Wyjątki C++ a mikrokontrolery

Na przykładzie implementacji wyjątków w systemie ISIX-RTOS

W wielu artykułach wspominałem o zaletach języka C++ w porównaniu do języka C oraz starałem się obalić mit mówiący o tym, że C++ nie nadaje się do pisania oprogramowania dla mikrokontrolera. Wielokrotnie mogliśmy się przekonać, że umiejętne stosowania języka C++ nie powoduje zwiększenia kodu wynikowego, a ścisłe przestrzeganie typów przez kompilator prowadzi do powstawania bardziej niezawodnych i bezpiecznych programów.

W niewielkich mikrokontrolerach mających 32…64 kB pamięci Flash, celowo pomijaliśmy zaawansowane mechanizmy prowadzące do zwiększenia zapotrzebowania na pamięć, takie jak wyjątki, czy RTTI (*Run Time Type Information*), tak aby minimalizować zajętość pamięci FLASH oraz RAM. Tworząc oprogramowanie dla „większych" mikrokontrolerów, jak np. rodzina Connectivity line STM32F107 czy Performance Line STM32F4, mających nawet kilkaset kB pamięci Flash oraz kilkadziesiąt kB pamięci RAM, możemy pokusić się o użycie dodatkowych funkcjonalności języka. Jednym z takich mechanizmów pozwalających zwiększyć niezawodność kosztem nieco większej zajętości zasobów pamięci jest użycie mechanizmu wyjątków. W artykule opiszę w jaki sposób wykorzystywać wyjątki C++ w systemie ISIX-RTOS oraz przedyskutuję ich ogólną wydajność.

Obsługa sytuacji wyjątkowych w C

Wyjątki są mechanizmem pozwalającym obsłużyć różne nietypowe sytuacje, które zdarzają się stosunkowo rzadko, i są sytuacjami nadzwyczajnymi. Zazwyczaj są to różnej kategorii błędy, z którymi mamy do czynienia podczas działania programu, na przykład błąd działania urządzenia, błąd alokacji zasobów np. pamięci itp. W języku C nie istnieje żaden specjalizowany mechanizm obsługi sytuacji wyjątkowych, a do ich obsługi najczęściej wykorzystywana jest wartość zwracana przez funkcję, która jest porównywana z pewnymi wartościami specjalnymi, reprezentującymi błąd. Typowym przykładem jest tutaj alokacja pamięci, która w programie napisanym w C wygląda, jak na **listingu 1**.

W wypadku niepowodzenia funkcja malloc() zwraca wartość specjalną NULL, która jest informacją o błędzie przydziału pamięci. Programista musi pamiętać, aby wraz z każdym wywołaniem malloc() zadbać o sprawdzenie czy alokacja powiodła się. Jeżeli w programie występuje wiele takich alokacji (co jest typowym przypadkiem), to mało komu wystarczy cierpliwości, aby obsłużyć

Listing 1. Przykładowa funkcja alokacji pamięci
```
struct MyObject *my = malloc( sizeof( struct MyObject) );
if (my == NULL)
{
  //Handle error
  return ERROR_ALLOC;
}
```

Listing 2. Fragment programu inicjalizujący układy peryferyjne
```
#define OK 0
#define FAIL -1

int init_device1(void)
{
    //Do something
    init_registers();
    unsigned device_bits = read_status_device();
    if( device_bits & 0x01 )
      return OK;
    else
      return FAIL;
}

void deinit_device1()
{
  //Deinitialize device
}

int init_all(struct DeviceStruct **dev)
{
  int result;
  result = init_device1();
  if( result == FAIL) return result;
  result = init_device2();
  if( result == FAIL)
  {
    deinit_device1();
    return FAIL;
  }
  /** Init structures **/
  *dev = malloc( sizeof( struct devStruct) );
  if( *dev == NULL )
  {
    deinit_device1();
    deinit_device2();
    return FAIL;
  }

}

int main()
{
   DeviceStruct *dev;
   int result = init_app(&dev);
   if( result == FAIL)
   {
      //Handle error
   }
}
```

Listing 3. Szkielet obsługi wyjątków

```
try
{
  //Kod ktory moze zglosic wyjatek
  init_all()
}
catch( exception1 &e1 )
{
  //obsluga wyjatku klasy exception1
}
catch( exception2 &e2)
{
  //Obsluga wyjatku klasy exception2
}
catch( ... )
{
  //Obsluga jakiegokolwiek wyjatku

}
```

wszystkie nieprawidłowe alokacje i jeszcze na dodatek – w żadnym miejscu programu nie zapomnieć o wspomnianym sprawdzeniu. W najlepszym wypadku brak sprawdzenia rezultatu alokacji pamięci w systemach operacyjnych z ochroną pamięci doprowadzi do błędu *Segmentation Fault*. Spójrzmy na typowy, przykładowy fragment programu inicjalizujący układy peryferyjne, gdy jeden zasób jest potrzebny do działania drugiego. Zamieszczono go na **listingu 2**.

Do prawidłowego działania programu jest konieczna poprawna inicjalizacja wszystkich zasobów, bez których nie może on działać poprawnie. W wypadku, gdy nie uda się nam prawidłowo zainicjalizować jednego z zasobów, musimy pamiętać, aby zwolnić wcześniejsze zasoby, które zostały prawidłowo przydzielone. Widzimy również cała masę sprawdzeń warunków, w każdym etapie wywołania, które też zajmują nie mało czasu procesora, jeżeli spojrzymy na to, że muszą być wykonywane wielokrotnie, praktycznie na każdym etapie wywołania. W tak zawiłym kodzie istnieje tutaj również wiele możliwości popełnienia błędu. Wystarczy jednokrotne pominięcie sprawdzenia warunku w dowolnym miejscu i cała logika obsługi błędów przestaje funkcjonować prawidłowo. Jest to jeden z klasycznych przykładów obalających teorie zwolenników C twierdzących, że programy w C wykonują się bardzo szybko i są proste i czytelne.

Obsługa sytuacji wyjątkowych w C++

Język C++ ma wbudowany mechanizm obsługi wyjątków. Jest on integralną częścią języka. Jest on pozbawiony wcześniej wspomnianych wad „ręcznej" obsługi sytuacji wyjątkowych. Do największych zalet należą tutaj brak konieczności ciągłego sprawdzania rezultatów, brak możliwości zignorowania zgłoszonego wyjątku (poprzez nie sprawdzenie zwróconego kodu błędu) oraz, co najważniejsze, automatyczne zwalnianie zasobów przez destruktory obiektów w momencie propagacji wyjątku. Wszystkie te czynniki powodują, że kod staje się znacznie bardziej czytelny oraz bardziej odporny na błędy. Wyjątek stanowi obiekt, który może być dowolnego typu, a w szczególności może być typem prostym POD, takim jak typ int. Możemy również zbudować hierarchię klas wyjątków czy skorzystać ze standardowej hierarchii klas wyjątków zawartych w pliku nagłówkowym *<exception>* oraz *<stdexcept>* Wyjątki w C++ obsługiwane są przez słowa kluczowe try, catch, throw. Mechanizm ich działania polega na tym, że kod w przypadku wystąpienia błędu zamiast zwracać rezultat funkcji informujący o błędzie rzuca wyjątek za pomocą wywołania throw exception(), gdzie exception() jest to klasa wyjątku, który będzie rzucony (najwygodniej jest w tym miejscu używać obiektów tymczasowych). Natomiast w innej części programu fragment, który może potencjalnie rzucić wyjątek zostaje umieszczony w sekcji try, po której następują klauzule przechwytywania poszczególnych klas wyjątków. Ostatnia klauzula catch(...) powoduje przechwycenie wszystkich pozostałych wyjątków, ale bez możliwości dostępu do obiektu wyjątku (**listing 3**).

Działanie mechanizmu wyjątków polega na tym że, rzucony wyjątek zwija stos bieżącej funkcji, niszcząc wszystkie zmienne lokalne, włącznie z wywołaniem ich destruktorów jeżeli zmienną lokalną stanowi obiekt klasy a nie typ prosty (np. int), a następnie próbuje skoczyć do najbliższej klauzuli catch która go obsłuży. Jeżeli w danej funkcji nie występuje klauzula catch, zwijany jest stos kolejnej funkcji która ją wywołała. Dzieję się tak aż do napotkania klauzuli catch, w kolejnej funkcji wywołującej. Jeżeli stanie się tak, iż zwijanie wyjątku dojdzie do funkcji main(), a funkcja ta również nie będzie zawierała klauzuli catch() odpowiadającej danemu wyjątkowi, nastąpi sytuacja którą nazywamy nie przechwyconym wyjątkiem. Niewyłapany wyjątek powoduje natychmiastowe wywołanie funkcji terminate(), która zwyczajowo po wypisaniu na standardowym wyjściu rodzaju wyjątku powoduje oddanie kontroli systemowi operacyjnemu, który kończy program. W przypadku znalezienia odpowiedniej klauzuli catch() dla danego wyjątku wykonuje się procedura jego obsługi, a po zakończeniu wykonywania kodu klauzuli catch, następuje dalsze wykonanie programu, które jest kontynuowane tuż za sekcjami catch, tak jak gdyby nigdy nic się nie wydarzyło. Naturalnie nie musimy w danej sekcji catch obsługiwać wszystkich rodzajów wyjątków, część z nich może być obsłużona w jednej funkcji, natomiast część w zupełnie innej. Dzięki temu, że w momencie wywołania wyjątku zwijany jest stos, oraz wywoływane są destruktory obiektów lokalnych, budując odpowiednio obiekty klas nie będzie potrzeby ręcznego zarządzania zasobami jak w przypadku języka C, a wszystko będzie odbywać się automatycznie. Spójrzmy jeszcze raz na przykład alokacji pamięci, gdzie w wypadku języka C trzeba było zarówno sprawdzić poprawność przydziału pamięci przez malloc(), jak i zadbać o odpowiednią inicjalizację struktury. W języku C++ do alokacji pamięci wykorzystujemy operator new, gdzie przykładowa alokacja z poprzedniego przykładu może wyglądać tak:

```
MyObject *my = new MyObject;
```

Porównując to kodem z początku poprzedniego punktu możemy zauważyć, iż jest dużo prostszy i bardziej czytelny. Jeżeli obiekt MyObject będzie klasą mającą konstruktor, to zostanie on automatycznie wywołany, co spowoduje wykonanie czynności początkowych zdefiniowanych w konstruktorze. Również standardowy operator new nie zwraca NULL, ale zgłasza wyjątek std::bad_alloc, którego nie musimy przechwytywać osobno w każdej funkcji. Wystarczy jedynie użycie klauzuli catch(std::exception &e) w funkcji main(), co powoduje znaczne zwiększenie czytelności kodu oraz redukuje konieczność ciągłego sprawdzania rezultatów funkcji, na każdym etapie wywołania. Należy tutaj przestrzec początkujących programistów przed nadmiernym używaniem systemu wyjątków i wykorzystaniem go nie do zgłaszania sytuacji nadzwyczajnych (wystąpienie błędu), a jako systemu przekazywania wartości. Zgłaszanie wyjątków jest stosunkowo czasochłonnym procesem z uwagi na wykonywanie czynności związanych ze zmianą przebiegu wykonania programu. Jednak umiejętne ich używanie, powoduje wzrost ogólnej wydajności programu, ponieważ podczas „prawidłowego" (bezbłędnego), przebiegu nie musimy wykonywać ciągłych i wielokrotnych sprawdzeń wartości.

W języku C++ istnieje kilka predefiniowanych klas wyjątków tak jak wspomniany wcześniej std::bad_alloc, które mogą być zgłaszane przez bibliotekę standardową. Klasą bazową dla wszystkich wyjątków jest tutaj klasa std::exception, której deklaracja wygląda następująco:

```
class exception {
public:
  exception () throw();
  exception (const exception&)
throw();
  exception& operator= (const
exception&) throw();
  virtual ~exception() throw();
  virtual const char* what()
const throw();
}
```

Najistotniejsza jest tutaj metoda wirtualna what(), która zwraca łańcuch tekstowy zawierający opis błędu. Wszystkie klasy wyjątków rzucane przez bibliotekę standardową jako klasę bazową wykorzystują exception, a zatem klauzula catch(exception &e)

powoduje przechwycenie wszystkich wyjątków, klas pochodnych które dziedziczą z tej klasy. Pozostałe wyjątki które mogą być zgłaszane przez bibliotekę standardową C++, podzielono na błędy logiczne wywodzące się z klasy bazowej logic_error (**tabela 1**) oraz błędy wykonania wywodzące się z klasy runtime_error (**tabela 2**).

Oczywiście każdy, z tych wyjątków, może być również zgłaszany nie tylko przez bibliotekę standardową, ale przez kod użytkownika. Bazując na powyższych klasach, hierarchia wyjątków może być rozbudowywana w razie potrzeb przez użytkownika.

Przykład praktyczny obsługi wyjątków w ISIX-RTOS

Po zapoznaniu się z podstawowymi wiadomościami teoretycznymi na temat wyjątków w C++ , pokażemy, że mogą być stosowane z powodzeniem również w przypadku nieco większych mikrokontrolerów. Pozwalając na znaczące podniesienie niezawodności działania programu, oraz przy stosowaniu z umiarem zapewniają również podniesienie ogólnej wydajności aplikacji. Posłużmy się teraz prostym przykładem (platforma STM32Butterfly), w którym jeden wątek będzie odpowiedzialny za mruganie diodą D1, i sygnalizował będzie jedynie prawidłowe działanie systemu operacyjnego. Natomiast drugi wątek, będzie zmieniał stan diody D2 na przeciwny w przypadku wciśnięcia klawisza joysticka OK. W przypadku wciśnięcia klawisza DOWN, lub UP, zgłaszany będzie wyjątek, który będzie symulował wystąpienie, błędu. Dodatkowo na linii PE0 wystawiany będzie stan 1 przez czas od momentu zgłoszenia wyjątku, do jego przechwycenia co pozwoli na zbadanie ogólnej wydajności obsługi wyjątków w GCC na architekturze CORTEX-M3. Kod programu przedstawiono na **listingu 4**.

Klasa led_blink odpowiedzialna jest za cykliczne mruganie diodą D1, które jest realizowane w pętli głównej wątku led_blink::main(). Mruganie diodą informuje nas o prawidłowym działaniu systemu operacyjnego i nie wymaga dalszego komentarza. Główna demonstracja mechanizmu wyjątków jest realizowana w klasie ledkey. W konstruktorze klasy led_blink porty PE.0 oraz PE.15 odpowiedzialne, odpowiednio: za pin testowy dla badania czasu wykonania wyjątku oraz sterowanie diodą LED, ustawiane są w kierunku wyjścia. Po wykonaniu konstruktorów, system operacyjny w osobnym wątku wykonuje metodę ledkey::main(). W sekcji try zawarto pętlę nieskończoną main(), która wywołuje metodę ledkey::execute_keycheck(), natomiast w sekcji przechwytywania wyjątków catch będziemy przechwytywać wyjątek

Tabela 1. Błędy wywodzące się z klasy logic_error

Klasa wyjątku	Opis
logic_error	Błąd logiczny. Klasa bazowa dla pozostałych błędów logicznych wywodząca się z exception.
domain_error	Błąd logiczny. Liczba poza dziedziną.
invalid_argument	Błąd logiczny. Błędny argument przekazany do funkcji.
length_error	Błąd logiczny. Błąd rozmiaru. Np przy próbie zmiany rozmiaru wektora.
out_of range	Błąd logiczny. Wartość poza zakresem.

Tabela 1. Błędy wywodzące się z klasy runtime_error

Klasa wyjątku	Opis
runtime_error	Błąd wykonania. Klasa bazowa dla pozostałych błędów wykonania wywodząca się z exception.
range_error	Błąd wykonania. Błąd zakresu.
overflow_error	Błąd wykonania. Błąd przepełnienia.
underflow_error	Błąd wykonania. Błąd niedomiaru.

Tabela 3. Zajętość pamięci Flash dla różnych wariantów programu

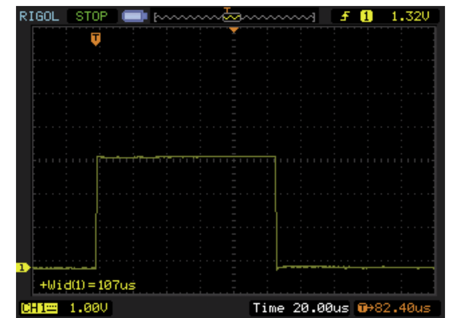
Opis	Wielkość FLASH
Kompletny przykład z wyjątkami std::logic_error oraz int	~35kB
Przykład z obsługą wyłącznie wyjątków POD typu int	~19KB
Przykład bez obsługi wyjątków	~7KB

Listing 4. Przykładowy program dla ISIX RTOS

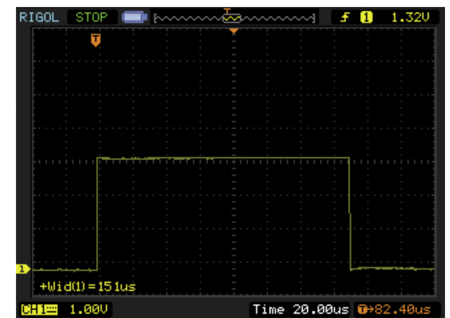
```
class ledkey: public isix::task_base
{
public:
  //Constructor
  ledkey()
    :task_base(STACK_SIZE,TASK_PRIO), is_enabled(false),
      KEY_PORT(GPIOE), LED_PORT(GPIOE)
      {
        using namespace stm32;
        //Enable PE in APB2
        RCC->APB2ENR |= RCC_APB2Periph_GPIOE;
        io_config(LED_PORT,LED_PIN,GPIO_MODE_10MHZ,GPIO_CNF_GPIO_PP);
        io_config(LED_PORT,NOTIFY_PIN,GPIO_MODE_10MHZ,GPIO_CNF_GPIO_PP);
      }
protected:
  //Main function
  virtual void main()
  {
    //Last key state
    bool state = true;
    //Task/thread main loop
    try
    {
      while(true)
      {
        execute_keycheck(state);
      }
    }
    catch( int &val)
    {
      stm32::io_clr( LED_PORT,NOTIFY_PIN );
      dbprintf(„INT exception [%d]", val);
    }
    catch( const std::exception &e )
    {
      stm32::io_clr( LED_PORT,NOTIFY_PIN );
      dbprintf(„std::exception [%s]", e.what());
    }
  }
private:
  //Execute keycheck function
  void execute_keycheck(bool &p_state)
  {
    //Change state on rising edge
    if(stm32::io_get(KEY_PORT, KEY_PIN) && !p_state)
    {
      is_enabled = !is_enabled;
    }
    //Get previous state
    p_state = stm32::io_get(KEY_PORT, KEY_PIN);
    //If enabled change state
    if(is_enabled) stm32::io_clr( LED_PORT, LED_PIN );
    else stm32::io_set( LED_PORT, LED_PIN );
    //Wait short time
    isix::isix_wait( isix::isix_ms2tick(DELAY_TIME) );
    if( !stm32::io_get(KEY_PORT, KEY_RAISE_LOGIC ))
    {
      /** From raise to catch 151us **/
      stm32::io_set( LED_PORT,NOTIFY_PIN );
      throw(std::logic_error(„critical error raised"));
    }
    if( !stm32::io_get(KEY_PORT, KEY_RAISE_INT ))
    {
      /** From raise to catch 108us **/
      stm32::io_set( LED_PORT,NOTIFY_PIN );
      throw(-1);
```

Listing 4. c.d.
```
 }
  }
private:
  static const unsigned STACK_SIZE = 2048;
  static const unsigned TASK_PRIO = 3;
  bool is_enabled;
  GPIO_TypeDef * const KEY_PORT;
  GPIO_TypeDef * const LED_PORT;
  static const unsigned LED_PIN = 15;
  static const unsigned KEY_PIN = 8;
  static const unsigned NOTIFY_PIN = 0;
  static const unsigned KEY_RAISE_LOGIC = 9;
  static const unsigned KEY_RAISE_INT = 10;
  static const unsigned DELAY_TIME = 25;
};
}

/* ------------------------------------------------------------------ */
//App main entry point
int main()
{
  dblog_init( stm32::usartsimple_putc, NULL, stm32::usartsimple_init,
  USART2,115200,true, config::PCLK1_HZ, config::PCLK2_HZ );
   dbprintf(„ Exceptions presentation app using ISIXRTOS „);
  //The blinker class
  static app::ledblink led_blinker;
  //The ledkey class
  static app::ledkey led_key;
  //Start the isix scheduler
  isix::isix_start_scheduler();
}
```



Rysunek 1. Czas przechwycenia wyjątku typu int



Rysunek 2. Czas przechwycenia wyjątku klasy logic_error

typu podstawowego int oraz wspomniany wcześniej wyjątek klasy std::logic_error(). W momencie zgłoszenia jednego z tych wyjątków przez metodę ledkey::execute_keycheck() wykonanie pętli nieskończonej zostanie przerwane poprzez przejście do sekcji catch jednego z wyjątków. Efektem tego będzie wysłanie przez interfejs szeregowy komunikatu o rodzaju wyjątku, a następnie przejście poza sekcję catch, co spowoduje zakończenie wykonania wątku. A zatem, program przestanie reagować na wciskanie klawisza OK. W cyklicznie wywoływanej metodzie ledkey::execute_keycheck() jest sprawdzany stan klawisza OK. W wypadku wykrycia wciśnięcia tego klawisza, stan diody zmieniany jest na przeciwny. Symulacja wystąpienia sytuacji wyjątkowych, jest realizowana po wciśnięciu klawisza UP lub DOWN. W przypadku wciśnięcia klawisza UP, zgłaszany jest wyjątek typu int o wartości -1. W przypadku wciśnięciu klawisza DOWN zgłaszany jest wyjątek klasy std::logic_error. Tuż przed zgłoszeniem wyjątków linia PE0 jest ustawiana. Wciśnięcie odpowiedniego klawisza i zgłoszenie wyjątku spowoduje przejście do odpowiedniej sekcji catch, w której jest zerowana linia PE0 oraz wypisywana albo wartość wyjątku int (–1), albo opis tekstowy wyjątku, który zwróci metoda what().

Przyjrzyjmy się teraz zasobom pamięci Flash zajmowanym przez mechanizm obsługi wyjątków. W tym celu opisany wcześniej przykład najpierw pozbawimy obsługi wyjątków realizowanych przez klasę (std::logic_error), a zostawimy zgłaszanie wyjątku typu prostego int. Kolejną czynnością będzie całkowite wyłączenie kodu obsługi wyjątków w tym również zmianę w pliku *Makefile* zmiennej CPP_EXCEPTIONS=n wyłączającej obsługę wyjątków w ISIX, a następnie sprawdzenie rozmiaru zajmowanej pamięci, co pozwoli mieć orientacyjny pogląd na to ile zajmuje dodatkowy kod obsługi wyjątków. Wyniki przedstawionych „badań" zamieszczono w **tabeli 3**.

Jak łatwo możemy zauważyć sam kod odpowiedzialny za podstawową obsługę wyjątków zajmuje około 12 kB pamięci Flash. W przypadku większych mikrokontrolerów z rodziny Connectivity Line czy Performance Line jest to wielkość pomijalna, ponieważ stanowi ona całkowity koszt użycia wyjątków niezależnie od tego, ile razy później będą użyte w programie. Dodatkowo, 16 kB będzie użyte w wypadku, gdy wykorzystamy standardowe wyjątki z hierarchii wyjątków biblioteki STD, co jest związane głównie z dołączeniem do kodu klasy std::string odpowiedzialnej za łańcuchy tekstowe. Jednak dla „większych" mikrokontrolerów jest to koszt całkowicie do zaakceptowania. Spójrzmy teraz na czas upływający od momentu zgłoszenia wyjątku do jego przechwycenia w klauzuli catch(...). Na **rysunku 1** oraz **rysunku 2** przedstawiono oscylogramy na linii PE0 reprezentujące czas przechwycenia wyjątku typu int oraz wyjątku klasy std::logic_error().

Czas jaki upłynął od momentu zgłoszenia wyjątku do jego przechwycenia, dla typu prostego int wynosi około 107 μs. Czas od momentu zgłoszenia wyjątku do jego przechwycenia dla typu std::logic_error() wynosi 151 μs przy taktowaniu mikrokontrolera częstotliwością 72 MHz. Zdaniem autora uzyskane czasy są całkiem zadowalające, ponieważ jak wspomnieliśmy, mechanizm wyjątków powinien być używany tylko do zgłaszania sytuacji wyjątkowych jak na przykład błędy. Należy także pamiętać o tym, że wykonanie programu, gdy przebiega on zgodnie ze ścieżką podstawową (bez wystąpienia wyjątku [błędu]), jest bardziej wydajne, niż w wypadku ręcznej obsługi wyjątków poprzez wartości zwracane jak w C, ponieważ nie ma konieczności, aby w każdej z warstw sprawdzać rezultat błędu. Warto tutaj wspomnieć jeszcze o zasobach pamięci RAM potrzebnych do obsługi wyjątków. Praktyczne próby pokazały że minimalna wartość stosu dla wątku, jeżeli chcemy wykorzystywać w nim wyjątki powinna wynosić około 1 kB.

Zaprezentowany przykład obsługi wyjątków dostępny jest wraz z pozostałymi przykładami dla systemu ISIX-RTOS na stronie *http://bryndza.boff.pl/index.php?dz=rozne&id=isixrtos*, a najświeższe przykłady można pobrać z repozytorium mercurial za pomocą polecenia: *hg clone http://ww.boff.pl/hg/isix/isix_samples*.

Zakończenie

Współczesne mikrokontrolery mają coraz więcej zasobów pamięci, dzięki czemu zaawansowane mechanizmy C++ stają w zasięgu programistów. Zaprezentowane przykłady pokazały, że dodatkowy zasoby potrzebne na wykorzystanie tych mechanizmów są w pełni akceptowalne, a dzięki wykorzystaniu mechanizmu sytuacji wyjątkowych, zyskujemy na niezawodności programu, głownia dlatego, że zgłoszonego wyjątku nie można po prostu zignorować, jak ma to miejsce dla mechanizmu zwracania kodów błędów poprzez wartość zwracaną z funkcji.

Lucjan Bryndza, EP