refVariable(SMInterfaceHandle* handle, VariableType type);
```

The `SMInterfaceHandle` carries the interface information for this state machine. If there is more than one state machine instance. Every state machine instance needs it's own interface handle. The handle is initialized by the call to `smi_initInterfaceHandle()`. As the state machine system should work without a working heap, there is no dynamical allocated data used in this call. So the handle can completely reside on the stack or in the static area.

The state machine itself is represented by a `RegionHandle`. This handle needs to be initialized with the call to `initRegionHandle()`. The second argument of this call is the interface handle, mentioned above. This handle does also not allocate any memory on the heap. The function prototypes can be found in the header file `startdef.h`.

The state machine is time triggered and is designed to work with cooperative operation systems. This means every state machine cycle must be triggered explicitly by calling `region_loop()`.

What is mendatory to use the system is a `sys.h` provided by the system, that contains a function to return the a absolute time reference for the `after()`-expression:

```
extern uint64 time();
```

This file is needed by some state machine header files.

5 Operating System and Drivers for the Display3000

To let the state machine run on the Display 3000 development board, the system needs a minimal **operating system** (OS) and some drivers for input and output. This operating system is found in the `environment` directory. This cooperative OS is written is C++ and contains an input driver for the 6 keys on the hardware board and an output driver for the display and an LED board that is connected to the *port A*.

Because of copyright restrictions, the display driver is not included into this example. the calls to the display interface can included by adding `-DWITH_DP3000_GL` to the compiler options.

The transfer of the LED data uses a simple software driven SPI-like interface with three connections (data, clock and inherit).

The rest, like shifting the data, is done by the hardware board.

Following files belong to the operating system:

```
definitions.h  event.h      scheduler.cpp  task.cpp
event.cpp      prioQueue.h  scheduler.h    task.h
```

The `key*` files contain the driver for a debounced key input. The `output*` files contain the driver code to create the output. To create the cycles for the state machine, this behavior is placed in the files `statemachine*`.