# Utilization of Simple Real-time Operating system on 8-bit microcontroller

DOLINAY J., VAŠEK V., DOSTÁLEK P.

***Abstract*—**The paper deals with small real-time operating system developed at our institute together with an application which uses this system and thus provides verification of this system in a portable data acquisition unit. The system, named RTMON is used mainly as a teaching aid for lessons of microcontroller programming. It makes it possible for students to write applications in C language with several concurrently running processes in a simple way. However, it can be used also in practical applications, which is demonstrated by the data acquisition unit described in this paper. The system is implemented for 8-bit microcontrollers with the HC(S)08 core made by Freescale and for Atmel AVR Mega8 microcontrollers.

***Keywords*—** microcontroller, HC08, real-time, operating system, data acquisition.

## I. Introduction

REAL-time operating system (RTOS) can help to solve the common problems related to programming microcontroller applications, such as need for executing multiple tasks concurrently, quick response to high priority events, managing hardware resources of the MCU, etc. In our lessons we include also RTOS based programming.

On 16-bit or 32-bit MCUs, RTOS are used often; on smaller 8-bit systems it is not so common because these systems have limited memory and CPU power and it is more efficient to write the required program without RTOS. However, if the RTOS is small enough to fit into such MCU, it can bring the same advantages as on bigger MCUs. At our department in lessons of Microcontroller programming we use 8-bit MCU from the HCS08 family made by Freescale. As we wanted to include a RTOS programming techniques into our lessons we needed a RTOS capable of running on this MCU. Such system would also be useful for our other projects, where we use Freescale HCS08 MCUs. There are quite many real-time operating systems available, but most of them are focused on bigger, 16 and 32-bit MCUs. There are some systems which support also small, 8-bit MCUs, for example, FreeRTOS [2] which is distributed under GPL license and currently officially ported to 23 architectures. Another example is MicroC/OS-II [1], [3], which is also free for educational, non-commercial use. It is suitable for use in safety critical embedded systems such as aviation or medical systems and is ported to many of architectures including Freescale HC08 and Atmel AVR. Certain disadvantage of using such system is that it is often quite complex due to the wide options it offers; typical RTOS for 32 bit MCU will contain drivers for USB, Ethernet etc. Despite the fact, that the systems are configurable to work in simple applications, the user can still have too many things to define and study. Moreover, we already had a real-time operating system developed at our institute for PC systems and ported also for Motorola (Freescale) HC11, and the interface of this system is known to the students. So, even if it would be possible to choose from existing systems, we decided to implement a light-weight version of our RTMON system for use on HCS08 microcontroller. Once the system was up and running for the HCS08 derivative used in lessons (GB60) it became useful to port it to other derivatives also. As a result, RTMON currently supports not only several members of the HCS08 MCU family but also Atmel AVR ATmega8. Adding new derivative is quite simple, so the list of supported derivatives will possibly grow in the future. Besides using the system in lessons, RTMON was also successfully used in design of portable data acquisition unit – DAQ, which allows simple and cheap interface between personal computer and a technological process. In the following text we describe the properties and usage of the RTMON operating system and also the DAQ device which uses this system.

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the Research Plan No. MSM 7088352102 and by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.

## II. RTMON operating system

RTMON is multitasking, pre-emptive operating system which is greatly simplified for easy use by the students. It is written in C language except for a small, platform-specific part written in assembler. The system supports execution of two different types of processes (tasks): normal processes which execute only once (such processes typically contain infinite loop) and periodical processes which are started automatically by the OS at certain period. These periodical processes are useful for many applications, typically in discrete controllers which need to periodically sample the input signal and update the outputs.

In user's application RTMON is utilized as a precompiled library accompanied by a header file. This simplifies the organization of the project and the build process. User enables RTMON usage in his program by including the header file (rtmon.h) in his source and adding the library to his project. Currently the library and sample projects are available

two development tools: Freescale CodeWarrior for Microcontrollers and Atmel AVR Studio with WinAVR suite.

If needed, user can also rebuild the RTMON library. Typically this is useful to change the configuration, such as for example, maximum number of tasks, length of the OS time period (tick), etc. There is documentation which describes the procedure and also projects for the two supported IDEs, which can be simply opened and built.

The effort to make both the implementation of the system itself and its usage as simple as possible brings several restrictions for the system features. First, the RAM memory for processes and their stacks is statically allocated according to the maximal number of processes defined in configuration file. For the user program, it is not possible to use this memory even if there are fewer processes defined. In case more RAM is needed for the user program, the maximum number of tasks and/or stack-pool size can be changed in configuration file and the RTMON library must be rebuild.

Priority of each task must be unique, so that in each moment one task (the one with highest priority) can be selected and executed on the CPU. Processes can be created on the fly, but it is not possible to free and reuse memory of a process. No more than the maximal number of processes can be created, even if some processes were previously deleted.

However, these restrictions do not present any problem for most applications and allow for small kernel code size and ease of use.

### *A. Kernel implementation*

There are only two objects which RTMON contains: a process and a queue. The queues are buffers for transferring data between processes. Several queues can be created, each containing a "message" (data buffer) of certain size. The size can be specified when creating the queue and is limited by the total size of RAM reserved for all buffers of all the queues (queue pool size). Processes can read and write data to the queue and wait for the queue to become empty or to become full. This allows using a queue also for process synchronization.

### *B. System internals*

The OS uses timer interrupt which occurs at certain period (e.g. 10 ms) to periodically execute the scheduler, which decides which process will run in next time-slice. The timer interrupt routine is implemented in assembler for HC08 MCUs and in C for AVR MCUs. It first stores CPU registers onto the stack and then calls RTMON kernel, which is a C function. The kernel then finds the process with highest priority which is in ready-to-run state and switches the context, so that the code of this process is executed after return from the interrupt service routine. If no process is ready to run, then a special dummy process is executed. This dummy process is contained within RTMON code and does nothing.

Task descriptor in RTMON is a C-language structure (IDPROC) which occupies 18 bytes of memory (given that char is 8-bit and int is 16-bit). The size of RAM required, for example, for 10 user-defined processes is then 12 x 18 = 216 bytes - there are two extra structures reserved for the init and dummy processes. The memory consumption may be reduced if we limit some of the values (e.g. stack size and time intervals) to 8 bits. This is enabled by RTMON_SMALL directive and it reduces the size of RAM required for one process to 14 bytes.

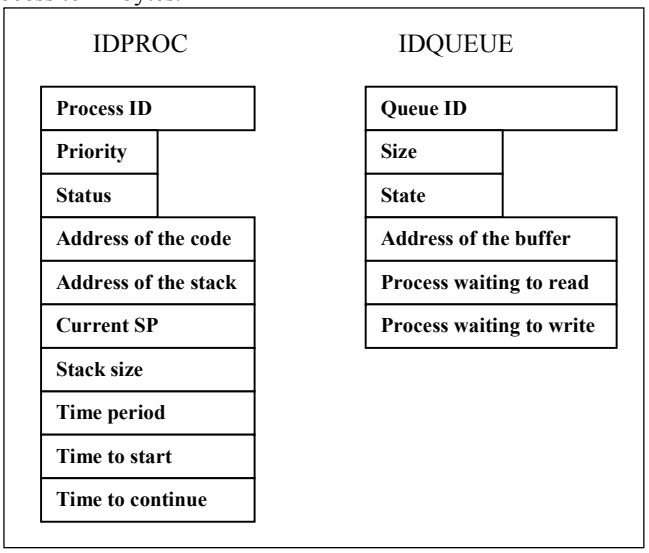


Fig. 1 RTMON data structures for a process and a queue

There is an array of these structures with the number of items defined by RTMON_MAXPROCESSES constant in RTMON configuration file.

The structure for a queue (IDQUEUE) requires 10 bytes of RAM and similarly as for processes, RTMON allocates array of IDQUEUE structures with the number of items defined by RTMON_MAXQUEUES constant.

### *C. System services*

The OS provides set of services to user applications to manipulate processes and queues. Each service corresponds to a function in the RTMON library which user program can call. There are services for processes which allow to:

- Create a process
- Start a process
- Stop a process
- Delay (sleep) a process
- Continue (wake up) a process
- Abort (delete) a process

For the queues there are the following services:

- Create a queue (specify size)
- Write to a queue with or without waiting
- Read from a queue with or without waiting

The services will now be described in more details.

*char rtm_init(IDPROC** init_id);*

This function will initialize rtmon data structures and create

2 built-in processes: init and dummy. The dummy is the idle process which runs when no other process is ready to run. It has priority 255, which is the lowest. The init process in rtmon is a special process with highest priority (0) and it is the process which controls creation and end of all other processes. By calling rtm_init the user program obtains ID of this init process and the C function which calls rtm_init (typically it is the main() function) becomes the body of the init process.

So from the call to rtm_init on from the rtmon's point of view the code of function main is now the code of the init process and it is executed until it delays itself by call to rtm_delay_p. Note that the init process will create other processes and then start them, but no other process will actually execute until init process puts itself into sleep because the init process has the highest priority.

*char rtm_end(IDPROC* init_id);*

It stops the init and dummy processes and stops the periodical timer interrupt which provides ticks for the OS. This service can be called at the end of user program (end of main function). However, since typical embedded application runs infinitely and never ends, it is often not needed to call this service at all.

*char rtm_create_p(const char* pname, unsigned char prio, void(*pfunc)( ), int stack_size, IDPROC** proc_id );*

This service creates a new process in rtmon and returns the caller (via the proc_id parameter) the identificator of this process which is then used for all further operations with this process.

Pname – this is descriptive name of the process. It is currently not supported by rtmon to save memory, so it is ignored and need not be set.

Prio – the priority of the process. This is integer between 1 and 254, the values 0 and 255 are reserved for rtmon. The higher is the number the lower is the priority.

Pfunc – pointer to the function which represents the process. This is the code of the process.

Stack_size – size of the stack for this process. Each process has its own stack which must be big enough to hold all cpu registers (saved on interrupt), all local variables (defined within the function which represents the process) and all return addresses for function calls made from the process including system calls. Rtmon keeps an array in RAM (stack pool) from which it allocates stacks as specified in call to rtm_create_p. The maximal size of stack of all processes cannot exceed the size of this stack pool. The pool size is defined in header file and can be changed but the rtmon library must be then rebuilt.

Size of stack depends on architecture and also on the number of variables and function calls in the process, typical minimum value is about 32 bytes for HCS08 and 48 for AVR, if the process uses virtually no stack. Recommended minimum is 64 bytes.

*char rtm_start_p(IDPROC* proc_id, int time_to_start, int time_period);*

Starts a process.

proc_id - ID of the process to start. It is obtained by previous call to rtm_create_p.

Time_to_start allows specifying delay for the start so that the process is not startet immediately but only after specified number of ticks. Typically this argument has value 0 which means the process is started immediately.

Time_period – this argument allows specifying the period with which the process is automatically restarted. If it is 0 the process is started only once (typical for processes which then run in infinite loop). If it is nonzero then is the period in ticks with which the process is started repeatedly. It is important to ensure that the process ends in shorter time than is the period of start in this case!

*char rtm_delay_p(IDPROC* proc_id, int time_to_delay);*

Allows pausing execution of given process for specified period of time (given in ticks). Value of 0 means infinite delay, which puts the process into sleep until it is waken up by other process.

*char rtm_continue_p(IDPROC * proc_id);*

This service continues execution of a process previously delayed by rtm_delay_p call.

*char rtm_ch_period_p(IDPROC * proc_id, int time_period);*

This service changes the period of a process. The argument time_period specifies the new period of start of the process. Typical use of this function is to cancel the periodical start of a process by calling this function with time_period equal to 0. This is necessary when periodical process is to be stopped because without changing the period to 0 (canceling periodical start) the process would automatically start in the next period.

*char rtm_stop_p(IDPROC * proc_id );*

Stop execution of a process. Most common use is to call this function at the end of the process in periodical processes. Such process must end by this call to return control to rtmon. Another use is to stop all processes when rtmon is ended if such situation is needed.

*char rtm_abort_p(IDPROC * proc_id );*

Remove process from the list of existing processes in rtmon. This function has currently no use because rtmon does not allow reclaiming memory occupied by a process when the process is aborted, so even if process is aborted, it does not allow creating a new process instead of it, if the maximum number of processes has been reached.

The queues are optional component which can be excluded from the system is not needed to save memory. In such case the services are not available.

There are three basic operations supported on a queue:

create, write and read. Deleting a queue is not supported.

*char rtm_create_q(const char* pname, char l_msg, char n_buff, IDQUEUE** pid_queue);*

pname is the name of the queue, it is not used similarly as in process name in rtm_create_p.

l_msg is the size of the buffer (message) which this queue can hold.

n_buff – the number of messages (buffers) this queue can hold. This argument must be 1, this version of RTMON does not support more than one message per queue.

Pid_queue - receives the id of the newly created queue. User program then uses this ID for all further operations with the queue.

For write and reading messages to queues there are two variants of functions – one with suffix _w which stands for "wait". Such function waits for the queue to become full/empty if it is not in the proper state. These wait functions can be used to synchronize process execution.

Queue read and write functions have the following prototypes:

*char rtm_write_q(IDQUEUE* pid_queue, void* pdata);*
*char rtm_write_q_w(IDQUEUE* pid_queue, void* pdata);*
*char rtm_read_q(IDQUEUE* pid_queue, void* pdata);*
*char rtm_read_q_w(IDQUEUE* pid_queue, void* pdata);*

pid_queue is the ID of the queues obtained by previous call to rtm_create_q and

pdata is pointer to memory buffer which receives or contains the data to be read from/written to the queue. Note that rtmon reads/writes the number of bytes defined by queue size into this buffer and the user must ensure it has the proper size (is is pointer to proper data type).

For example, when defining queue for passing integers with size sizeof(int) then the queue is read by command:

rtm_read_q(q, &data); where data is defined as: int data;

### D. Using the system in user application

To create an application which takes advantage of RTMON the user needs to perform just several simple steps:

1) Define variables for process identificators, e.g.: IDPROC* init, *p1;
2) Initialize RTMON (typically in the main function): rtm_init(&init);
3) Create user processes:

rtm_create_p("proc1", 10, proc1, 64, &p1);

This call creates process with priority 10 and stack size of 64 bytes. The body of the process is in function proc1 which should have the following prototype: void proc1(void). The variable p1 receives the ID of the newly created process and is used in all further calls to RTMON services to manipulate this process.

4) Start one or more processes:

rtm_start_p(p1,0,5);

This call starts process p1. The number 0 means that the process is started immediately (with delay of 0 ticks) and the number 5 means the process is started with period 5 ticks (it will be automatically started by RTMON each 5 ticks).

5) Delay the init function (the main process):

rtm_delay_p(init,0);

By this call the init process (main function) puts itself into infinite sleep and thus allows other processes to run. At this line the execution of main stops and it moves to the process with highest priority.

Code of each user process is contained in a C function. Example of a simple process could be:

```
void proc1(void)
{
  rtm_stop_p(p1);
}
```

This process does nothing, it just calls rtm_stop_p(p1) informing the system that it stopped execution.

## III. Data Acquisition Device with RTMON

As already mentioned, RTMON is used in our lessons of microcontroller programming. But besides this usage the operation of the system was also verified by using it in one of our devices – a multi-channel portable data acquisition device DAQ. This device was developed in our department mainly for controlling and monitoring of educational laboratory models. It offers cheap alternative to professional I/O cards and modules when a technological process needs to be controlled or monitored from a computer [8].

### A. Hardware of the DAQ

Hardware design of the DAQ device offers 16 analog inputs with 12-bit resolution, 8 digital inputs and outputs and one analog output with 12-bit resolution. The design focuses on low power consumption which allows long operation when battery supply is used. The block diagram of the DAQ device electronic circuits is depicted in the Fig. 1.

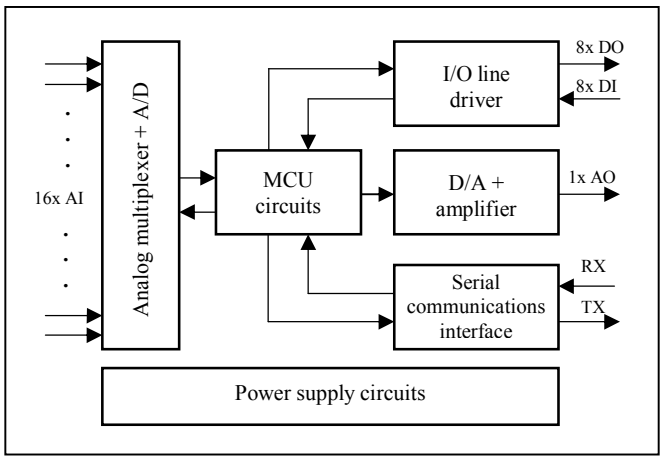


Fig. 1 Electronic circuits block diagram

The core of the data acquisition device is 8-bit general purpose Motorola microcontroller 68HC908GP32 with Von-Neumann architecture which is fully up-ward compatible with the 68HC05 family. On the chip are integrated many useful peripherals including: timer interface with input capture and output compare functions, 8-channel analog-to-digital converter with 8-bit resolution, up to 33 general-purpose I/O pins, clock generator module with PLL, serial communication interface and serial peripheral interface. The MCU has implemented several protective and security functions such as low-voltage inhibit which monitors power supply voltage, computer operates properly (COP) counter and FLASH memory protection mechanism preventing unauthorized reading of the user's program. Internal RAM memory has capacity of 512B and FLASH memory 32 KB. Internal clock frequency can be 8 MHz at 5 V operating voltage or 4 MHz at 3 V operating voltage. The MCU also supports wait and stop low-power modes [4], [5]. It is the part of the MCU circuits block incorporating all necessary electronic circuits for its operation (Pierce crystal oscillator, PLL filter circuits).

Analog-to-digital conversion is performed by Linear Technology A/D converter LTC1298. It is micro power, 2-channel, 12-bit switched-capacitor successive approximation sampling A/D converter which can operate on 5 V to 9 V power supplies. Communication with microcontrollers is handled by 3-wire synchronous serial interface. It typically draws only 250 μA of supply current during conversion and only 1 nA in power down mode in which enters after each conversion [6]. It is supplied from high-precision 5 V voltage reference LM336-Z5.0 which is powered from adjustable current source LM334 to achieve very stable voltage over the specified input voltage range. Analog inputs are multiplexed by two 8-channel high-speed CMOS analog multiplexers 74HC4051 with turn-on and turn-off delay of 20 ns.

Digital-to-analog circuit uses 12-bit D/A converter Burr-Brown DAC7611 with internal reference and high speed rail-to-rail amplifier. It requires a single 5 V supply. Power consumption is only 2.5 mW at 5 V. Build-in synchronous serial interface is compatible with variety of digital signal processors and microcontrollers [7]. Its output is amplified by general purpose MC1458 operational amplifier to standard voltage range of 0 to 10 V.

Technical parameters of the data acquisition device are summarized in the Table I.

Table I Basic parameters of the DAQ device

| | |
|---|---|
| Digital inputs | 8 channels, TTL compatible |
| Digital outputs | 8 channels, TTL compatible |
| Analog inputs | 16 channels, 12 bits resolution, 0–10 V |
| Analog outputs | 1 channel, 12 bits resolution, 0–10 V |
| Supply voltage | 6.5 to 9V DC |
| Communication | RS232 interface, 57600 Bd |

*B. Software of the DAQ*

The software in the DAQ device is based on the RTMON operating system described above. The software is formed of RTMON core and individual processes which perform all necessary tasks. Each process activity is controlled by operating system core on the basis of process priority and other information stored in the task descriptor. Structure of the DAQ device firmware is depicted in the Fig. 2. As can be seen in the figure, there are 4 main processes and 1 interrupt handling routine.

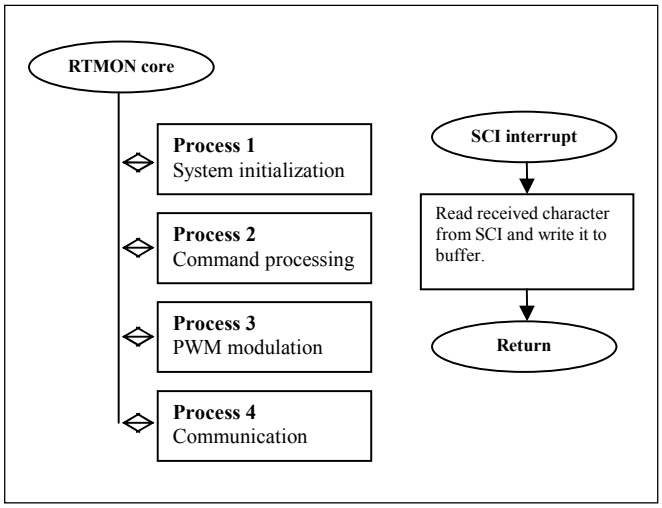


Fig. 2 Internal software structure of the DAQ

Process 1 is highest priority process which performs DAQ device initialization after power up or reset. It sets all digital outputs to low state (logic 0), setups serial communications interface to communication speed of 57600 Bd, 8-bit data frame, 1 start bit and 1 stop bit, sets analog output to 0 V and finally initializes all necessary data structures. Because of its highest priority no other processes can be switched by OS core into the "run" state. After all initializations are done process suspends itself.

Process 2 performs all tasks related to command interpretation and execution. It waits for complete command string in the receiver buffer which is handled by serial communication interface (SCI) interrupt routine. This interrupt routine is automatically called when SCI receive one character from the higher-level control system. When command is completely received in the buffer, process will decode it and executes required action.

Process 3 is periodically activated process performing pulse-width modulation (PWM) on all 8 digital output channels when it is demanded. Its priority is set to higher level than process 2 and process 4 because the PWM is time critical function sensitive to accurate timing. Its 8-bit resolution allows setting of 256 different duty cycles at output. Period of the PWM signal is after initialization automatically set to 1000 ms which is optimal value for many controlled systems with higher time constants. This value can be changed during device operation by user command.

Process 4 provides communication via RS232 serial interface with supervisory system. It generates responses to all commands regarding to defined communication protocol include error processing. It has the lowest priority from all the processes.

### *C. Communication protocol*

Data acquisition device communicates with supervision system using standard serial interface RS232 which is fully platform independent. In order to achieve compatibility with many software platforms universal ASCII-based communication protocol was choose. Very advantageous is possibility to send all implemented commands using generic terminal program that is contained in most operating systems. Each command can be divided up to five parts depending on actual function implementation.

Communication with DAQ device starts with character "~" followed by command name with fixed length of two characters (for example "AO" means set analog output). After it is first command parameter with length of one character (channel index), next character is space followed by second parameter (value) which can be in integer or floating point format. Because of device supports 16 analog input channels and for channel index is reserved only one character, channel indexes are send as hexadecimal number. Whole command must be terminated by CRLF sequence.

Communication protocol example is in Fig. 3, list of supported commands is provided in the Table II.

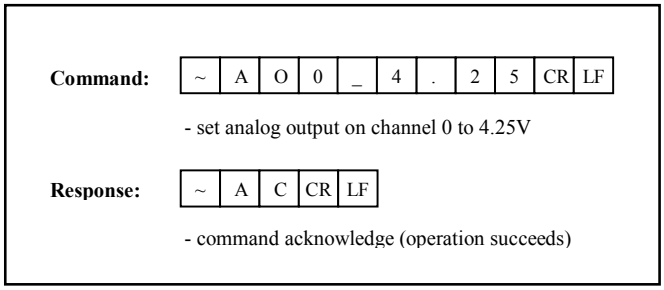


Fig. 3 Communication protocol example

Table II List of supported commands

| Command string | Description |
|---|---|
| ~AI<channel_index><br><CRLF> | Start analog to digital conversion on specified channel. Returned value is in Volts.<br>Device response:<br>AI<channel_index>=<value>CRLF |
| ~AO<channel_index><br><value><CRLF> | Set analog output on specified channel to specified value in Volts.<br>Device response:<br>AC<CRLF> |
| ~DI<channel_index><br><CRLF> | Read logical state of specified digital input.<br>Device response:<br>DI<channel_index>=<value>CRLF |
| ~DIA<CRLF> | Read all 8 digital inputs state (byte access).<br>Device response:<br>~DIA=<value><CRLF> |
| ~DO<channel_index><br><value><CRLF> | Set digital output to specified logical state.<br>Device response:<br>AC<CRLF> |
| ~DOA<value><br><CRLF> | Set all 8 digital outputs to specified value (byte access).<br>Device response:<br>AC<CRLF> |
| ~TM1<value><br><CRLF> | Set PWM period to specified value in seconds.<br>Device response:<br>AC<CRLF> |

Data acquisition device sends after internal command processing response string which format depends on actual command type. Commands writing data to device are acknowledged by response string "AC<CRLF>" in case of success. Commands reading data from device (for example analog input value in Volts) are acknowledged by response string containing data source and corresponding value: "AI0=1.25V<CRLF>". In case of error device returns string "ER<error_number><CRLF>" where error number provides information about error reason:

- 1 – unknown command
- 2 – command parameter 1 is out of range
- 3 – command parameter 2 is out of range
- 4 – PWM is active – period cannot be changed

## IV. DATA ACQUSITION DEVICE SOFTWARE SUPPORT

Although communication protocol is very simple and easy to understand it is more comfortable in a control application to call functions which can automatically generate commands for the data acquisition device and consequently process its response. This simplification results in faster program development and reduction of debugging time. For the portable data acquisition device was created support program library for Matlab 6.5 and higher versions software environment which is installed on computers in the laboratory of automatic control.

Created library incorporates all functions implemented in the device firmware including error processing. Each function is available in separate m-file, so it is very simple to modify them by the user. In the Table III are listed all implemented library functions for Matlab 6.5 environment.

For device testing and diagnosis, a DAQ test utility in MS Visual C++ 6.0 was created. This program can test all functions of the DAQ device and may be very helpful for testing wire connections to the monitored or controlled system. Main window of diagnostic utility is depicted in the Fig. 4. Left part of the window "Analog inputs readings" contains 16 edit boxes indicating actual voltage levels applied in the analog inputs. "Digital inputs readings" fields show same information but for digital inputs. The window part "Digital output setting" contains 8 buttons for changing the logical state of the each digital output. And finally last control "Analog output setting" providing slider for setting the output voltage level on analog output channel.

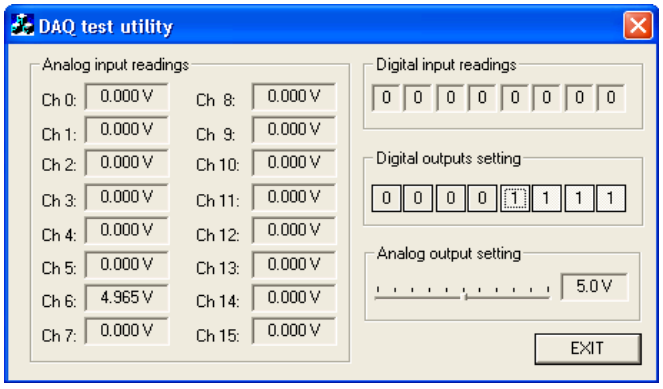

Fig. 4 Data acquisition device test utility

Table III Implemented library functions for Matlab environment

| Function | Description |
|---|---|
| *spobj* = open_device (*comm*) | Opens device connected to specified serial port *comm* ( for example "COM1", "COM2", …) and returns serial port object *spobj*. |
| close_device (*spobj*) | Closes device specified by *spobj* and removes serial port object from memory. |
| set_digital_out (*spobj*, *channel*, *value*) | Sets specified digital channel (0 to 7) to desired value (0 or 1). When floating point value in the range (0; 1) is specified, PWM modulation is activated. |
| set_analog_out (*spobj*, *channel*, *value*) | Sets analog output on specified channel to desired value in volts <0; 10V>. Function accepts values in floating-point format. |
| *value* = get_digital_in (*spobj*, *channel*) | Function returns state of the selected digital channel. |
| *value* = get_analog_in (*spobj*, *channel*) | Function returns voltage in the range <0; 10V> measured on specified analog input (0 to 15). |
| set_pwm_period (*spobj*, *pwm_period*) | Sets PWM period to specified value in seconds <0; 10s> |

Implementation of the library functions in user program is very simple. It can be divided into the several basic steps:

1) Open data acquisition device connected to specified serial port of the computer, e.g.: *s1 = open_device ('COM1')* if DAQ device is connected to COM1 port. Function returns serial port object which is stored to *s1*. This serial port object is required as argument of other library functions.
2) Use library function required by your application, e.g.: *voltage = get_analog_in (s1, 0)* where first argument is serial port object returned in previous step by open_device function and second argument is analog input channel number in range <0; 15>. Function returns voltage measured on channel 0.
3) Before the program will exits it is important to correctly cleanup serial port data structures from memory. This task performs function close_device, e.g.: *close_device (s1)*.
4) Program can finish now.

## V. Verification and Results

DAQ device with RTMON was tested on educational laboratory model of heating plant system with one temperature measurement output channel with unified analog output 0 – 10 V and one digital input channel for heating element control. Data acquisition unit was connected with standard personal computer via RS232 serial communications interface.

For educational and demonstration purposes was created simple application with graphical user interface using GUI design environment GUIDE running in Matlab 7.3 environment. The software supports step response measurement of the system and control of the controlled variable using digital PID controller. All measured data are automatically saved to the workspace in the matrix form and to the user definable text file with format suitable for import to spreadsheet processor.

After the program is executed by command "start_main" in the command window of the Matlab environment the main window depicted in the Fig. 5 will appear. The window is divided into the two parts – left part is dedicated to displaying measured system variables in the form of auto-scale graph and right part contains all necessary control components for program configuration and control.

Step measurement panel contains three text boxes for entering following parameters: actuating signal vector (u_vect), actuating signal change time vector (t_vect) and sampling period ($T_S$). After entering desired values measurement can be started by pressing "Start" button. PSD controller panel enables to parameterize vector of set point values (w_vect), set point change time vector (t_vect), controller sampling period (TS) and regulator parameters $q_0$, $q_1$ and $q_2$.

The implemented digital PID controller is described by discrete transfer function (1); actuating signal value $u(k)$ is computed by equation (2).

$$G_R(z)=\frac{U(z)}{E(z)}=\frac{q_0+q_1z^{-1}+q_2z^{-2}}{1-z^{-1}} \tag{1}$$

$$u(k)=u(k-1)+q_0e(k)+q_1e(k-1)+q_2e(k-2) \tag{2}$$

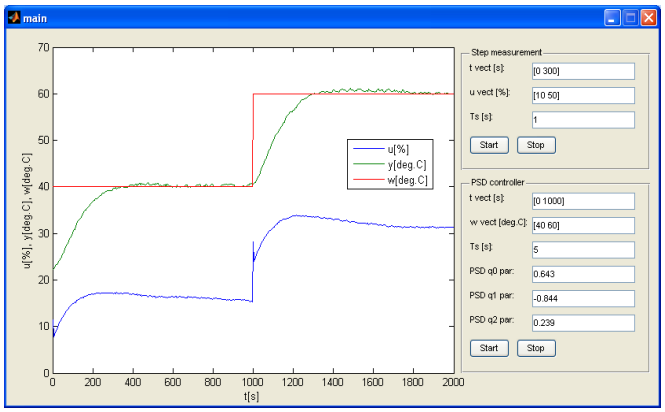

Fig. 5 Demonstration program for Matlab environment

Step response measurement of the heating plant model is depicted in the Fig. 6. It was measured with actuating signal change from 20 % to 50 % of its maximum value. Figures 7 and 8 shows example control processes achieved with PS and PSD controllers.

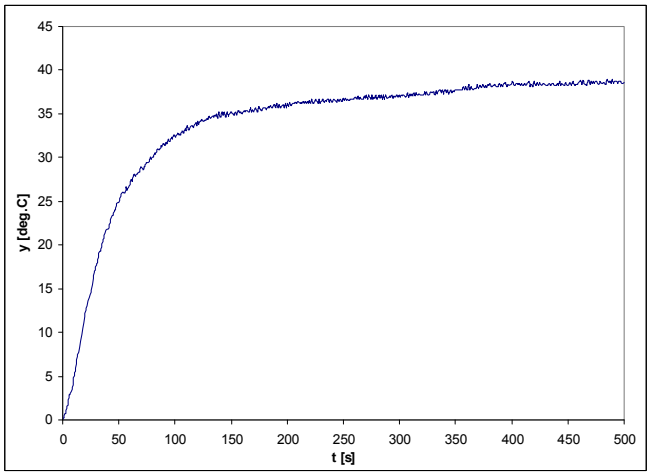


Fig. 6 Measured step response of the controlled system

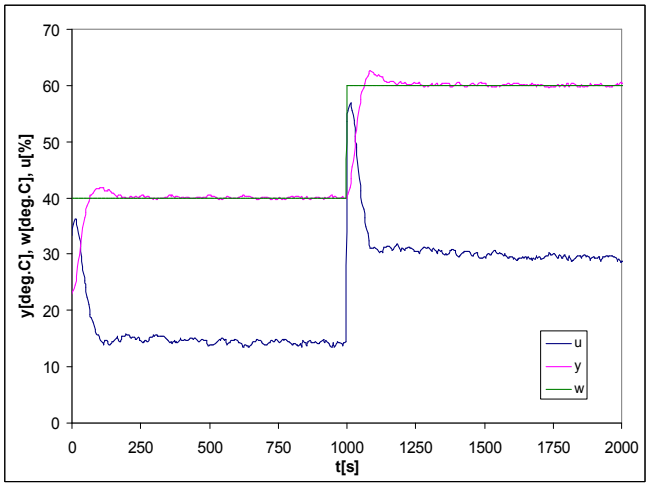


Fig. 7 PS controller control process

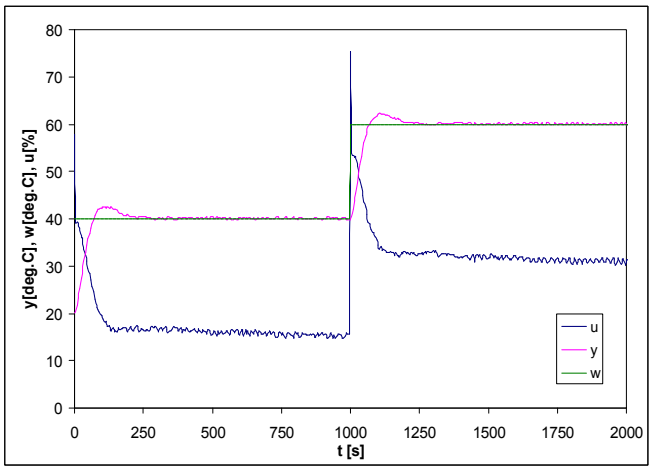


Fig. 8 PSD controller control process

## VI. CONCLUSION

This paper presented our simple real-time operating system for Freescale HCS08 microcontrollers and an application of this system in portable data acquisition unit. The system is used as a teaching aid for lessons of MCU programming. Its interface is based on older version of operating system for PC and HC11 microcontroller. However, the internals of the system were written completely from the scratch to allow it to work with limited data and code memory of small 8-bit microcontrollers. RTMON is pre-emptive multitasking system which allows defining processes up to certain number (default is 10) and running these processes either in infinite loops or periodically with a given period. System services are very simple due to the limited memory of the target microcontrollers and intended use of the system, but still the system provides the advantage of easy implementation of embedded system as a set of independent, concurrently running tasks.

RTMON proved to be functional during the lessons at our department, where it is used to demonstrate to students the basics of programming applications with operating systems, and it was also used in design of portable data acquisition unit DAQ. This device was also developed at our department and is used for control and monitoring related tasks. It is designed with respect to possible battery operation enabling measurement in areas where power source is not available. It provides sixteen analog inputs with 12-bit resolution, eight TTL compatible digital inputs and outputs protected against electrostatic discharge and overloading and one analog output channel equipped with 12-bit D/A converter. Communication with supervision system is realized with RS232 serial interface. It uses universal ASCII-based communication protocol which can be easily implemented in many software environments.

## REFERENCES

[1] Morton, T. D., *Embedded Microcontrollers*, Prentice Hall, 2001.
[2] FreeRTOS, The FreeRTOS Project, [Online]. Available: http://www.freertos.org
[3] Micrium, Micrium RTOS and Tools, [Online]. Available: http://micrium.com/page/products/rtos/os-ii
[4] Freescale Semiconductor, M68HC08 Microcontrollers: MC68HC908 GP32 Data Sheet. [Online]. Available: http://www.freescale.com
[5] Freescale Semiconductor, CPU08 Central Processor Unit Reference Manual, rev.4. [Online]. Available: http://www.freescale.com
[6] Burr-Brown, DAC7611: 12-Bit Serial Input Digital-to-Analog Converter. [Online]. Available: http:// www.burr-brown.com/.
[7] Linear Technology, LTC1286/LTC1298 Micropower Sampling 12-Bit A/D Converters, 1994. [Online]. Available: www.linear.com
[8] Dostálek, P.; Vašek, V.; Dolinay, J. "Design and implementation of portable data acquisition unit in process control and supervision applications", In proceedings of the 13th WSEAS International Conference on CIRCUITS, Rhodes 2009, pp. 799-808, ISSN 978-960-474-096-3.
[9] Dolinay, J.; Vašek, V.; Dostálek, P. "Implementation and Application of a Simple Real-time OS for 8-bit Microcontrollers", In proceedings of the 10th WSEAS International Conference on ELECTRONICS, HARDWARE, WIRELESS and OPTICAL COMMUNICATIONS (EHAC '11), Cambridge 2011, pp. 023-026, ISSN 1792-8133.