Systemy czasu rzeczywistego są coraz częściej wykorzystywane przez konstruktorów urządzeń mikroprocesorowych. Po przełamaniu pierwszej bariery, którą jest zrozumienie mechanizmów działania RTOS i nieco inne konstruowanie algorytmów oprogramowania okazuje się, że stosując takie systemy można osiągnąć wiele korzyści.

KaRTOS

8-bitowe jądro czasu rzeczywistego (4)

W tej części opisu systemu KaRTOS zostanie zaprezentowany moduł obsługi przetwornika analogowo–cyfrowego – KaRTOS_ADC. Z jego pomocą możemy w prosty sposób dokonywać pomiarów wielkości analogowych, tak powszechnych w otaczającym nas świecie. Tradycyjnie już prezentacją możliwości opisywanego modułu będzie aplikacja demonstracyjna: tym razem KaRTOS TERMO. Już sama nazwa sugeruje jej przeznaczenie, aczkolwiek nadmienić należy, że jest to implementacja o rozszerzonej funkcjonalności. Oprócz klasycznego termostatu z interaktywnym menu udostępnianym przy pomocy terminala, aplikacja posiada również funkcję sterownika klimatyzacji oraz termometru cyfrowego. Zapraszamy zatem do zapoznania się z kolejną odsłoną systemu KaRTOS.

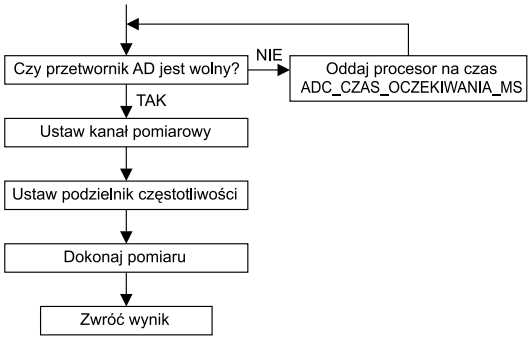
Czy w ogóle potrzebujemy przetwornika?

Środowisko, w którym żyjemy (my ludzie i mikrokontrolery także) ma charakter analogowy. O ile człowiek jest przystosowany do przetwarzania sygnałów o takim charakterze, o tyle mikrokontrolery od początku swego istnienia operują na danych cyfrowych. Konieczne jest zatem stworzenie mostu, który połączyłby dwa różne światy: nasz analogowy oraz ich cyfrowy. I tutaj właśnie spotykamy bohatera naszego dzisiejszego artykułu. Przetwornik A/C (ADC – *Analog to Digital Converter*) próbkuje sygnały analogowe i zamienia zgodnie z zaimplementowanym algorytmem na dane cyfrowe, na których może już operować mikroprocesor. Początkowo przetworniki A/C istniały głównie jako samodzielne wyspecjalizowane układy peryferyjne połączone z jednostką nadrzędną za pomocą magistrali zewnętrznej. Obecnie autonomicznymi układami pozostały głównie przetworniki z tzw. górnej półki, mające zastosowanie przede wszystkim w multimediach wraz z procesorami DSP.

Przetworniki A/C o przeciętnych parametrach (częstotliwość próbkowania rzędu kilku/kilkunastu kiloherców i rozdzielczość 8, 10 lub 12 bitów) są również dostępne jako jednostki samodzielne, aczkolwiek w ostatnich latach nastąpił proces migracji tych układów do wnętrza struktury półprzewodnikowej układów programowalnych. Lwia część obecnie produkowanych mikrokontrolerów posiada „na pokładzie" zintegrowany blok przetwornika A/C, co znacząco ułatwia i przyspiesza jego zastosowanie, a jednocześnie obniża koszty systemu jako całości. Przyjrzyjmy się zatem, jak system KaRTOS udostępnia aplikacjom 10–bitowy przetwornik ADC znajdujący się w mikrokontrolerze ATMega8(L) (i nie tylko).

KaRTOS_ADC – systemowy moduł obsługi przetwornika analogowo–cyfrowego

Aby dowiedzieć się czegoś o obsłudze przetwornika A/C w systemie KaRTOS, należy zajrzeć do pliku *KaRTOSAdc.h*, który znajduje się w katalogu *\KaRTOS*. Za pomocą zmiennych kompilatora możemy skonfigurować układ do pracy zgodnie z wymaganiami naszej aplikacji. I tak, kolejne zmienne oznaczają:



Rys. 12. Algorytm dokonywania pomiaru w funkcji *KaRTOSAdcPomiar*

`ADC_CZAS_OCZEKIWANIA_MS` – określa czasowy interwał kontroli stanu zajętości przetwornika A/C (**rys. 12**).

`ADC_PORT_MASK` – jest maską bitową określającą, które piny dedykowanego portu mikrokontrolera posłużą jako wejścia przetwornika A/C. Jedynka na danej pozycji oznacza, że pin ten zostanie skonfigurowany jako wejście. Dla przykładu: piny portu C kontrolera ATMega8(L) od PC.0 do PC.5 mogą być multipleksowanymi wejściami przetwornika A/C. Przypisanie zmiennej `ADC_PORT_MASK` wartości 0x05 oznacza, że podczas inicjalizacji ADC piny PC.0 oraz PC.2 zostaną skonfigurowane jako wejścia.

`REF_VREF` – deklarując tę zmienną ustalamy, że napięcie odniesienia dla ADC jest pobierane z pinu AREF kontrolera.

`REF_AVCC` – deklarując tę zmienną ustalamy, że napięcie odniesienia dla ADC jest pobierane z pinu AVCC kontrolera.

`REF_INTERNAL256` – deklarując tę zmienną ustalamy, że napięcie odniesienia dla ADC jest pobierane z wewnętrznego źródła napięcia odniesienia o wartości 2,56 V.

Oczywiście w poprawnie skonfigurowanym systemie tylko jedna ze zmiennych `REF_xxx` może zostać zadeklarowana (pozostałe powinny być zakomentowane).

`RESULT_ADJUST_LEFT` – zadeklarowanie tej zmiennej powoduje wyrównanie wyniku przetwarzania do najstarszego bitu słowa szesnastobitowego. Oznacza to, że wynik dwunastobitowy jest przesuwany w lewo o cztery bity, a najmłodsze pozycje uzupełniane są zerami. Zabieg ten np. upraszcza wyciąganie średniej z serii pomiarów niewielkich wartości redukując błędy zaokrągleń.

Aby móc korzystać z funkcji zawartych w bloku systemowym KaRTOS_ADC należy włączyć go do kompilowanej aplikacji. Dokonujemy tego deklarując zmienną kompilatora o nazwie `KaRTOS_ADC_ON` w pliku *\KaRTOS\ATMega8\KaRTOS.conf*. Odpowiednia linia kodu (`#define KaRTOS_ADC_ON`) jest już zawarta w wymienionym pliku i wystarczy zgodnie z potrzebą „zakomentować" ją, bądź pozostawić „aktywną".

System KaRTOS w wersji 3.01 udostępnia dwie funkcje służące do obsługi układu przetwornika. Są to:

void KaRTOSAdcInit(void) – inicjalizuje przetwornik oraz konfiguruje odpowiednie piny mikrokontrolera zgodnie z wartością zmiennej kompilatora `ADC_PORT_MASK`.

u16 KaRTOSAdcPomiar(u08 u08Admux, u08 u08Adps) – funkcja wykonuje pomiar w zadanym kanale i z zadaną częstotliwością zegara taktującego. Przyjmuje następujące parametry:

- `u08Admux` – określa próbkowany kanał zgodnie z **tab. 3**.
- `u08Adps` – określa częstotliwość taktującą układ ADC. Częstotliwość ta jest otrzymywana przez podział sygnału zegara taktującego mikrokontroler zgodnie z **tab. 4**.

Zwracanym rezultatem działania funkcji jest wartość odczytana z przetwornika wyrównana do prawej lub lewej strony słowa szesnastobitowego zgodnie z ustawieniem zmiennej `RESULT_ADJUST_LEFT`.

Na **list. 6** przedstawiono kompletny kod umożliwiający dokonywanie cyklicznych pomiarów w kanale zerowym (ADC0 – na pinie PC.0.) z okresem 10 sekund.

Oczywiście należy pamiętać o odpowiednim ustawieniu zmiennych kompilatora w pliku *\KaRTOS\KaRTOSAdc.h* (np. jak na **list. 7**) oraz uruchomieniu modułu KaRTOS_ADC pozbawiając znaków komentarza linię `#define KaRTOS_ADC_ON` w pliku *\KaRTOS\ATMega8\KaRTOS.conf*.

Sposób dokonywania pomiaru wartości analogowych zaimplementowany w funkcji *KaRTOSAdcPomiar* został pokazany na diagramie z rys. 12. Na wstępie funkcja sprawdza czy blok przetwornika A/C jest wolny (czy zakończyła się ewentualna poprzednia konwersja). Jeśli nie, oczekuje czas określony zmienną `ADC_CZAS_OCZEKIWANIA_MS` i dokonuje kolejnego sprawdzenia. Po uzyskaniu pozytywnego wyniku ustawia kanał oraz dzielnik i dokonuje pomiaru. Zwracany jest 12–bitowy wynik konwersji w 16–bitowej zmiennej.

Tab. 3. Konfiguracja kanałów przetwornika ADC z pośrednictwem parametru `u08Admux`

Lp.	`u08Admux`	Próbkowany kanał
1	0	ADC0...PC.0
2	1	ADC1...PC.1
3	2	ADC2...PC.2
4	3	ADC3...PC.3
5	4	ADC4...PC.4
6	5	ADC5...PC.5
7	6	ADC6
8	7	ADC7
9	14	1,23 V
10	15	0 V – GND

Tab. 4. Podział częstotliwości zegara systemowego w zależności od parametru `u08Adps`

Lp.	`u08Adps`	dzielnik
1	0	2
2	1	2
3	2	4
4	3	8
5	4	16
6	5	32
7	6	64
8	7	128

List. 6. Program dokonujący cyklicznych pomiarów w kanale ADC0

```
KaRTOSAdcInit() ; //-inicjalizacja ADC
for(;;)           //-pozostań w nieskończonej pętli
{
//- dokonaj pomiaru
 wart_16_bit=KaRTOSAdcPomiar(0,7);
//- odczekaj zadany okres czasu
 TimeSleepms(10000);
};
```

List. 7. Program dokonujący cyklicznych pomiarów w kanale ADC0

```
#define ADC_CZAS_OCZEKIWANIA_MS    1
#define ADC_PORT_MASK              0x01
//#define REF_VREF
#define REF_AVCC
//#define REF_INTERNAL256
//#define RESULT_ADJUST_LEFT
```

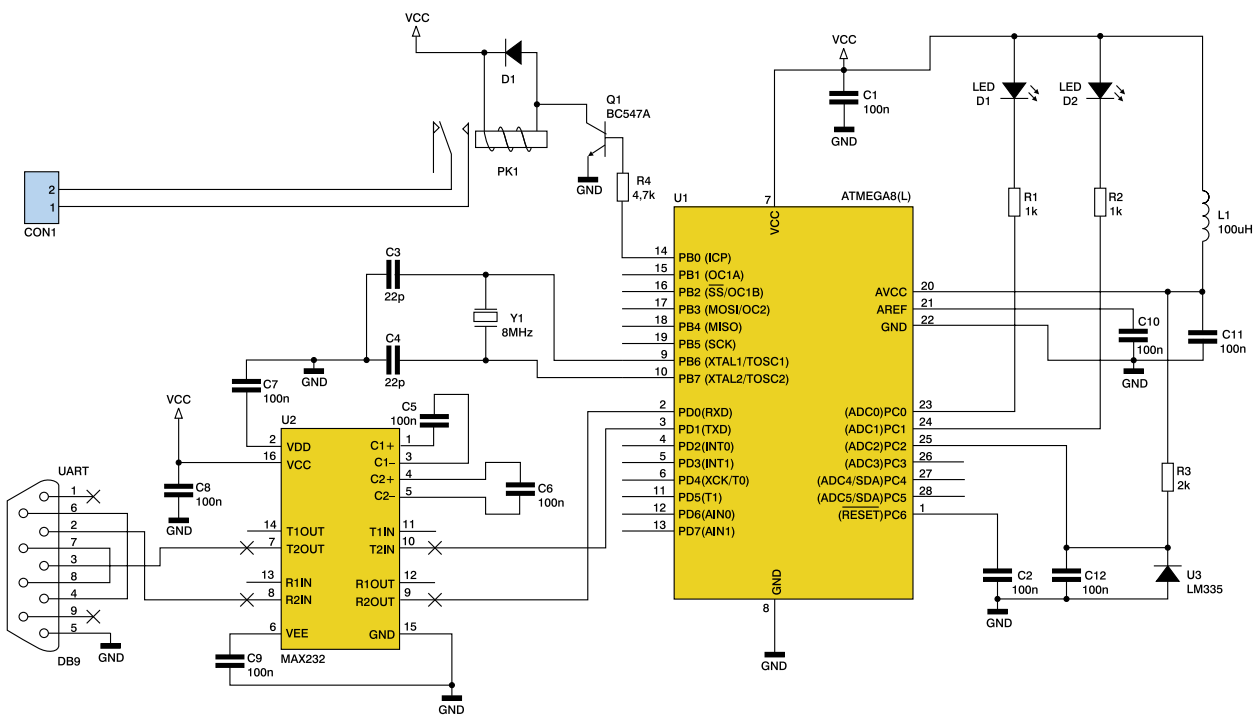
Moduł KaRTOS TERMO

Kolejna aplikacja, którą napiszemy dla systemu KaRTOS będzie miała praktyczne zastosowanie. Mimo, iż jej głównym celem jest dydaktyka oraz prezentacja środowiska systemu KaRTOS, to po niewielkich udoskonaleniach zgodnych z inwencją Czytelnika stanie się w pełni funkcjonalnym samodzielnym urządzeniem. Pewną modyfikację aplikacji polegającą na możliwości zapamiętania ustawień sterownika przeprowadzimy również „oficjalnie" w kolejnej części artykułu przy okazji prezentacji bloku obsługi wewnętrznej nieulotnej pamięci EEPROM kontrolera.

Jak już nadmieniono wcześniej, KaRTOS TERMO to konfigurowany z terminala poprzez port szeregowy sterownik utrzymujący stałą temperaturę określonego środowiska. Może on pracować w dwóch trybach: jako sterownik elementu grzejnego lub chłodzącego zasilanych z sieci 230 VAC i nie tylko. Poza tym wskazuje również aktualną temperaturę oraz wyświetla parametry i tryb pracy sterownika.

Na **rys. 13** przedstawiono schemat układu, który posłuży do zrealizowania naszego projektu. Znajdujemy tam kilka bloków funkcjonalnych:

- Blok generacji zegara taktującego z kwarcem o częstotliwości 8 MHz – Y1, C3, C4. Jego obecność nie jest konieczna, ponieważ programując kontroler do pracy z wewnętrznym oscylatorem otrzymamy taki sam efekt działania aplikacji.



Rys. 13. Schemat układu dla aplikacji KaRTOS TERMOSTAT

- Blok komunikacji z terminalem – konwerter napięć z układem U2 i elementami pomocniczymi.
- Blok umożliwiający poprawną pracę przetwornika A/C – L1, C11, C10.
- Blok pomiaru temperatury – U3, R3, C12.
- Blok sygnalizacyjny – D1, D2, R1, R2.
- Blok wykonawczy – PK1, D1, Q1, R4, CON1. Zamiast pokazanego na schemacie bloku wykonawczego można oczywiście zastosować wiele innych rozwiązań w zależności od potrzeb. Może to być np. triak, tyrystor lub tranzystor mocy sterujący modułami Peltiera na napięcie 12 V do klimatyzacji samochodowej. Pokazane rozwiązanie należy potraktować jako przykładowe i niekoniecznie optymalne.

Na schemacie nie zamieszczono bloku zasilania, ponieważ w zależności od docelowego przeznaczenia układu, będzie on miał różną postać. Powinien dostarczać napięcie o wartości 5 V, pobierany prąd nie powinien przekroczyć 100 mA (przy modyfikacji bloku wykonawczego należy to zweryfikować). Na schemacie brak jest również złącza SPI programującego mikrokontroler. Zrezygnowaliśmy z umieszczania go na schemacie pozostawiając Czytelnikowi „trud" jego podłączenia, ponieważ jest to uzależnione od typu wykorzystywanego programatora..

Konfiguracja zadań

Nasza aplikacja będzie się składać z trzech zadań, dlatego w pliku *main.c* należy je uruchomić (**list. 8**).

Uruchomienie modułów systemowych

Moduły systemowe, których będziemy potrzebować uruchomimy deklarując odpowiednie zmienne w znanym już pliku *KaRTOS\ATMega8\KaRTOS.conf*, jak to pokazano na **list. 9**.

Konfiguracja UART–a jest identyczna jak w poprzednich projektach, z prędkością transmisji równą 38,4 kbps, jednym bitem startu, jednym bitem stopu, bez bitu parzystości i kontroli przepływu – plik *KaRTOS\ATMega8\KaRTOSUart.h* (**list. 10**).

Konfiguracja pinów mikrokontrolera

Spojrzenie na schemat z rys. 13 pozwala zorientować się w potrzebnej konfiguracji pinów kontrolera. I tak: PB.0 – wyjście; PC.0 i PC.1 – wyjście; PC.2 – wejście ADC. Zatem w pliku *KaRTOS\ATMega8\nitm8.c* powinny znaleźć się linie kodu jak poniżej:

```
DDRB=0x01;
PORTB=0x00;
DDRC=0x03;
PORTC=0x03;
```

Implementacja kodu zadań

Jak już nadmieniono, stworzymy trzy zadania, które zajmą się odpowiednio:

Task_1 – wykonywaniem co sekundę pomiaru temperatury, obliczaniem średniej z czterech pomiarów, wyświetlaniem informacji na terminalu.

List. 8. Linie kodu uruchamiające zadania opisywanej aplikacji

```
KaRTOSTaskInit(&KaRTOSIdleTask,1,255,50) ;
KaRTOSTaskInit(&(Task_1),TASK_1_PID,TASK_1_
PRIORITY,TASK_1_STACK);
KaRTOSTaskInit(&Task_2,TASK_2_PID,TASK_2_PRIORITY,TASK_
2_STACK);
KaRTOSTaskInit(&Task_3,TASK_3_PID,TASK_3_PRIORITY,TASK_
3_STACK);
```

W pliku main.h konfiguracja zadań może wyglądać np. tak:

```
#define TASK_1_PID              10
#define TASK_1_STACK            150
#define TASK_1_PRIORITY         10

#define TASK_2_PID              20
#define TASK_2_STACK            100
#define TASK_2_PRIORITY         20

#define TASK_3_PID              30
#define TASK_3_STACK            50
#define TASK_3_PRIORITY         30
```

List. 9. Instrukcje uruchamiające moduły systemowe

```
#define KaRTOS_UART_ON
#define KaRTOS_ADC_ON
#define KaRTOS_STRING
```

Wartości pozostałych zmiennych jak w poprzednich projektach:

```
#define SYS_STACK             20
#define NO_TASKS_RAM_ADDR     619
#define KaRTOS_1MS_OCR_WART   125
```

List. 10. Konfigurowanie układu UART

```
#define UART_ROZM_BUF_OUT           10
#define UART_ROZM_BUF_IN            5
#define UART_OKRES_WYSYLANIA_MS     1
#define UART_OKRES_ODBIERANIA_MS    1
```

Jeśli chodzi o konfigurację przetwornika A/C, to dokonujemy jej edytując plik *KaRTOS\KaRTOSAdc.h*. PC.2 jest wejściem pomiarowym, a napięcie odniesienia dla przetwornika pobierane jest z pinu AVCC (**list. 11**).

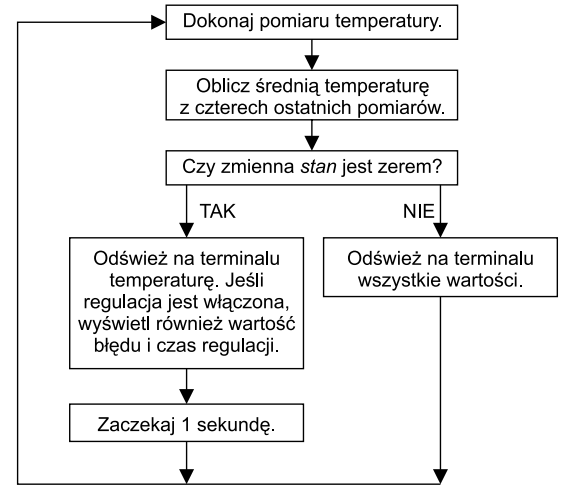
List. 11. konfigurowanie przetwornika A/C

```
#define ADC_CZAS_OCZEKIWANIA_MS     1
#define ADC_PORT_MASK               0x04
#define REF_AVCC
```

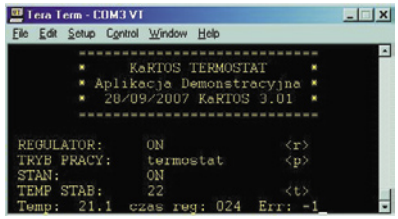
Task_2 – dokonywaniem zmian ustawień sterownika zgodnie z żądaniami nadchodzącymi z portu szeregowego.

Task_3 – sterowaniem poprzez port PB.0 układem wykonawczym.

Algorytm zaimplementowany w zadaniu Task_1 przedstawiono na **rys. 14**. Pierwszą operacją jest pomiar temperatury poprzez wywołanie funkcji znajdującej się w pliku *Task_1.c* o nazwie *ZmierzTemperature()*, a której kod przedstawiono na **list. 11**. Wyjaśnienia wymaga sposób przeliczania wartości znajdującej się w dwubajtowej zmiennej `tmpl` otrzymanej z przetwornika A/C na temperaturę wyrażoną w stopniach Celsjusza. Ponieważ napięcie odniesienia ADC ma wartość 5 V, a przetwornik rozdzielczość 10 bitów, zatem wartość napięcia odpowiadająca najmłodszemu bitowi jest równa: $5/2^{10}=5/1024=4{,}883$ mV. Użyty czujnik temperatury LM335 posiada charakterystykę o nachyleniu 10 mV/K, co oznacza, że np. w temperaturze pokojowej równej 25°C na czujniku odłoży się napięcie (273+25)*10 [mV]=2,98 [V]. Z powyższego wynika, że zmiana na najmłodszym bicie oznacza zmianę temperatury o „troszeczkę" poniżej 0,5°C (dokładnie 0,4883°C). Błąd wynosi dokładnie 2,34% (1–4,883 [mV]/5 [mV]=0,0234). I właśnie kod przedstawiony na list. 11 dokonuje korekty zmierzonej wartości o 2,4%. Dodatkowym zabiegiem poprawiającym dokładność przeliczeń jest wykonywanie operacji arytmetyczno–logicznych na wartościach dziesięciokrotnie większych. Oznacza to, że aby otrzymać temperaturę w stopniach Celsjusza, od wyniku w Kelwinach podawaną przez LM335 należy odjąć zaokrągloną wartość równą 2732 (0°C=273,15 K). Wynikowa temperatura jest oczywiście dziesięciokrotnie powiększona. Otrzymując przykładowo w wyniku wywołania funkcji wartość temperatury równą 218 należy zinterpretować ją jako 21,8°C.



Rys. 14. Algorytm zadania Task_1



Rys. 15. Uruchomiona aplikacja

Po dokonaniu pomiaru otrzymana wartość jest dopisywana do czteroelementowego bufora kołowego. Następnie korzystając ze zgromadzonych próbek liczona jest średnia wartość temperatury, która jest zapisywana do zmiennej globalnej `temp_akt`. Nadmienić należy, że pomimo iż na ekranie terminala wyświetlana jest temperatura z dokładnością do dziesiątej części stopnia Celsjusza, to regulacja odbywa się na podstawie wartości przechowywanej właśnie w zmiennej `temp_akt` o dokładności jednego stopnia.

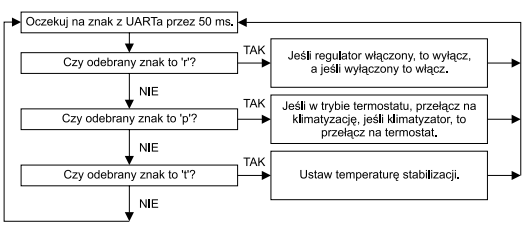
Kolejną czynnością dokonywaną przez omawiane zadanie jest odświeżenie informacji na terminalu. W zależności od wartości zmiennej `stan` jest to tylko uaktualnienie temperatury, bądź wszystkich parametrów sterownika, które można zobaczyć na **rys. 15** zawierającym zrzut ekranowy działającej aplikacji.

Działanie zadania Task_2 sprowadza się do skanowania portu szeregowego w oczekiwaniu na określony bajt będący żądaniem zmiany parametrów sterownika. W odpowiedzi na otrzymanie poprawnego znaku zadanie wykonuje:

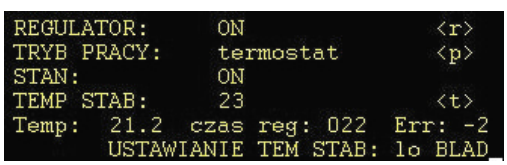
- zmianę trybu pracy termostat/klimatyzator – znak „p”;
- włącza/wyłącza regulator – znak „r”;
- ustawia temperaturę stabilizacji – znak „t”.

Algorytm działania zadania Task_2 przedstawiono na diagramie z **rys. 16**. Skomentowania może wymagać sposób wprowadzania temperatury stabilizacji. Po naciśnięciu przycisku „t” na klawiaturze sterownik oczekuje przez 10 sekund na wprowadzenie dwóch znaków, które zostaną zinterpretowane jako dwucyfrowa liczba. Wprowadzenie niepoprawnych danych (litery lub wartości większej od 90) spowoduje błąd i komunikat widoczny na zrzucie z **rys. 17**. Po poprawnym przyjęciu danych zobaczymy komunikat potwierdzający, jak na **rys. 18**.

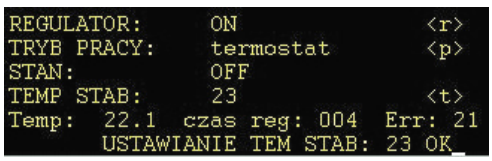
Pozostało do omówienia ostatnie z zadań Task_3, które realizuje algorytm sterowania blokiem wykonawczym. Sprowadza się to do włączania oraz wyłączania elementu wykonawczego (grzejnika lub klimatyzatora) w zależności od różnicy temperatur: rzeczywistej i zadanej oraz



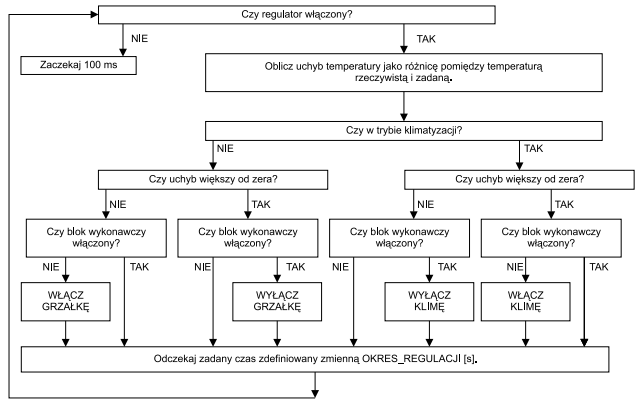
Rys. 16. Algorytm zadania Task_2



Rys. 17. Błąd podczas wprowadzania temperatury



Rys. 18. Temperatura wprowadzona poprawnie



Rys. 19. Algorytm realizowany przez zadanie Task_3

od trybu pracy sterownika – termostat czy klimatyzator (**rys. 19**). Jest nieskończoną pętlą wykonywaną z okresem, którego długość trwania ustala w sekundach wartość zmiennej kompilatora `OKRES_REGULACJI` zdefiniowanej w pliku *Task_3.h*. Ponieważ temperatura w naszej aplikacji należy do wartości wolnozmiennych, toteż okres ten może być ustalony na kilkadziesiąt sekund, a nawet na pojedyncze minuty. Maksymalna wartość to 255, ze względu na ośmiobitowy rozmiar licznika.

Działanie aplikacji

Po włączeniu zasilania na ekranie terminala zostaną wyświetlone informacje o stanie sterownika (rys. 15). Od góry w liniach są to kolejno:

REGULATOR – Stan regulatora – jeśli jest włączony, oznacza to, że proces regulacji trwa, a w ostatniej linii oprócz aktualnej temperatury wyświetlany jest także czas pozostały do kolejnego kroku regulacji i ostatnio obliczona różnica pomiędzy temperaturą zadaną i rzeczywistą. Zmiany dokonujemy wciskając klawisz „r”.

TRYB PRACY regulatora – termostat lub klimatyzator. Termostat włącza urządzenie wykonawcze, gdy aktualna temperatura jest poniżej zadanej, a klimatyzator odwrotnie. Zmiany dokonujemy wciskając klawisz „p”.

STAN – informuje o stanie elementu wykonawczego.

TEMP STAB – to wartość temperatury zadanej, którą można ustawić wciskając klawisz „t”.

Temp – jest to wartość aktualnej temperatury.

Czas reg – jest to czas, jaki pozostał do wykonania kolejnego kroku algorytmu sterującego zawartego w zadaniu Task_3. Wartość ta jest uaktualniana tylko jeśli regulator jest włączony.

Err – jest to wartość uchybu temperatury rzeczywistej w stosunku do oczekiwanej, obliczonego podczas wykonywania ostatniego kroku regulacji (wyrażony w stopniach). Wartość ta jest uaktualniana tylko jeśli regulator jest włączony.

Domyślnym stanem regulatora po włączeniu zasilania jest praca w trybie termostatu. Regulator jest wyłączony, a temperatura zadana ma wartość równą zero. Aby uruchomić regulator naciskając klawisz „r” musimy wpierw wprowadzić żądaną wartość temperatury oczekiwanej. Jeśli regulator jest włączony, to co sekundę odświeżana jest wartość temperatury i czas regulacji. Wartość Err jest uaktualniana w każdym kroku regulacji, a wszystkie pozostałe parametry aktualizowane są w momencie zmiany ich stanu.

Gotowy projekt jest dostępny jako plik *KaRTOS TERMO.zip*. Zawiera kod źródłowy z bogatymi komentarzami oraz pliki wynikowe *main.hex* i *main.bin* do zaprogramowania kontrolera.

Mariusz Żądło
iram@poczta.onet.pl

Autor zachęca Czytelników do kształtowania treści kolejnych odcinków cyklu. Napisz w e–mailu czy bardziej interesuje Cię teoria działania, czy raczej wolisz aby prezentowane były przykładowe aplikacje i projekty dla systemu KaRTOS. Inne uwagi również mile widziane.