Podstawy Systemów Wbudowanych

Wykład 6: Systemy czasu rzeczywistego

Angelika Tefelska   Dariusz Tefelski

Zakład Fizyki Jądrowej, Wydział Fizyki PW

6 kwietnia 2017

O czym będzie wykład...

1. RTOS
2. FreeRTOS
3. ChibiOS
4. Internet of things

RTOS

Definicja RTOS

System operacyjny czasu rzeczywistego (ang. real-time operating system, RTOS) – komputerowy system operacyjny, który został opracowany tak, by spełnić wymagania narzucone na czas wykonywania zadanych operacji. Systemy takie stosuje się jako elementy komputerowych systemów sterowania pracujących w reżimie czasu rzeczywistego - system czasu rzeczywistego. [a]

[a] Źródło: https://pl.wikipedia.org/

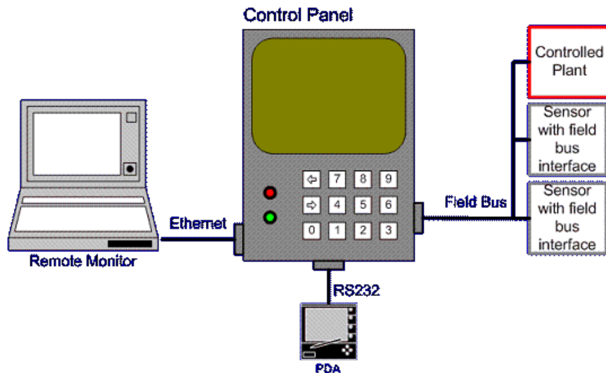
Zastosowanie

- w technice wojskowej - precyzyjne sterowanie rakietami.
- centrale telefoniczne
- lądowniki NASA
- ABS w samochodach
- i wiele, wiele innych

Kiedy potrzebujemy RTOS?

- Standardowy program składa się z pętli loop, której funkcje wykonują się na przemian. Gdybyśmy chcieli aby co określony czas wykonywały się pewne informacje to można skorzystać z timer-ów (patrz poprzednie wykłady).
- W prostych programach takie rozwiązanie w zupełności wystarczy. Natomiast jeśli pomiędzy wywołaniem przerwania wykonywanie są złożone operacje, na podstawie której mają być podjęte decyzje o sterowaniu pewnym modułem, to okazuje się, że system działa nieefektywnie.
- Kolejną konsekwencją zastosowania przerwań jest brak skalowania powstałego oprogramowania wraz z rozbudową układu.
- Rozwiązaniem dobrym przy złożonych systemach jest wykorzystanie RTOS, który wprowadza priorytet pewnych operacji.

Przykład typowego systemu kontrolnego



Układ składa się z:

- Systemu wbudowanego z kontrolnym terminalem.
- Czujników komunikujących się przez magistralę.
- Jakiegoś modułu kontrolnego np. silnik, płyta grzewcza itd.
- Kontrolnego panelu
- Jakiegoś serwera zapisującego dane na komputerze.

Przykład typowego systemu kontrolnego

1. Algorytm sterowania modułem kontrolnym, wykonywany przez system wbudowany:
    - Wysłanie zapytania do czujników (co 10us)
    - Odczyt danych z czujników. (5us)
    - Wykonanie kontrolnego algorytmu na podstawie, którego zostanie podjęta decyzja.
    - Wysłanie polecenia do modułu kontrolnego
2. Obsługa kontrolnego panelu:
    - Skanowanie klawiatury co 15ms w celu sprawdzenia czy użytkownik nie wpisał jakiś wartości.
    - Odświeżanie wyświetlacza LCD do 50ms.
    - Dioda zielona wskazuje, że system kontrolny działa bez zarzutu. Dioda czerwona wskazuje na pojawienie się błędu. Informacja, która dioda ma być zapalona jest wysyłana co 50ms.
3. Serwer systemu wbudowanego pobiera dane do zapisu co 1s.

Przykład typowego systemu kontrolnego

W przedstawionym systemie kontrolnym duże znaczenie ma sekwencja i czas wykonywania poszczególnych czynności. Ze względu na wymagania czasowe cały system możemy podzielić na 3 kategorie:

1. Ścisły czas wykonania - pobieranie informacji od czujników, sterowanie modułem kontrolnym.
2. Elastyczny czas wykonywania - moduły, które mają przedział czasowy między którym powinny być wykonywane np. dioda LED na panelu kontrolnym.
3. Najmniejszy priorytet czasu wykonywania - moduły, które mają podany maksymalny czas wykonywania, ale każde odchylenie podanego czasu o 100% jest akceptowalne np. panel LCD.

Implementacja systemu przy użyciu standardowych metod

Przykład implementacji algorytmu sterowania modułem kontrolnym na podstawie pseudo-kodu:

```
void PlantControlCycle( void )
{
    TransmitRequest();
    WaitForFirstSensorResponse();

    if( Got data from first sensor )
    {
        WaitForSecondSensorResponse();

        if( Got data from second sensor )
        {
            PerformControlAlgorithm();
            TransmitResults();
        }
    }
}
```

Implementacja systemu przy użyciu standardowych metod

```c
int TimerExpired;

void TimerInterrupt( void )
{
    TimerExpired = true; //Timer co 10us
}

int main( void )
{
    Initialise();
    for( ;; )
    {
        if( TimerExpired )
        {
            PlantControlCycle();
            TimerExpired = false;

            ScanKeypad();
            UpdateLCD();
            ProcessLEDs();
            ProcessRS232Characters();
            ProcessHTTPRequests();
        }
    }
}
```

Wady przedstawionego przykładu:

- Opóźnienie spowodowane transmisją danych z czujników przez magistralę wydłużą czas wykonywania funkcji PlantControlCycle. Ponadto wskazane rozwiązanie nie zapewnia wymaganego czasu wykonania funkcji związanych z panelem kontrolnym.
- Czas wykonania funkcji związanych z panelem kontrolnym również wpłynie na czas pomiędzy wykonywaniem się funkcji PlantControlCycle.
- Niekorzystnym zjawiskiem jest również różny czas wykonywania się pętli przy kolejnych iteracjach, który będzie związany m.in z funkcją ProcessHTTPRequests. Czas wykonywania się tej funkcji jest zaniedbywalnie mały gdy uzyskujemy od razu odpowiedź z serwera. Natomiast gdy odpowiedź z różnych przyczyn nie przyjdzie od razu to całkowity czas wykonania pętli diametralnie się zwiększy.
- Funkcje do komunikacji są obsługiwane tylko raz w czasie jednego cyklu pętli.

Alternatywne rozwiązanie - przykład 1

```
typdef enum eCONTROL_STATES
{
    eStart, //Zacznij nowy cykl
    eWait1, //Poczekaj na odpowiedz od pierwszego czujnika
    eWait2  //Poczekaj na odpowiedz od drugiego czujnika
} eControlStates;

void PlantControlCycle( void )
{
static eControlState eState = eStart;

    switch( eState )
    {
        case eStart :
            TransmitRequest();
            eState = eWait1;
            break;
```

Alternatywne rozwiązanie - przykład 1

```
        case eWait1;
                if( Got data from first sensor )
                {
                    eState = eWait2;
                }
                break;

        case eWait2;
            if( Got data from first sensor )
            {
                PerformControlAlgorithm();
                TransmitResults();

                eState = eStart;
            }
            break;
    }
}
```

Alternatywne rozwiązanie - przykład 2

```
int main( void )
{
int Counter = -1;

    Initialise();

    //Maszyna stanow
    for( ;; )
    {
        if( TimerExpired )
        {
            Counter++;

            switch( Counter )
            {
                case 0  : ControlCycle();
                          ScanKeypad();
                          break;
```

Alternatywne rozwiązanie - przykład 2

```
                    case 1  : UpdateLCD();
                              break;

                    case 2  : ControlCycle();
                              ProcessRS232Characters();
                              break;

                    case 3  : ProcessHTTPRequests();
                              Counter = -1;
                              break;
                }

                TimerExpired = false;
            }
        }
        return 0;
    }
```

Pokazane przykłady umożliwią redukcję utraconych cykli ale nie rozwiążą problemu zachowania ścisłego czasu wykonywania kluczowych operacji. Do tego niezbędny jest RTOS.

Wielozadaniowość

Priority	Tasks
2	PlantControlTask
1	RS232Task, KeyScanTask
0	IdleTask, LEDTask, WebServerTask

- Poszczególne czynności są zmieniane w zadania o różnym priorytecie. Zadania ze ścisłym czasem wykonywania mają największy priorytet.
- Zadania są blokowane aż do momentu wywołania ich przez zdarzenie. Zdarzenia mogą być: zewnętrzne (np. naciśnięcie przycisku na klawiaturze) lub wewnętrzne (upłynięcie określonego czasu).
- Tylko zadania o największym priorytecie nie są blokowane przez system. Gdy nadchodzi czas na ich wykonanie, system blokuje aktualnie wykonywane zadanie o mniejszym priorytecie.

Zalety i wady

- Segmentacja i elastyczny system, łatwy do rozszerzenia.
- Procesor przełącza zadania w zależności od potrzeby bez żadnych dodatkowych akcji.
- Brak utraconych cykli.
- Rozwiązanie wymaga stworzenia wielu zadań oraz zużywa wiele pamięci RAM. Z tego powodu zaleca się stosowanie RTOS na płytkach posiadających wiele pamięci RAM np. arduino mega.

Implementacja z wykorzystaniem FreeRTOS - Plant Control Task

```
#define CYCLE_RATE_MS       10
#define MAX_COMMS_DELAY     2

void PlantControlTask( void *pvParameters )
{
TickType_t xLastWakeTime;
DataType Data1, Data2;

    InitialiseTheQueue();

    // Inicjalizacja xLastWakeTime, ktora przechowuje
      ↪ informacje kiedy ostatni raz wykonalo sie
      ↪ zadanie
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        //Funkcja, ktora informuje jadro ze zadanie
          ↪ powinno byc wykonywane co 10ms.
        vTaskDelayUntil( &xLastWakeTime, CYCLE_RATE_MS );
```

Implementacja z wykorzystaniem FreeRTOS - Plant Control Task

```
            //Wyslanie zapytanie o dane do czujnikow.
            TransmitRequest();

            //Odbior danych z magistrali xFieldBusQueue i
                ↪ zapisanie do zmiennej Data1
            if( xQueueReceive( xFieldBusQueue, &Data1,
                ↪ MAX_COMMS_DELAY ) )
            {
                if( xQueueReceive( xFieldBusQueue, &Data2,
                    ↪ MAX_COMMS_DELAY ) )
                {
                    PerformControlAlgorithm();
                    TransmitResults();
                }
            }
        }

    }
```

Implementacja z wykorzystaniem FreeRTOS - RS232 Interface oraz Embedded Web Server Task

```
void RS232Task( void *pvParameters )
{
        DataTypeB Data;

    for( ;; )
    {
        if( xQueueReceive( xRS232Queue, &Data, MAX_DELAY ) )
        {
            ProcessSerialCharacters( Data );
        }
    }
}

void WebServerTask( void *pvParameters )
        {
        DataTypeA Data;

            for( ;; )
            {
                if( xQueueReceive( xEthernetQueue, &Data,
                    ↪ MAX_DELAY ) )
                {
                    ProcessHTTPData( Data );
                }
            }
```

Implementacja z wykorzystaniem FreeRTOS - Keypad Scanning Task

```c
#define DELAY_PERIOD 4

void KeyScanTask( void *pvParmeters )
{
char Key;
TickType_t xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );

        // Scan the keyboard.
        if( KeyPressed( &Key ) )
        {
            UpdateDisplay( Key );
        }
    }
}
```

Implementacja z wykorzystaniem FreeRTOS - LED Task

```
#define DELAY_PERIOD 1000

void LEDTask( void *pvParmeters )
{
TickType_t xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );

        if( SystemIsHealthy() )
        {
            FlashLED( GREEN );
        }
        else
        {
            FlashLED( RED );
        }
    }
}
```

Więcej na temat FreeRTOS można znaleźć na stronie: http://www.freertos.org/tutorial/.

FreeRTOS na Arduino

```
#include <Arduino_FreeRTOS.h>
// define two tasks for Blink & AnalogRead
void TaskBlink( void *pvParameters );
void TaskAnalogRead( void *pvParameters );

// the setup function runs once when you press reset or power the
    ↪ board
void setup() {
  // Now set up two tasks to run independently.
  xTaskCreate(
    TaskBlink
    ,  (const portCHAR *)"Blink"   // A name just for humans
    ,  128  // Stack size
    ,  NULL
    ,  2  // priority
    ,  NULL );
  xTaskCreate(
    TaskAnalogRead
    ,  (const portCHAR *) "AnalogRead"
    ,  128 // This stack size can be checked & adjusted by reading
        ↪  Highwater
    ,  NULL
    ,  1  // priority
    ,  NULL );
  // Now the task scheduler, which takes over control of
      ↪ scheduling individual tasks, is automatically started.
}
```

FreeRTOS na Arduino

```
void loop()
{
}
void TaskBlink(void *pvParameters)  // This is a task.
{
  (void) pvParameters;
  pinMode(13, OUTPUT);
  for (;;) // A Task shall never return or exit.
  {
    digitalWrite(13, HIGH);
    vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one
        ↪ second
    digitalWrite(13, LOW);
    vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one
        ↪ second
  }
}
void TaskAnalogRead(void *pvParameters)  // This is a task.
{
  (void) pvParameters;
  Serial.begin(9600);
  for (;;)
  {
    int sensorValue = analogRead(A0);
    Serial.println(sensorValue);
    vTaskDelay(1);  // one tick delay (15ms) in between reads for
        ↪ stability
```

Dlaczego ChibiOS?

- FreeRTOS jest najbardziej rozpowszechniony, ale ma zasadniczą wadę. Pamięć jest alokowana dynamicznie co powoduje duże zużycie pamięci RAM.
- ChibiOS jest oparty na Unix i jego zasadniczą zaletą jest statyczna alokacja pamięci. Ponadto charakteryzuje się kompaktowością i prostotą kodu.
- Twórcy ChibiOS stworzyli bardzo szczegółową książkę dla użytkowników, który jest dostępny na stronie: `http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:start`

Przykład użycia ChibiOS - deklaracja zadań

```
// Example to demonstrate thread definition, semaphores, and
    ↪ thread sleep.
#include <ChibiOS_AVR.h>

// The LED is attached to pin 13 on Arduino.
const uint8_t LED_PIN = 13;

// Declare a semaphore with an inital counter value of zero.
SEMAPHORE_DECL(sem, 0);
//------------------------------------------------------------
// Thread 1, turn the LED off when signalled by thread 2.

// 64 byte stack beyond task switch and interrupt needs
static THD_WORKING_AREA(waThread1, 64);

static THD_FUNCTION(Thread1, arg) {

  while (!chThdShouldTerminateX()) {
    // Wait for signal from thread 2.
    chSemWait(&sem);

    // Turn LED off.
    digitalWrite(LED_PIN, LOW);
  }
}
```

Przykład użycia ChibiOS - deklaracja zadań

```
//
    ↪ ------------------------------------------------------------
    ↪
// Thread 2, turn the LED on and signal thread 1 to turn the LED
    ↪ off.

// 64 byte stack beyond task switch and interrupt needs
static THD_WORKING_AREA(waThread2, 64);

static THD_FUNCTION(Thread2, arg) {
  pinMode(LED_PIN, OUTPUT);
  while (1) {
    digitalWrite(LED_PIN, HIGH);

    // Sleep for 200 milliseconds.
    chThdSleepMilliseconds(200);

    // Signal thread 1 to turn LED off.
    chSemSignal(&sem);

    // Sleep for 200 milliseconds.
    chThdSleepMilliseconds(200);
  }
}
```

Przykład użycia ChibiOS - deklaracja zadań

```
//
    ↪ ----------------------------------------------------------
    ↪
void setup() {

  chBegin(chSetup);
  // chBegin never returns, main thread continues with mainThread
      ↪ ()
  while(1) {
  }
}
//
    ↪ ----------------------------------------------------------
    ↪
// main thread runs at NORMALPRIO
void chSetup() {

  // start blink thread
  chThdCreateStatic(waThread1, sizeof(waThread1),
    NORMALPRIO + 2, Thread1, NULL);

  chThdCreateStatic(waThread2, sizeof(waThread2),
    NORMALPRIO + 1, Thread2, NULL);

}
//
    ↪ ----------------------------------------------------------
```

Przykład użycia ChibiOS - korzystanie ze wspólnych zmiennych

```
#include <ChibiOS_AVR.h>

const uint8_t X_PIN = 0;
const uint8_t Y_PIN = 1;
const uint8_t Z_PIN = 2;

//
    ↪ ---------------------------------------------------------
    ↪
// Shared data, use volatile to insure correct access.

// Mutex for atomic access to data.
MUTEX_DECL(dataMutex);

// Time data was read.
volatile uint32_t dataT;

// Data X value.
volatile int dataX;

// Data Y value.
volatile int dataY;

// Data Z value.
volatile int dataZ;
```

Przykład użycia ChibiOS - korzystanie ze wspólnych zmiennych

```
// Thread 1, high priority to read sensor.
static THD_WORKING_AREA(waThread1, 64);

static THD_FUNCTION(Thread1, arg) {

  while (1) {
    chThdSleepMilliseconds(13);

    // Use temp variables to acquire data.
    uint32_t tmpT = millis();
    int tmpX = analogRead(X_PIN);
    int tmpY = analogRead(Y_PIN);
    int tmpZ = analogRead(Z_PIN);

    // Lock access to data.
    chMtxLock(&dataMutex);

    // Copy tmp variables to shared data.
    dataT = tmpT;
    dataX = tmpX;
    dataY = tmpY;
    dataZ = tmpZ;

    // Unlock data access.
    chMtxUnlock(&dataMutex);
  }
}
```

Przykład użycia ChibiOS - korzystanie ze wspólnych zmiennych

```
//
    ↪ ------------------------------------------------------------
    ↪
// thread 2 - print data every second.
static THD_WORKING_AREA(waThread2, 128);

static THD_FUNCTION(Thread2, arg) {

  // Print count every second.
  systime_t wakeTime = chVTGetSystemTime();
  while (1) {
    // Sleep for one second.
    wakeTime += MS2ST(1000);
    chThdSleepUntil(wakeTime);

    // Lock access to data.
    chMtxLock(&dataMutex);

    // Copy shared data to tmp variables.
    uint32_t tmpT = dataT;
    int tmpX = dataX;
    int tmpY = dataY;
    int tmpZ = dataZ;

    // Unlock data access.
    chMtxUnlock(&dataMutex);
```

Przykład użycia ChibiOS - korzystanie ze wspólnych zmiennych

```
    Serial.print("dataAge: ");
    Serial.print(millis() - tmpT);
    Serial.print(" ms, dataX: ");
    Serial.print(tmpX);
    Serial.print(", dataY: ");
    Serial.print(tmpY);
    Serial.print(", dataZ: ");
    Serial.println(tmpZ);
  }
}
//
    ↪ ------------------------------------------------------------
    ↪
void setup() {
  Serial.begin(9600);
  chBegin(mainThread);
}
//
    ↪ ------------------------------------------------------------
    ↪
// main thread runs at NORMALPRIO
void mainThread() {
  // start blink thread
  chThdCreateStatic(waThread1, sizeof(waThread1),
                            NORMALPRIO + 2, Thread1, NULL);
  // start print thread
  chThdCreateStatic(waThread2, sizeof(waThread2),
```

- Internet of things (IoT) - sieć internetowa połączonych np. czujników, która zbiera i przekazuje informacje.
- Celem Internetu rzeczy jest stworzenie inteligentnych przestrzeni np. inteligentnych miast, inteligentnego transportu, sieci pomiarowych, budynków, systemów ochrony zdrowia, inteligentnych sieci energetycznych itd. Realizację tego zapewnić może odpowiednio opracowana technologia (hardware i software).
- Dużym ułatwieniem dla IoT byłoby zastosowanie systemu RTOS i zestandaryzowanie sposobu przekazywania danych. Jednym ze systemów operacyjnych stworzonych z myślą o IoT jest Contiki, który znalazł powszechne zastosowanie w: zarządzaniu światłami ulicznymi, monitoringu dzwięków, monitoringu poziomu radiacji oraz w alarmach. Więcej na stronie: http://contiki-os.org/support.html

Charakterystyka IOT:

- Wiele heterogenicznych urządzeń, począwszy od prostych sensorów sterowanych 8-bitowymi mikrokontrolerami, do wydajnych 32-bitowych procesorów z zaawansowanymi trybami oszczędzania energii
- Zarówno „duże" - standardowe systemy operacyjne jak i systemy przeznaczone dla sieci czujników nie spełniają wszystkich wymogów stawianych przed IoT.
- Wyraźnie pojawia się potrzeba standaryzacji i usunięcia konieczności programowania każdego z urządzeń w innym systemie tak aby współpracowały ze sobą.
- System przeznaczony dla IoT powinien pozwalać na obsługę różnych platform sprzętowych w ten sam sposób.
- System powinien wykorzystywać minimalnie zasoby sprzętowe, ułatwiać programowanie, zapewniać wielozadaniowość jak i zdolności systemów czasu rzeczywistego.
- Obecnie rozwija się „rynek" na różne konkurencyjne rozwiązania, które zwykle różnią się wsparciem dla konkretnych platform sprzętowych

Przykładem obecnie dosyć popularnego systemu skierowanego na rozwój IoT jest RIOT, który posiada wsparcie dla wielu platform sprzętowych, chociaż często nie każda funkcjonalność jest dostępna np. pod Arduino.

Strona projektu: https://www.riot-os.org/

Porównanie parametrów systemów, które najbardziej spełniają potrzeby IoT:

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	○	✘	○	✓	○	○
Tiny OS	<1kB	<4kB	✘	✘	○	✓	✘	✘
Linux	~1MB	~1MB	✓	✓	✓	✘	○	○
RIOT	~1.5kB	~5kB	✓	✓	✓	✓	✓	✓

Porównanie systemów, ○ - częściowe wsparcie ✗ - brak wsparcia, ✓ - pełne wsparcie

THAT'S ALL FOR TODAY....
ANY QUESTION??