Microcontroller Software Library for Process Control

JAN DOLINAY, PETR DOSTÁLEK, VLADIMÍR VAŠEK
Department of Automation and Control Engineering
Tomas Bata University in Zlin
nám. T. G. Masaryka 5555, 76001 Zlín
CZECH REPUBLIC
dolinay@fai.utb.cz

Abstract: - This paper deals with library of program modules for process control applications running on microcontrollers. The aim of the library is to make it easier to create control applications for microcontrollers. It should allow creating such applications by putting together existing program modules provided in the library with little new code required. The software is written in C language and works with Freescale HCS08 8-bit microcontrollers and the Kinetis series 32-bit ARM-based microcontrollers. It should also be easy to port the code to other platforms.

Key-Words: - discrete controller, hcs08, kinetis, microcontroller, program library, pwm

1 Introduction

Nowadays, microcontrollers (MCU) can be encountered in all areas of our life. MCU applications range from simple devices, such as toys, to complex embedded systems found in modern cars or aircrafts. Many of the MCU applications require implementing some control algorithm in the program. These applications could benefit from a library, which would contain such control algorithms, and possibly also some related useful code, in a form ready-to-use, without the need to write the algorithm from scratch and then debug it.

Generally speaking, one of biggest challenges in software development is the reuse of existing code. Probably everyone will agree that it can save considerable time and money, but in reality the situation is far from ideal and huge amount of code is rewritten over and over again. The higher-level programming languages used for PC programming encourage code reuse in some ways, but most of the MCU applications are written in C language and the code reuse is more in the hands of the programmer. Writing reusable code requires carefully designed and implemented code modules which are not focused just on fulfilling the task on given MCU, but also take into account the possible reuse on a different MCU. This is naturally harder than writing the code just for the application currently in focus and it probably explains why it seems that in MCU programming big part of the code is developed from scratch for every application. Certainly, there are some good reasons for this, such as that the hardware differs much across the MCU applications, but it may be in part also caused by the lack of effort to write portable code. This can hardly be surprising given the tight deadlines and pressure for high performance from the employers and the human nature of choosing the easier way to achieve the goal. For the programmer it is easier to write hardware-specific code for single MCU than write more generic code which is ready for future porting to another MCU.

Whatever the reasons are, the result is that the cost of embedded software is high and time-to-market is long. Or, in some cases, the quality of the software is poor.

One possible solution is in the usage of program libraries, which provide code usable in different applications. In the area of embedded programming, however, the hardware may be very diverse in different applications - which use different MCUs. Due to great variety in the hardware, it is in general not easy to create program libraries, which would be usable on several types of MCUs. Nevertheless, there are areas of problems which are virtually hardware-independent. For such areas it is possible to create libraries which will work on wide range of devices. One such area is system control. The control algorithm may be relatively complex, which makes it hard and time-consuming to implement and debug, but on the other hand, once the code is written and debugged in a portable programming language, such as C, it can be used on many devices without change.

In this paper we describe such program library for control applications. The main part of the library consists of discrete controllers, but it also proposes the framework for the whole control application

which allows separating the hardware-dependent code from the hardware-independent and thus makes the application easily portable to new MCUs. In the course of writing and testing the library also some supporting code (hardware drivers) was created, which is also included in the library.
The described software was developed for Freescale HCS08 8-bit microcontrollers [2] and for Freescale Kinetis family, which are 32-bit MCUs with ARM architecture [3], [4]. The idea of such a library originates from our previous work [7], [8], but the design and code is completely new.

2 Control library

When designing the library, the following requirements were defined:

- Provide discrete controllers usable in many common MCU applications
- The library should have easy-to-use programming interface
- It should be easily portable to new MCUs

From the logical point of view the library can be divided into four main parts:

- controller modules
- template code for user application
- software-PWM generator
- supporting code
- documentation and examples

The core of the library is represented by the controller modules. The word module is used here in the meaning common in C-language programming; i. e. the module is physically a pair of header (.h) and source (.c) file, which contains the definition and implementation of a single controller.
In the following subsections the main parts of the library are described in more details.

2.1 Controller modules

As already mentioned, the controller modules can be considered the core part of the library. From the logical point of view, the controller module is an “object” of one type of a controller, e.g. a discrete PID controller. The interface of the controller modules implemented so far in the library consists of two C functions:

- initialization function which initializes the internal data of the controller
- and “step” function which computes the controller output in each step of the control process (in each sampling period).

2.2 Template code for user applications

Besides the actual working code - the controller modules, the library also suggest preferred way of using these controllers by providing template files for user application. These template files contain skeleton code of a general control application. The user is advised to add these template files into his/her program and use it as a starting point for his/her application. The template files should guide the user in implementing the application-specific and hardware-specific code.
However, the user is not forced to use these application templates or any other particular style of programming. He/she is free to use any part of the library separately, for example, just the controller module(s), which are C functions and therefore can be called from any C program.

2.3 PWM generator

Another part of the library is a multi-channel generator of pulse-with-modulated signal. This signal can be used as a simple replacement for digital-to-analogue converter and therefore is utilized in many control applications for driving the actuators, e.g. for turning a heating element on and off in applications which control temperature.
In a typical microcontroller, there are several hardware timers capable of generating PWM signal, but if the application can use PWM signal with relatively long period (about 1 second or more), the software generator included in the library can be advantageous, because it is easier to use than the hardware PWM generator contained in the MCU and the code is hardware-independent.

2.4 Supporting code

The library also contains supporting code, which was created during the development of the library for testing its functions on real hardware. An example of such code is the driver for serial communication interface (UART).
This code can be directly used in applications targeting one of the MCUs supported by the library. For other MCUs it may at least provide starting point for the user’s own implementation of similar code, which will be likely needed when using the library on any MCU.

2.5 Documentation and examples

The library also contains documentation and example programs. The documentation consists of a user manual, which describes the library and its use and reference documentation created from the source code comments using the Doxygen tool.

There are also example programs which demonstrate the recommended use of the library in real application. For user convenience, the example programs are provided with complete projects for the CodeWarrior IDE.

3 Usage of the library

This section describes how the library is used in the client program. This description should also provide better insight into the overall design of the library.
The following picture shows the main components of the application which uses the library in the recommended way.

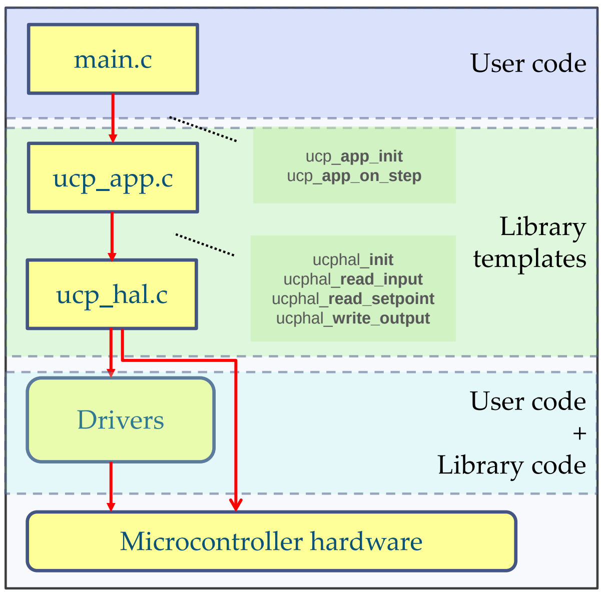


Fig. 1 Overall view of the application using the library

As can be seen in the figure 1, there are three main parts of the software of the application (on top of the MCU hardware): user code, library template files and device drivers.
Considering just the library code, there are two main parts:

- Application (ucp_app);
- Hardware abstraction layer (ucp_hal).

These main parts are located in two template files provided by the library: ucp_app.c and ucp_hal.c.
The interface between these files and the user application is defined by the library and consists of the following functions:
For the application module (ucp_app):

- ucp_app_init
- ucp_app_on_sample

For the HAL module (ucp_hal):

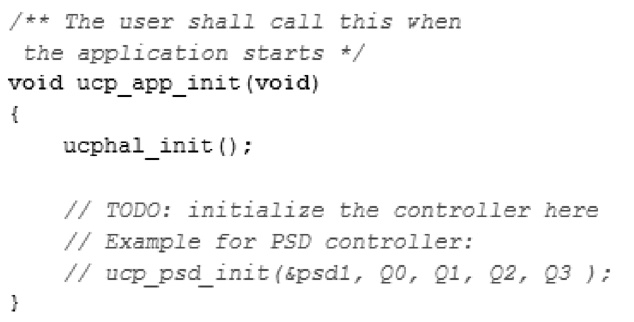
- ucphal_init
- ucphal_read_input
- ucphal_read_setpoint
- ucphal_write_output

These functions are described in detail in the following chapter. For explaining the concepts here, their names should be sufficiently self-explanatory.
As shown in the figure 1, the basic user code is contained within the main.c file. From this file the ucp_app interface functions are called. These ucp_app functions then contain most of the hardware-independent functionality of the application, such as initializing the controllers and calling the controller “on_step” function to calculate new value of the actuating signal in each sampling period.
Example of a simple main.c file can be seen in figure 2.

```
#include "ucp_app.h"

void main()
{
    ucp_app_init();

    for(;;) {
        ucp_app_on_sample();
        delay();
    }
}
```

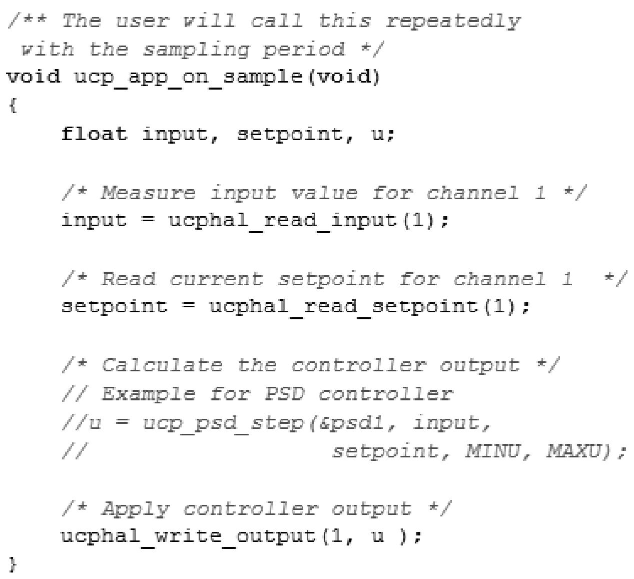
Fig. 2 The main function of a control application with the library

As can be seen in the code listing, all that happens in the main function is a call to the *ucp_app_init* function at the start of the program and then repeated calls to the *ucp_app_on_sample*. These calls should occur with the sampling period of the system. In real-world application it would probably use better timing tool than the busy-wait delay used in this simple example.
Both the *ucp_app_init* and *ucp_app_on_sample* functions are implemented in the template file ucp_app.c provided by the library. Obviously, the template cannot deliver the functionality for the user application. It is the user of the library who needs to write the program. However, the library tries to make this task easier by providing as much code as possible; for example, calling the HAL API where needed and providing skeleton of the application together with example code. The default contents of these two functions can be seen in figures 3 and 4.



```
/** The user shall call this when
 the application starts */
void ucp_app_init(void)
{
    ucphal_init();

    // TODO: initialize the controller here
    // Example for PSD controller:
    // ucp_psd_init(&psd1, Q0, Q1, Q2, Q3 );
}
```

Fig. 3 The ucp_app_init function in the template file



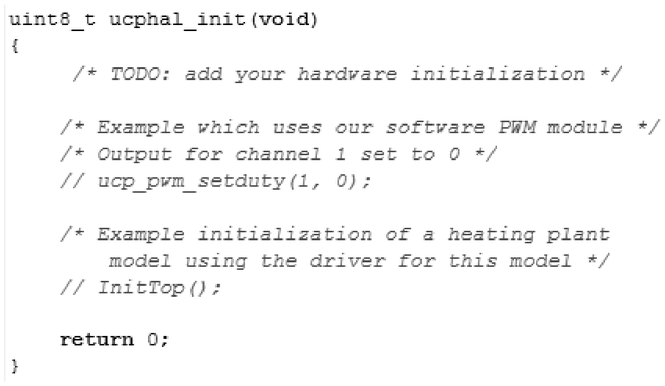
```
/** The user will call this repeatedly
 with the sampling period */
void ucp_app_on_sample(void)
{
    float input, setpoint, u;

    /* Measure input value for channel 1 */
    input = ucphal_read_input(1);

    /* Read current setpoint for channel 1  */
    setpoint = ucphal_read_setpoint(1);

    /* Calculate the controller output */
    // Example for PSD controller
    //u = ucp_psd_step(&psd1, input,
    //                 setpoint, MINU, MAXU);

    /* Apply controller output */
    ucphal_write_output(1, u );
}
```

Fig. 4 The ucp_app_on_sample function

For accessing the MCU hardware the library recommends using its hardware-abstraction layer (HAL) which is represented by the ucp_hal interface in the library. As can be seen in the listings in figures 3 and 4, the ucp_app functions do not access the hardware directly, but use the ucp_hal functions, for example, *ucphal_init* or *ucphal_read_input*. Besides the calls to the HAL, there is some example code in the comments which shows how to initialize the controller and how to obtain the new controller output in each sampling period.

The following two figures show the functions contained in the HAL template file. As in case of the ucp_app functions, also the HAL functions contain some example code in comments.

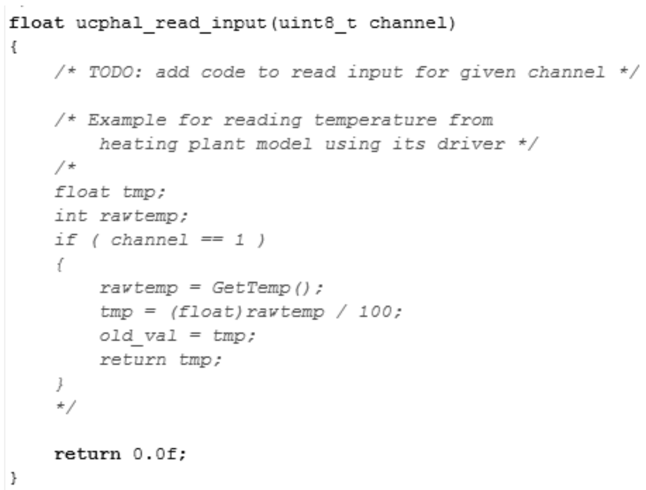
Figure 5 shows the *ucphal_init* function, which is called when the application starts and should perform all the initialization for the hardware used by the application. The example code provided in the template initializes the software PWM generator, which is part of the library. It also initializes the model of the heating plant, which was used for testing the library, as mentioned later in this article. The model is initialized by calling function *InitTop* from its software driver. Obviously, the user will most likely not need this particular code, but it illustrates how the various peripherals used by the application can be initialized using their own software drivers. Another option is to perform the initialization by directly accessing the MCU's peripheral registers from the *ucphal_init* function.



```
uint8_t ucphal_init(void)
{
     /* TODO: add your hardware initialization */

    /* Example which uses our software PWM module */
    /* Output for channel 1 set to 0 */
    // ucp_pwm_setduty(1, 0);

    /* Example initialization of a heating plant
        model using the driver for this model */
    // InitTop();

    return 0;
}
```

Fig. 5 The ucphall_init function in the HAL template

Figure 6 shows the *ucphal_read_input* function which is called in each step of the control process from the *ucp_app_on_sample* and should obtain the current value of the output of the controlled plant. In the template this is again illustrated by reading the temperature from the model of the heating plant using its driver function *GetTemp*. In general, the user will write here the code to access the peripheral used to measure the signal, for example, analog-to-digital converter.



```
float ucphal_read_input(uint8_t channel)
{
    /* TODO: add code to read input for given channel */

    /* Example for reading temperature from
        heating plant model using its driver */
    /*
    float tmp;
    int rawtemp;
    if ( channel == 1 )
    {
        rawtemp = GetTemp();
        tmp = (float)rawtemp / 100;
        old_val = tmp;
        return tmp;
    }
    */

    return 0.0f;
}
```

Fig. 6 The ucphall_read_input function in the HAL template

It should be noted, that the library supports multiple controlled variables and controllers in one application. In the code the term channel is used in the meaning of one pair of input and output signal or one controlled property (one instance of a

controller). As can be seen in the code listings, the functions such as *ucphal_read_input* receive the channel number as their input parameter.
The recommended scenario for using the library is as follows:

- User will create an instance of a controller by defining one variable (a C structure which holds the private data of this instance of the controller).
- After creating the variable for controller data, user passes this variable into the controller initialization function.
- Then he/she ensures that controller "step" function is called regularly with the period equal to sampling period of the system. This can be done using simple busy loop in the main function, or (preferably) by using hardware timer.

The user is assumed to add the template files ucp_app.h and .c and ucp_hal.h and .c to his/her project and implement the application and hardware specific code in these files. The application logic including the controller variable(s) is contained in ucp_app.c file.
Note that the fact that the user provides the memory for storing the controller data is advantageous for embedded systems with limited amount of RAM, because only as much memory is occupied as is needed. Other approaches which provide the memory internally and automatically in the library may be somewhat easier to use, but they require more resources. Either the memory needs to be statically allocated inside the library, which limits the number of available controllers and wastes memory of the unused controllers, or there needs to be dynamic memory management used, which brings large overhead to the code.

4 API of the library

This section describes the application programming interface of the library's Application (ucp_app) and HAL (ucphal) modules.

4.1 Application API

Currently, the ucp_app API consists of just two functions:

- ucp_app_init
- ucp_app_on_sample

As already mentioned, these functions are implemented in the ucp_app.c file.
The *ucp_app_init* function initializes the application including the hardware. Its main task is to call the HAL initialization function, which initializes the MCU peripherals used by the application. This code needs to be written by the user of the library. It cannot be provided in the template files as the actual peripherals (e.g. I/O pins) are strictly application-specific. However, the user can take advantage of the library drivers for some peripherals or write custom code for the required hardware.
The *ucp_app_on_sample* function is called by the main program in every step of the control process. In general, it performs these steps:

- Read the input(s)
- Read the setpoint(s)
- Compute the actuator signal(s)
- Output the actuator signal(s)

The HAL API provides standardized functions for these tasks. However, the internals of these functions need to be written by the user. For example, the reading of the input signal can be performed using the Analog-to-Digital Converter (ADC) in the MCU, by processing a signal from a sensor, such as value encoded by PWM, or even through some communication interface such as SPI or UART. This depends on the application and cannot be handled in the library templates.
Reading the setpoint is similar, although the setpoint is typically entered by the user. There are many ways how entering the desired output of the controlled process can be handled in the application. Therefore the UCP library uses the same ways of abstracting access to this property as with input and output signals. As a result, it does not matter for the application whether the setpoint is obtained by reading an analog value set by a potentiometer, read from memory where the value is stored and updated when user presses some buttons, etc. This hardware-specific code is isolated in the *ucphal_read_setpoint* function from the rest of the application.
The computation of the actuator signal is one part of the code which is handled from the biggest part by the library itself. This is done by calling the controller "on step" function, for example *ucp_psd_on_step*.
The realization of the output signal also depends on the application greatly. One often used type of output is a PWM signal. As already mentioned, the library provides built-in support for this kind of output. For other types of output signal, for example, digital to analog converter or simple discrete output, the user must provide his/her own code. In any case this hardware-specific code should be hidden from the rest of the application in the *ucphal_write_output* function.

4.2 HAL API

The ucphal API consists of four functions:

- ucphal_init
- ucphal_read_input
- ucphal_write_output
- ucphal_read_setpoint

These functions are implemented in the ucp_hal.c file.

The *ucphal_init* function should perform all the initialization of the hardware. As with all functions in the HAL module, this needs to be written by the user of the library according to the needs specific for given application. The default content of this function in the template file provided by the library can be seen in figure 5 above.

The *ucphal_read_input* function should return the current value of the controlled-process output for given channel. The requested channel number is specified as input parameter to this function. As already discussed, this function provides uniform interface to the control application for obtaining the input signal without regard to the actual hardware principle for obtaining this input. The default implementation from the template file can be seen in figure 6 above.

The *ucphal_write_output* function can be described as opposite to the previous function. It takes care of applying the output calculated by the controller to the actuator for given channel. Again, this function represents uniform interface for the control application for performing this task, hiding the actual principle used for controlling the actuator from the application code.

The *ucphal_read_setpoint* function is responsible for obtaining the current value of the setpoint for given channel. As already discussed, the means of obtaining the setpoint value can vary and the purpose of this function is to allow the control application to use one standard way of obtaining the setpoint value without dealing with these differences.

5 Library controller modules

Currently, two control algorithms are implemented in the library, i.e. two types of controllers are available: discrete PID controller (PSD controller) and On-Off controller with hysteresis.

5.1 Discrete PID controller (PSD controller)

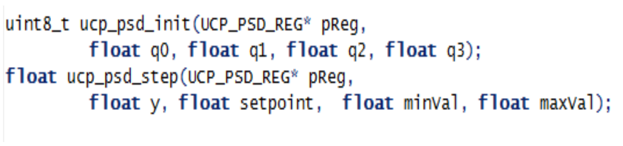
This module implements the incremental version of the discrete PID algorithm. The recursive equation used to compute the control signal in each sampling period is:

$$u(k) = u(k-1) + q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) \quad (1)$$

Where *k* denotes the step number, for example, e(k) is the control error in current step and e(k-1) is the error in previous step.

The coefficients of the controller q0, q1 and q2 can be obtained by methods for controller tuning, such as [1].

From the programmer's perspective, the interface of this module is represented by data structure *UCP_PSD_REG* and functions *ucp_psd_init* and *ucp_psd_step*. Signatures of these functions can be seen in figure 7:



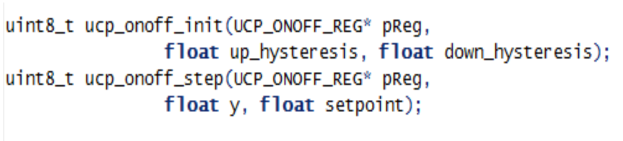
```
uint8_t ucp_psd_init(UCP_PSD_REG* pReg,
        float q0, float q1, float q2, float q3);
float ucp_psd_step(UCP_PSD_REG* pReg,
        float y, float setpoint,  float minVal, float maxVal);
```

Fig. 7 Interface of the discrete PID controller

The "init" function is called by user program once at the beginning to initialize the controller. The "step" function should be called in every sampling period to obtain new value of the control signal. Both these functions receive pointer to the data structure of the controller as their first parameter. Parameter *y* is the current output of the controlled plant; *setpoint* is the desired value of the controlled signal; the *minval* and *maxval* are the boundary values for the control signal (e.g. 0 and 100 percent for PWM signal). These values need to be provided as the input parameters of the controller function, because the old values of u(k) stored inside the controller must correspond to control signal really applied to the controlled process.

5.2 On-off controller

This module implements simple On-Off controller with optional hysteresis. The data structure for this controller is called *UCP_ONOFF_REG*. As in the case of the discrete PID controller, there are two functions. The signatures are shown in the figure 8.



```
uint8_t ucp_onoff_init(UCP_ONOFF_REG* pReg,
            float up_hysteresis, float down_hysteresis);
uint8_t ucp_onoff_step(UCP_ONOFF_REG* pReg,
            float y, float setpoint);
```

Fig. 8 Interface of the on off controller

As can be seen in the figure, the user can specify two values of the hysteresis; one for the "up" direction of the controlled signal and one for the "down" direction. The other parameters have the same meaning as explained in the discrete PID controller section.

6 Target hardware

The library aims to be as much hardware-independent as possible. The hardware-specific code which is necessary should be easily portable to other MCUs. However, it was necessary to implement and test the library on some specific hardware. We selected Freescale's 8-bit HCS08 MCUs and 32-bit ARM-based Kinetis series MCUs as two representatives of relatively different MCU platforms. Important role in the selection played the availability of the device in a development kit. For these reasons we used the development kit from the MCU-programming lessons at our faculty, which contains HCS08 GB60 MCU for the 8-bit microcontroller implementation. For the 32-bit version a new, very affordable development kit FRDM-KL25Z with the Kinetis MCU was used [4].

6.1 HCS08 8-bit microcontrollers

As mentioned above, for testing the library on 8-bit MCUs we used development kit M68EVB908GB60. This kit is no longer manufactured, but still used in our lessons. The GB60 derivative used in this kit is still available and besides, it should require little effort to use the code for the GB family on another (more modern) member of the wide HCS08 product line, e.g. the QG or SH families.

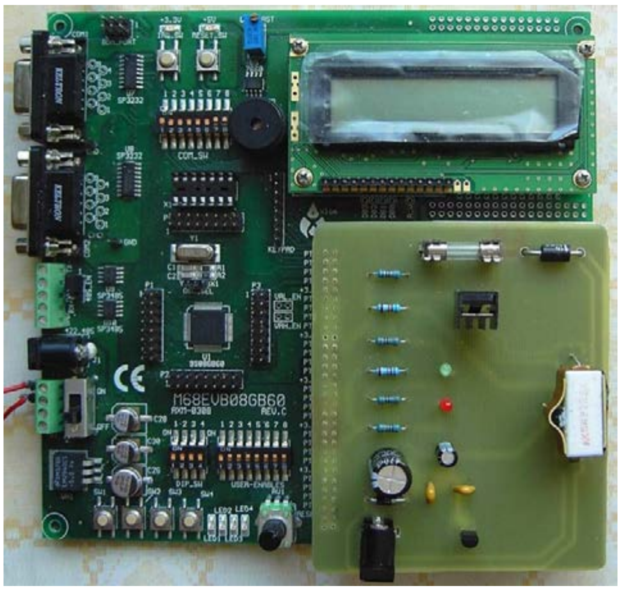


Fig. 9 HCS08 development kit with heating-plant model attached

In figure 9 the evaluation kit can be seen. There can also be seen the model of a heating plant [5] attached to this kit, which was used for experimental verification of the library. This model contains a resistor which can be heated using digital output of the MCU. There is also temperature sensor attached to this resistor whose output is attached to the MCU.

6.2 Kinetis 32-bit microcontrollers

The second hardware platform for which the library was implemented is Freescale Kinetis series of 32-bit microcontrollers. Compared to the HCS08 core, the 32-bit MCUs offer higher computing power and more memory, which makes them suitable even for more demanding control applications.

There is a line of low cost evaluation boards available for the Kinetis microcontrollers called Freedom platform [3]. In our case FRDM-KL25Z board was used. This board contains KL25Z128VLK4 microcontroller with 128 kB of Flash and 16 kB of RAM memory together with programming and debugging interface (openSDA). The layout of the board is compatible with the layout of popular Arduino platform [9] which makes it possible to connect expansion boards (so called shields) for Arduino to this board.

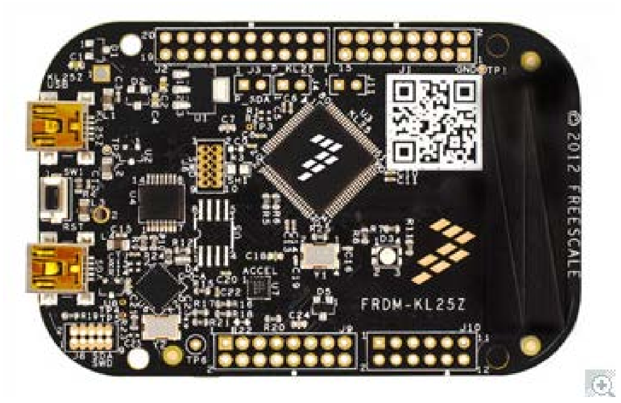


Fig. 10 FRDM-KL25Z board used for tests [4]

7. Experimental verification

To verify the functionality of the library several control applications were created. The controlled system for this verification was represented by a model of heat plant, which we also use in programming lessons. This model was described in detail in [5]. The model represents a 2nd order system with transfer function approximately:

$$G(s) = \frac{0.8}{(86.5s+1)(18.2s+1)} \qquad (2)$$

This model was attached to the evaluation board with HCS08 microcontroller by the means of its connector – the same way it is used in our lessons. In case of the Kinetis board, which does not have compatible connector, the model was connected using wires.

Figure 11 shows the result of control process with the discrete PID controller module. It is very simple control process, but it demonstrates the correct function of the program library. The parameters of the controller were designed using method [1]. The control signal is generated using the software PWM module which is part of the library. In the figure, the set point and output of the plant are depicted in degrees Celsius; the control signal is shown in per-cents. The set point value was 50 °C.

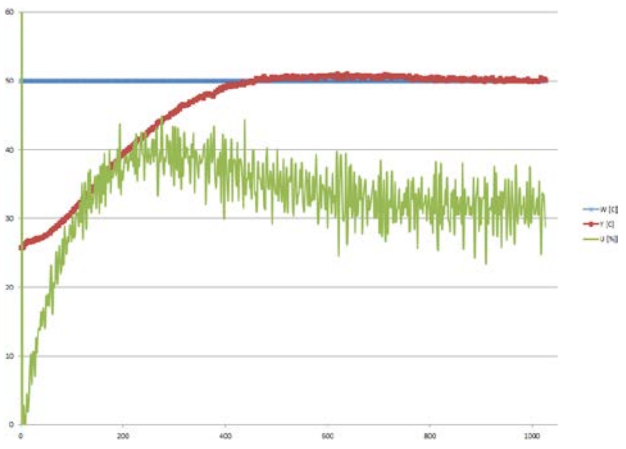


Fig. 11 Experimental verification of the PSD controller

8 Conclusion

This article described program library we created for control applications. The library makes it easier and quicker to create microcontroller applications for control systems. This aim is achieved by set of hardware-independent modules, which are ready-to-use and also by providing framework for writing the control application. This framework includes also hardware-dependent code and the interface between this code and the rest of the application.

The core of the library is formed by controller modules which can be directly deployed in user application. There are also template files which provide the skeleton of the whole control application with strict separation of hardware-specific and generic code. Besides those two components, the library contains also supporting code such as device drivers and software PWM generator. Documentation and example programs are provided as well.

The library is written in C language and currently implemented and tested on two types of microcontrollers: Freescale's 8-bit HCS08 and 32-bit Kinetis (ARM) series. However, porting it to other microcontrollers should be relatively easy and straightforward.

This work was supported by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089)

References:

[1] A. Víteček, M. Vítečková, Inverse Dynamics Method Tuning and Basic Quality Indices, *9th International Scientific Conference CO-MAT-TECH 2001*, Slovenská technická univerzita, 2001, pp. 412-417.

[2] Freescale Semiconductor, *CPU08 Central Processor Unit Reference Manual, rev.4*, Available: http://www.freescale.com.

[3] Freescale Semiconductor, *Freescale Freedom Development Platform*. Available: http://www.freescale.com/webapp/sps/site/overview.jsp?code=FREDEVPLA.

[4] Freescale Semiconductor, *FRDM-KL25Z: Freescale Freedom Development Platform for Kinetis KL14/15/24/25 MCUs*, Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL25Z.

[5] J. Dolinay, P. Dostálek, V. Vašek, Educational models for lessons of microcontroller programming, *in Proc. 11th International Research/Expert conference TMT 2007*, Hammamet 2007, pp. 1447-1450.

[6] P. Dostálek, V. Vašek, J. Dolinay, Design and implementation of portable data acquisition unit in process control and supervision applications, *in Proc. 13th WSEAS International Conference on CIRCUITS*, Rhodes 2009, pp. 799-808.

[7] J. Dolinay, V. Vašek, P. Dostálek, Implementation and Application of a Simple Real-time OS for 8-bit Microcontrollers, *in Proc. 10th WSEAS International Conference on ELECTRONICS, HARDWARE, WIRELESS and OPTICAL COMMUNICATIONS (EHAC '11)*, Cambridge 2011, pp. 023-026.

[8] J. Dolinay, P. Dostálek, V. Vašek, P. Vrba, Platform for teaching embedded programming, *International journal of mathematical models and methods in applied sciences*, vol. 5, no. 6, pp. 1110-1117, 2011.

[9] Arduino, *Open-source electronics prototyping platform*, Available: arduino.cc.