Exercise E: Model-Based Software Development using EasyLab

Overview

In previous sessions, we learned some basics about programming a microcontroller by manually writing C code. You might already observe that it needs a lot of target-dependent knowledge to program a microcontroller and manual coding is very error-prone. There are ways to do such a jobs much easier, that is, to use modern model based development tools. The model based development paradigm puts emphasis on reducing the designer's burden and accelerating the design process by raising the abstraction level. Applications are usually composed by selecting components and specifying their connections via a graphical user interface (GUI). Formal model analysis or simulation can be supported to allow the designer to verify the design in early phases. The backend of the tools typically generates efficient code that implements the specified functionality. Deployment and debugging support may also be provided.

EasyLab

EasyLab[1] is a model-based development tool developed at the chair for robotics and embedded systems, targeting mechatronic applications.

Graphical Modeling Languages

EasyLab supports two graphical modeling languages:

a) *Structured flow chart*: The structured flow chart (SFC) language describes the states of a program and how state transitions are performed. States of SFC programs are references to sub-programs, which is in our case an SDF program (see below). Figure 1a shows some example SFC programs. The SFC program contains a unique *initial* state where the execution starts, represented using a double-lined frame (the swing-up state in the first example). The current state is left as soon as the *transition condition* that follows it is fulfilled. In the example, we leave the swing-up state and enter the balance state as soon as the angle α_i is less than a certain constant, meaning that the bar is almost erected.

b) *Synchronous data flow*: The synchronous data flow (SDF) language models a directed multigraph where each node is the instance of a certain actor type. An actor type is defined by a set of typed input and output connectors as well as internal state variables. Data processing and control tasks are usually described using SDF graphs.

Execution Modes

EasyLab supports two executions modes, namely simulation mode and code generation mode. In simulation mode, the actor instances of SDF programs are *virtually* executed on the host machine. It is mainly used to verify the functional correctness of the design. In code generation mode, C code is generated based on code templates that are associated with each actor type. The generated code can be directly compiled into an executable and uploaded to the target hardware. To provide the designer a more comfortably debugging environment, EasyLab supports debugging, in which the current values of each input/output port are forwarded from the target to the host and are displayed directly in the GUI. To enable debugging, you should check the *enable debug mode* check box in the code generation dialog. The debugging mode also allows you to modify the parameters of function blocks on-the-fly and see the changes in behavior immediately.

[1] See http://www.easy-kit.de/

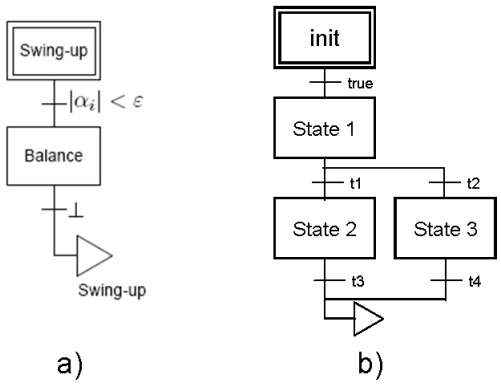

Table 1: SFC Programs

Easykit Starter Board and Match-X Blocks

Easykit Starter is a custom application board for EasyLab. It is designed based on a 32-bit ARM Cortex-M3 processor (STM32F103 from ST Microelectronics). On the board there are two sockets in which various so-called *Match-X Blocks* can be plugged to assemble hardware systems. In this session, we will use the following two types of blocks:

- `AIO1.1` enables the analog input 1 and analog output 1 on the respective stack.
- `DIO1.1` enables the digital input 1 and digital output 1 on the respective stack.

Multiple blocks of different types can be stacked upon each other.

Application Development using EasyLab

Exercise E.1: Simple Applications

a) Start EasyLab and connect the USB cable to Easykit Starter. The software should detect the presence of the EasyKit Starter board and show the available actors in the component library. Load the sample application `ButtonLED.easy` from `C:\Program Files\EasyLab\examples\EasyKit Starter`. Try to understand the simple SFC and SDF program.

b) Switch to code generation mode, enable debug mode and upload the sample application to the target device. After debugging is established, you should be able to visualize the current status of the buttons/LEDs in the GUI.

c) Familiarize yourself with the environment and develop a simple application, e.g., to toggle the LEDs.

Exercise E.2: Motor Speed Control

In this section, we will re-do the control theory exercise using EasyLab.

a) Develop the PI controller introduced in exercise 5.3 to regulate the speed of the motor.

- Do not forget to connect power and ground for the motor board.

- The analog output function block for Easykit Starter is actually a PWM generator. The input port is the duty cycle. The analog output ports on Easykit Starter board can be directly connected to the *Motor Analog/PWM* port to drive the motor.
- Digital input ports can be used for input capture. Use the *Digital Edge Detection* function block to implement the functionality. The output of this function block is the time in μs bwtween two successive edges.
- The PI controller component is already available in the generic library. The output of the controller can be directly applied to the duty cycle of the PWM generator to control the motor.
- You can adjust the coefficients c_p and c_i on-the-fly during debugging.

b) Optional: Try to model the PI controller by yourself (using primitive actor types like addition, summation, etc.), using the same parameter you found in the previous step.

- The stop watch function block provides you the time interval between two executions of the SDF program. It can then be used to compute the time for the PI controller. To avoid timer overflow, modify the output type of stop watch to `int32_t`.

Exercise E.3: Stairway Light Control

In this section we will develop an application similar to a stairway light controller. It works as follows: In a stairway, there are multiple buttons (e.g., one on each floor) to turn on/off the light. If the light is currently off, a button press will turn it on. If the light if already on, a button press will turn it off. For the sake of energy saving, the light should only be turned on if it is dark enough inside the building. In this section, LEDs are used to simulate the light in the stairway and the photo transistor on the motor board (labeled by T1) is used to measure the brightness. The photo transistor converts the brightness into an analog voltage value. The higher the brightness is, the higher the output voltage is.

a) Connect `Verstärker OUT` port on motor board to an analog input port on EasyKit Starter. Develop an application to turn an LED on when the transistor is kept away from a light source and turn it off again when the transistor is held towards a light source.

- The photo transistor is directional, i.e. it is more sensitive to light coming from the front direction.
- The `Verstaŕker OUT` port on the motor board can be either the light barrier output or the photo transistor output. You can select the signal from light barrier by putting the jumper `JP2` to 1-2 or select the signal from the photo transistor by putting the same jumper to 2-3.

b) Extend the previous exercise to perform the stairway light control. All the three buttons (`TA1`, `TA2`, `TA3`) should be able to control the light.

Exercise E.4: State Based Programming

The previous exercises have only a single *state*. However, in real world applications, a system usually incorporates multiple states. In this case, SFC is used to model the states and the *transition conditions* between them. In EasyLab, you can currently model sequential states, alternative branches and jumps to other states. The transition conditions can be Boolean constants (true/-false) or Boolean values resulting from algorithmic operations on *global variables*.

a) Develop a simple application that implements the SFC program shown in figure 1 b). Use the buttons as transition conditions so that you can control explicitly which state to enter. You might want to use some LEDs to illustrate the current state.

- Only global variables can be use to compute transition conditions. To create a global variable, right click the *Variables* device in the component library and choose *Add*. After entering the variable name, you should have the actors for reading/writing the global variable. In the SFC program, you should be able to see the list of global variables. Transition conditions can be written in C-like syntax using the name of global variables, for example:

$$\text{my\_var1} > \text{my\_var2}$$

b) Imagine we want an application to regulate a speed of the motor to a certain value (e.g. 3500 rounds per minute). When the difference between the current speed and the expected value is larger than a threshold (e.g. 500 rounds per minute), we consider a *fault* is happening. For safety reasons, we force the system to enter a *safe mode* (figure 2), in which the system will try to repair it self, i.e., to adjust the speed of the motor towards the expected value. On success, the system will leave the safe mode and enter normal mode again. Develop such an application in EasyLab.

- Use two buttons to increase/decrease the speed of the motor so that we can artificially cause a "fault".

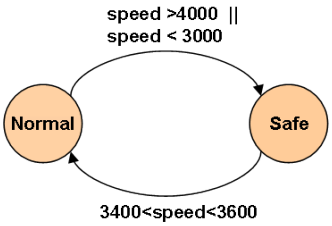

Table 2: State Machine of Motor Speed Regulator