Programowanie systemów Internetu Rzeczy i aplikacji sieciowych

Programowanie mikrokontrolerów z użyciem RiotOS

Wprowadzenie

Tematem zajęć jest tworzenie oprogramowania uruchamianego na mikrokontrolerze STM. Celem jest zapoznanie studentów z procesem programowania mikrokontrolerów z użyciem systemu operacyjnego czasu rzeczywistego: RiotOS.

RiotOS jest oprogramowaniem rozbudowanym, o niskim poziomie wejścia, lecz wysokim poziomie trudności zrozumienia działania całości. Wśród systemów operacyjnych jest systemem o bardzo niskim wykorzystaniu zasobów, najważniejsze cechy można zrozumieć czytając pliki w katalogu *core*.

Studenci w trakcie trwania zajęć będą poinformowani o tym z jakich funkcji najlepiej korzystać do osiągnięcia założonego celu, natomiast warto, żeby użycie poszczególnych funkcji sami próbowali zrozumieć poprzez czytanie zawartych w każdym pliku nagłówkowym komentarzy.

1. Przygotowanie do pracy

a. Uruchomienie maszyny wirtualnej.

Należy uruchomić program VirtualBox, a następnie wejść w ustawienia maszyny o nazwie ***riot_programming.*** Jeśli nie ma takiej maszyny należy za pomocą przycisku *Plik/Importuj urządzenie wirtualne* ją dodać z obrazu znajdującego się na dysku komputera. W zakładce USB, do listy przechwytywanych urządzeń musi być dodane wejście o nazwie: *STMicroelectronics STM32 STLink*. Jeśli lista jest pusta, lub nie ma na niej tego urządzenia, należy podpiąć płytkę używaną podczas laboratorium do dowolnego portu USB komputera, a następnie kliknąć przycisk *Dodaj nowy filtr* w tej samej zakładce po prawej stronie, następnie z listy wybrać *STMicroelectronics STM32 STLink.* W ustawieniach maszyny w zakładce *System* można także zmienić przydzielony RAM na 4GB lub więcej, jeśli komputer dysponuje więcej niż 8GB pamięci RAM. W następnym kroku można uruchomić maszynę wirtualną.

b. Uruchomienie środowiska pracy.

Po włączeniu maszyny wirtualnej należy zalogować się na konto **student** za pomocą hasła: **hehe**. Pracę z nowym projektem RIOT OS powinno się zacząć od pobrania Eclipse IDE for C/C++ Developers oraz sklonowania z repozytorium GIT najnowszej wersji RIOT:

```
git clone https://github.com/RIOT-OS/RIOT.git
```

Należy również sprawdzić czy wszystkie wymagane dodatkowe narzędzia są zainstalowane i aktualne:

```
sudo apt install git gcc-arm-none-eabi make gcc-multilib openocd gdb-multiarch doxygen wget unzip python3-serial
```

Warto również sprawdzić czy zainstalowane są aktualizacje:

```
sudo apt update
```

W celu przygotowania środowiska do cross kompilacji należy również sprawdzić czy zainstalowana jest wtyczka Eclipse Embedded C/C++ (**Help→Eclipse Marketplace**).

Powyżej opisane czynności zostały już przeprowadzone na przygotowanej maszynie wirtualnej. Jednak należy je wykonać w przypadku tworzenia projektu do podstaw lub importowania projektu na nowym systemie.

Eclipse znajduje się w lokalizacji:

/home/ecplise

Natomiast najnowsza wersja systemu RIOT OS w folderze:

/home//RIOT

Tę lokalizację będziemy oznaczać *RIOT_BASE*. W katalogu *RIOT_BASE/PBL5/lab* znajdują się pliki (nowy projekt), które będą używane podczas laboratorium. Plik *main.c* zawiera program wykorzystywany w danej części laboratorium oraz *Makefile* który definiuje zmienne preprocesora wymagane do kompilacji danej aplikacji. Podkatalog z plikiem *Makefile* standardowo definiuje pojedynczą aplikację. Przygotowany projekt został opracowany na podstawie domyślnego projektu jaki znajduje się w lokalizacji:

RIOT_BASE/examples/default

W przypadku RIOT każdy projekt ma własny plik *Makefile* którego struktura jest następująca:

```
# name of the application
APPLICATION = lab

# If no BOARD is found in the environment, use this default:
BOARD ?= b-l072z-lrwan1

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../..

# Comment this out to disable code in RIOT that does safety checking
# which is not needed in a production environment but helps in the
# development process:
DEVELHELP ?= 0

# Change this to 0 show compiler invocation lines by default:
QUIET ?= 1

USEMODULE += xtimer
FEATURES_REQUIRED += periph_gpio_irq
```

include $(RIOTBASE)/Makefile.include

APPLICATION – określa nazwę aplikacji. Taką też nazwę będzie miał plik wynikowy (bin oraz elf)
RIOTBASE ?= $(CURDIR)/../.. – określa lokalizację folderu *RIOT_BASE*. Należy zwrócić uwagę na ilość podfolderów prowadzących do projektu kompilowanego.
BOARD ?= native - określa domyślną płytę na jaką będzie kompilowany projekt.
DEVELHELP ?= 1 – włącza funkcję DEBUG (wysyłania na standardowy port szeregowy znaków wskazanych przez użytkownika).

W systemie RIOT różne funkcjonalności dodawane są za pomocą tzw. modułów poprzez dodane w pliku *Makefile* linii `USEMODULE += xyz`. Użycie tych modułów jest potem „widziane" w projekcje jako definicje preprocesora.
W pliku tym można również dodawać flagi jakie będą „widziane" w projekcje jako definicje preprocesora np.:

```
CFLAGS += -DCONFIG_GNRC_PKTBUF_SIZE=512
```

c. Importowanie projektu do Eclipse IDE.

Po włączeniu Eclipse IDE należy stworzyć nowy projekt poprzez wybranie kolejno:

File→New→Makefile Project with Existing Code

Następnie w linii **Existing Code Location** należy wskazać lokalizację folderu z systemem RIOT

/home/RIOT (*RIOT_BASE*) i kliknąć OK.

Następnym krokiem jest przygotowanie zmiennych. Po wybraniu projektu (musi on być zaznaczony na liście **Project Explorer**) klikamy kolejno **Project→Properties**. Następnie rozwijamy **C/C++ Build**, wybieramy **Build Variables,** klikamy **Add…** i tworzymy zmienną o nazwie **`RIOT_PROJECT`** gdzie wskazujemy projekt jaki będziemy kompilować (w naszym wypadku będzie to PBL5/lab). Zatwierdzamy klikając **Apply and Close**.

Kolejnym krokiem jest konfigurowanie ARM cross toolchain. Ponownie klikamy **Project→Properties** i wybieramy **Tool Chain Editor**. Odznaczmy **Display compatible toolchains only** i z rozwijanego menu wybieramy **Cross ARM GCC**.

Następnie klikamy **Settings**. W celu wskazania ścieżki do gcc-arm-none-eabi należy kliknąć **Global path** i wskazać folder *bin*. W używanej maszynie toolchain znajduje się w lokalizacji:
/usr/local/gcc_arm/gcc-arm-none-eabi/bin
Wszystko zatwierdzamy klikając **Apply and Close.**

Po wykonaniu tych czynności należy przygotować środowisko. Ponownie wchodzimy do menu **Project→Properties**. Tym razem wybieramy **C/C++ Build** (bez rozwijania) i odznaczmy **Use default build command** W polu **Build Location** dodajemy **`${RIOT_PROJECT}`**. Tak przygotowana linia powinna wyglądać następująco:
`${workspace_loc:/RIOT}/${RIOT_PROJECT}`
Gdzie **`RIOT_PROJECT`** jest utworzoną wcześniej zmienną ze wskazaniem na kompilowany projekt.

Wszystko zatwierdzamy klikając **Apply and Close.**

Kolejnym korkiem jest wygenerowanie ścieżek i symboli dla kompilowanego projektu. Aby tego dokonać otwieramy lokalizację projektu:
/home/student/RIOT/PBL5/lab
W folderze tym uruchamiamy terminal (prawy przycisk myszy -> Open in Terminal) i tworzymy plik xml z konfiguracją projektu wywołując komendę

```
make BOARD= b-l072z-lrwan1 eclipsesym
```

BOARD powinna wskazywać na płytkę jaka będzie użyta w projekcie. W przypadku tych zajęć jest to płyta z kontrolerem STM32L072 o oznaczeniu b-l072z-lrwan1.
Jeżeli jest to nowy projekt wywołanie poprzedniej komendy zakończy się błędem. Dlatego należy wcześniej skompilować cały projekt RIOT wywołując komendę:

```
make all
```

Tak wygenerowany plik xml (*eclipsesym.xml*) należy zaimportować do projektu. W tym celu wracamy do Eclipse IDE i otwieramy menu **Project→Properties**. Następnie wybieramy **C/C++ General→Paths and Symbols** i klikamy **Restore Defaults** aby usunąć ewentualne wcześniejsze konfiguracje. Później klikamy na **Import Settings...** i wybieramy utworzony wcześniej plik **`eclipsesym.xml`** i klikamy **Finish**. Aby zakończy import ustawień przebudowujemy indeksy wybierając **Project→C/C++ Index→Rebuild**.

Tak przygotowany projekt można spróbować skompilować klikając ikonę Build.

d. Podstawowe komendy w pracy z RiotOS

Nawet bez Eclipe projekt można zbudować. W tym celu należy użyć komendy make. Musi ona zostać użyta w podkatalogu zawierającym aplikację. Plik *Makefile* może definiować domyślną płytkę, dla której aplikacja będzie budowana, natomiast my będziemy robić to z poziomu wywołania komendy make:

```
make BOARD=b-l072z-lrwan1
```

(jeśli pokaże się błąd, że system nie może znaleźć kompilatora gcc-arm-none-eabi, należy go dodać do ścieżki za pomocą polecenia: *export PATH="/usr/local/gcc_arm/gcc-arm-none-eabi/bin:$PATH"*)

Żeby zbudować program a następnie od razu wgrać go na płytkę, można użyć polecenia make flash:

```
make flash BOARD=b-l072z-lrwan1
```

W ten sposób, gdy program zostanie zbudowany, uruchamiany jest programator (dla płytki b-l072z-lrwan1 jest to OpenOCD), który odczytuje pliki konfiguracyjne dla danego celu, a następnie programuje mikrokontroler, za pośrednictwem wbudowanego w płytkę programatora (na płytkach z kontrolerami STM jest to ST-link).

e. Odczytywanie informacji ze standardowego wyjścia płytki

W celu odczytywania informacji ze standardowego wyjścia płytki można uruchomić program Putty (jeśli nie jest zainstalowany w systemie to zainstalować komendą: *sudo apt-get install putty*), jako wejście zaznaczyć *Serial*, następnie ustawić *Serial Line*: /dev/ttyACM0, *Speed*: 115200. Przed uruchomieniem można także w zakładce *Terminal* włączyć opcje *Implicit CR in every LF*, pomoże to w czytelnym formatowaniu tekstu w terminalu.

f. Programowanie płytki

Jest kilka dróg programowania. Jedną z najprostszych jest wykorzystanie programatora STLink wbudowanego w płytkę i dedykowanego oprogramowania dla Linux st-link. Jednak w tym wypadku jest to niemożliwe, gdyż ze względu na błąd oprogramowanie to nie wspiera kontrolerów STM32L0 nowej generacji.

Dlatego też na zajęciach wykorzystany zostanie sposób polegający na kopiowaniu pliku bin jaki zostaje utworzony w lokalizacji:

RIOT_BASE/RIOT_PROJECT/RIOT_BOARD

W naszym przypadku będzie to:

/home/RIOT/PBL5/lab/b-l072z-lrwan1

na dysk jaki jest utworzony po podłączeniu płytki do komputera.

2. Programowanie mikrokontrolera

Zajęcia są podzielone na części, w których należy zaprogramować funkcjonalność opisaną w instrukcji. Całość aplikacji zawsze znajduje się w pliku main.c. Wszystkie potrzebne pliki nagłówkowe zostały już załączone do kodu. Podstawowa, skrócona dokumentacja części potrzebnych funkcji i struktur znajduje się na końcu instrukcji. Polecamy także czytanie dokumentacji w plikach nagłówkach. Na początku pliku może znajdować się przydługi opis co może się tam znajdować i jak będzie działać, natomiast można od razu wyszukać interesującą nas funkcję, jej parametry wejściowe i wyjściowe będą opisane.

a. Tworzenie wątku

Celem pierwszej części jest utworzenie wątku i uruchomienie w nim funkcjonalności polegającej na miganiu zieloną diodą na płytce.

1. Otwieramy plik *main.c* w katalogu *RIOT_BASE/PBL5/lab*.
2. Tworzymy wątek za pomocą funkcji *thread_create*, przy wywołaniu należy podać m.in przestrzeń w pamięci na stos, wielkość stosu, priorytet wątku, dodatkowe flagi tworzenia wątku, wskaźnik do kodu wykonywanego w tworzonym wątku oraz można dodać opis wątku. Przestrzeń pamięci jest już dodana w kodzie pod nazwą: *stack_thread_blinking_green*. Czym

mniejszy jest parametr opisujący priorytet tym faktyczny priorytet wątku jest większy, w naszym przypadku może być ustawiony na *THREAD_PRIORITY_MAIN – 1*. Flagę możemy ustawić na *THREAD_CREATE_STACKTEST*. Jako wykonywany kod ustawiamy wskaźnik na utworzoną już funkcję *thread_blinking_green*.

3. W funkcji *thread_blinking_green* należy napisać funkcjonalność polegającą na miganiu zieloną diodą:

   w nieskończonej pętli używamy makro migającego diodą (*GREEN_LED_TOGGLE*), oraz używamy funkcji *xtimer_usleep*, która wybudzi wątek po określonym czasie.

Oczekiwanym rezultatem jest miganie zielonej diody na płycie mikrokontrolera.

PYTANIE/ZADANIE:

W funkcji *thread_blinking_green* należy zmodyfikować funkcjonalność migania zieloną diodą:

a. tworzymy zmienną o typie *xtimer_ticks32_t*, która będzie odniesieniem w stosunku do którego będziemy budzić działanie funkcji,
b. przypisujemy do zmiennej teraźniejszą wartość czasu (*xtimer_now()*),
c. w nieskończonej pętli używamy makro migającego diodą (GREEN_LED_TOGGLE), oraz używamy funkcji xtimer_periodic_wakeup, która wybudzi wątek po określonym czasie

Jak wpływ na działanie programu będzie miała zamiana funkcji *xtimer_usleep* na *xtimer_periodic_wakeup*?

b. Obsługa przerwania

Celem drugiej części jest napisanie kodu obsługującego przerwanie wywoływane przez naciśnięcie przycisku na płytce mikrokontrolera.

1. W funkcji main.c należy użyć funkcji *gpio_init_int* to zainicjalizowania przerwania na standardowym wejściu. W parametrach wejściowych funkcji należy podać oznaczenia przycisku który będzie używany (makro jest stworzone w pliku main.c), tryb pracy, zbocze pracy (kolejny podpunkt będzie wymagał aktywnego zbocza zarówno narastającego jak i opadającego), funkcję wywoływaną w trakcie przerwania oraz można przesłać do funkcji argument, na ten moment można ustawić na NULL.
2. Napisać w funkcji *user_button_callback* funkcjonalność, która liczyłaby w prosty sposób czas wciśnięcia przycisku. Uzyskiwanie aktualnego czasu za pomocą funkcji *xtimer_now(),* obliczanie różnicy pomiędzy momentami czasu za pomocą funkcji *timer_diff().* Wyświetlanie wyniku w konsoli można realizować za pomocą funkcji *DEBUG()* lub *printf()*.

Oczekiwanym rezultatem jest wyświetlanie w konsoli czasu przez który przycisk był przyciśnięty.

PYTANIE/ZADANIE:

Sprawdzić działanie wywłaszczania w RIOT i porównać z FreeRTOS.

c. Wspólne zasoby

Celem trzeciej części laboratorium jest taka modyfikacja programu, żeby to wątek obsługujący migającą diodę drukował czas przyciśnięcia przycisku. Realizowane to będzie poprzez dostęp do wspólnych zasobów.

1. Zadeklaruj zmienną globalną, która będzie przechowywała czas przyciśnięcia przycisku.
2. Zmień działanie funkcji *user_button_callback()* w taki sposób, żeby czas przyciśnięcia przycisku był zapisywany do zmiennej globalnej. Optymalny sposób byłby taki, żeby adres zmiennej był podawany jako argument funkcji *user_button_callback()* przy inicjalizacji przerwania.
3. Zmodyfikuj działanie wątku *thread_blinking_green()* w taki sposób, żeby czas przyciśnięcia przycisku był odczytywany ze zmiennej globalnej, a następnie drukowany w konsoli. Optymalny sposób byłby taki, żeby adres zmiennej był podawany jako argument przy inicjalizacji wątku.

Oczekiwanym rezultatem jest drukowanie w konsoli czasu przyciśnięcia przycisku, podczas gdy zielona dioda dalej będzie migać.

PYTANIE/ZADANIE:

Czy jest to optymalne rozwiązanie? Jak wpływa na działanie aplikacji i zużycie energii

d. Przesyłanie wiadomości między wątkami

Celem tej części laboratorium uregulowanie dostępu do wspólnego obszaru pamięci.

1. Zmień utworzoną zamienną globalna na tablice 10-cio elementową.
2. Zmodyfikuj wątek w taki sposób, aby obliczał on i wyświetlał wartość średnią czasu z ostatnich 10 przyciśnięć.
3. Zmodyfikuj *user_button_callback()* tak aby zapisywał wartość zmierzonego czasu do odpowiedniej komórki tablicy.
4. Aby zapobiec przed równoczesnym dostępem do wspólnego obszaru pamięci (busfault) użyj funkcji *irq_disable* oraz *irq_restore*.

Oczekiwanym rezultatem jest wyświetlenie średniego czasu przytrzymania przycisku w konsoli bez kolizji (busfault).

e. Przesyłanie wiadomości między wątkami

Celem tej części zajęć jest stworzenie systemu w którym czas przyciśnięcia przycisku będzie wyświetlany przez wątek do którego wartość tego czasu zostanie przesłana poprzez system wiadomości systemu RIOT.

1. Usuń z wątku obsługującego zieloną diodę funkcjonalności związane z drukowaniem czasu przytrzymywania przycisku.
2. Stwórz drugi wątek w podobny sposób co w pierwszym podpunkcie zajęć. Zmień przestrzeń pamięci, priorytet oraz funkcję obsługującą wątek. Otrzymany PID wątku zapisz do już

utworzonej zmiennej globalnej – będzie on potrzebny do przesyłania wiadomości między wątkami.
3. Zmodyfikuj inicjalizację przerwania po wciśnięciu przycisku, teraz jako argument do funkcji powinien być przekazany wskaźnik na zmienną globalną przetrzymującą PID wątku czerwonego.
4. Zmodyfikuj *user_button_callback()* w taki sposób, żeby zamiast wpisywania czasu przytrzymywania przycisku do zmiennej, przesyłała go do wątku *thread_red* (będzie potrzebny PID tego wątku) za pomocą funkcji msg_send().
5. Napisz kod odbierający wiadomości w wątku *thread_red*, za pomocą funkcji *msg_receive*. Następnie wyświetl jego zawartość.

Oczekiwanym rezultatem jest wyświetlenie czasu przytrzymania przycisku w konsoli.

f. MUTEXy

Celem tej części zajęć jest stworzenie systemu, w którym dwa wątki będą korzystały ze wspólnego zasobu jakim będzie przetwornik ADC, a do jednego z wejść przetwornika będzie podłączony potencjometr. Aby zapobiec kolizjom w dostępie do tego zasobu należy użyć mutexu.

5. Usuń funkcje związane z obsługą przerwania.
6. Oba stworzone wątki przystosować do mrugania diodami (zieloną oraz czerwoną) ale z różnymi częstotliwościami.
7. Stwórz zmienną globalną *mutex_t*.
8. Inicjalizuj przetwornik ADC oraz jedno jego wejście (sprawdź na schemacie jakie wejścia są dostępne).
9. Odczytuj wartość z przetwornika ADC w obydwu wątkach i wyświetlaj odczytaną wartość (w każdym wątku oddzielnie).
10. W współdzielenie dostępu do ADC ureguluj przy pomocy funkcji *mutex_lock*.

Oczekiwanym rezultatem jest wyświetlenie aktualnego pomiaru z przetwornika ADC w dwóch wątkach bez zakłócania pracy mikrokontrolera.

ZADANIE

Zaimplementować użycie funkcji nieblokujących.

ZADANIE KOŃCOWE

Mamy układ monitorowania pracy silnika. Należy do niego stworzyć oprogramowanie składające się z 4 wątków.

1. Mierzący wartość średnią prądu pobieranego z sieci (przekładnik 5A/V)
2. Mierzący wartość prędkości obrotowej wentylatora (przetwornik 500 obr/min/V)
3. Liczący wartość średnią prądu (z 1 sek.)

4. Sygnalizujący uszkodzenie – przekroczenie wartości prądu RMS powyżej 15A lub prędkości powyżej 1500 obr/min.

Częstotliwość próbkowania prądu to 1 ksps a szybkości obrotowej to 150 sps. Pomiary są wysyłane do odpowiedniego wątku za pomocą wiadomości. Wątek sygnalizujący ma rozpoznawać czego dotyczy wiadomość za pomocą ID wątku od którego dostał wiadomość. W momencie przekroczenia którejś z wartości zapalana jest dioda czerwona. Użytkownik może ją skasować za pomocą przycisku (obsługa w przerwana).

Dostęp do ADC przez wątki zorganizować za pomocą mutexów. Wartości obliczone są cyklicznie wypisywane na konsoli.

Skrócona dokumentacja potrzebnych funkcji i struktur:

```
/**
 * @brief Creates a new thread.
 *
 * For an in-depth discussion of thread priorities, behavior and and flags,
 * see @ref core_thread.
 *
 * @note Avoid assigning the same priority to two or more threads.
 * @note Creating threads from within an ISR is currently supported, however
it
 *       is considered to be a bad programming practice and we strongly
 *       discourage you from doing so.
 *
 * @param[out] stack    start address of the preallocated stack memory
 * @param[in] stacksize the size of the thread's stack in bytes
 * @param[in] priority  priority of the new thread, lower mean higher
priority
 * @param[in] flags     optional flags for the creation of the new thread
 * @param[in] task_func pointer to the code that is executed in the new
thread
 * @param[in] arg       the argument to the function
 * @param[in] name      a human readable descriptor for the thread
 *
 * @return              PID of newly created task on success
 * @return              -EINVAL, if @p priority is greater than or equal to
 *                      @ref SCHED_PRIO_LEVELS
 * @return               -EOVERFLOW, if there are too many threads running
already
 */
kernel_pid_t thread_create(char *stack,
                           int stacksize,
                           uint8_t priority,
                           int flags,
                           thread_task_func_t task_func,
                           void *arg,
                           const char *name);
```

```
/**
 * @brief will cause the calling thread to be suspended until the absolute
 * time (@p last_wakeup + @p period).
 *
 * When the function returns, @p last_wakeup is set to
 * (@p last_wakeup + @p period).
 *
 * This function can be used to create periodic wakeups.
 * @c last_wakeup should be set to xtimer_now() before first call of the
 * function.
 *
 * If the result of (@p last_wakeup + @p period) would be in the past, the
function
 * sets @p last_wakeup to @p last_wakeup + @p period and returns immediately.
 *
 * @param[in] last_wakeup   base time stamp for the wakeup
 * @param[in] period        time in microseconds that will be added to
last_wakeup
 */
static inline void xtimer_periodic_wakeup(xtimer_ticks32_t *last_wakeup,
uint32_t period);
```

```
/**
 * @brief xtimer timestamp (32 bit)
 *
 * @note This is a struct in order to make the xtimer API type strict
 */
typedef struct {
    uint32_t ticks32;       /**< Tick count */
} xtimer_ticks32_t;


/**
 * @brief get the current system time as 32bit time stamp value
 *
 * @note    Overflows 2**32 ticks, thus returns xtimer_now64() % 32,
 *          but is cheaper.
 *
 * @return  current time as 32bit time stamp
 */
static inline xtimer_ticks32_t xtimer_now(void);
```

```
/**
 * @brief Compute difference between two xtimer time stamps
 *
 * @param[in] a  left operand
 * @param[in] b  right operand
 *
 * @return @p a - @p b
 */
```

```
static    inline    xtimer_ticks32_t    xtimer_diff(xtimer_ticks32_t    a,
xtimer_ticks32_t b);
```

```
/**
 * @brief   Initialize a GPIO pin for external interrupt usage
 *
 * The registered callback function will be called in interrupt context
every
 * time the defined flank(s) are detected.
 *
 * The interrupt is activated automatically after the initialization.
 *
 * @note    You have to add the module `periph_gpio_irq` to your project to
 *          enable this function
 *
 * @param[in] pin       pin to initialize
 * @param[in] mode      mode of the pin, see @c gpio_mode_t
 * @param[in] flank     define the active flank(s)
 * @param[in] cb        callback that is called from interrupt context
 * @param[in] arg       optional argument passed to the callback
 *
 * @return              0 on success
 * @return              -1 on error
 */
int gpio_init_int(gpio_t pin, gpio_mode_t mode, gpio_flank_t flank,
                  gpio_cb_t cb, void *arg);
```

```
/**
 * @brief   Available pin modes
 *
 * Generally, a pin can be configured to be input or output. In output mode,
a
 * pin can further be put into push-pull or open drain configuration. Though
 * this is supported by most platforms, this is not always the case, so
driver
 * implementations may return an error code if a mode is not supported.
 */
#ifndef HAVE_GPIO_MODE_T
typedef enum {
    GPIO_IN ,               /**< configure as input without pull resistor
*/
    GPIO_IN_PD,            /**< configure as input with pull-down resistor
*/
    GPIO_IN_PU,             /**< configure as input with pull-up resistor
*/
    GPIO_OUT,              /**< configure as output in push-pull mode */
    GPIO_OD,                 /**< configure as output in open-drain mode
without
                              *   pull resistor */
    GPIO_OD_PU             /**< configure as output in open-drain mode with
                              *   pull resistor enabled */
```

```
} gpio_mode_t;
#endif
```

```
/**
 * @brief   Override the GPIO flanks
 *
 * This device has an additional mode in which the interrupt is triggered
 * when the pin is low.
 *
 * Enumeration order is important, do not modify.
 * @{
 */
#define HAVE_GPIO_FLANK_T
typedef enum {
    GPIO_LOW,           /**< emit interrupt when pin low */
    GPIO_BOTH,          /**< emit interrupt on both flanks */
    GPIO_FALLING,       /**< emit interrupt on falling flank */
    GPIO_RISING,        /**< emit interrupt on rising flank */
} gpio_flank_t;
/** @} */
#endif /* ndef DOXYGEN */
```

```
/**
 * @brief Describes a message object which can be sent between threads.
 *
 * User can set type and one of content.ptr and content.value. (content is
a union)
 * The meaning of type and the content fields is totally up to the user,
 * the corresponding fields are never read by the kernel.
 *
 */
typedef struct {
    kernel_pid_t sender_pid;    /**< PID of sending thread. Will be filled
in
                                     by msg_send. */
    uint16_t type;              /**< Type field. */
    union {
        void *ptr;              /**< Pointer content field. */
        uint32_t value;         /**< Value content field. */
    } content;                  /**< Content of the message. */
} msg_t;
```

```
/**
 * @brief Send a message (blocking).
 *
 * This function sends a message to another thread. The ``msg_t`` structure
has
 * to be allocated (e.g. on the stack) before calling the function and can
be
 * freed afterwards. If called from an interrupt, this function will never
 * block.
 *
```

```
 * @param[in] m              Pointer to preallocated ``msg_t`` structure,
must
 *                           not be NULL.
 * @param[in] target_pid    PID of target thread
 *
 * @return 1, if sending was successful (message delivered directly or to a
 *            queue)
 * @return 0, if called from ISR and receiver cannot receive the message now
 *            (it is not waiting or it's message queue is full)
 * @return -1, on error (invalid PID)
 */
int msg_send(msg_t *m, kernel_pid_t target_pid);
```

```
/**
 * @brief Receive a message.
 *
 * This function blocks until a message was received.
 *
 * @param[out] m    Pointer to preallocated ``msg_t`` structure, must not
be
 *                  NULL.
 *
 * @return  1, Function always succeeds or blocks forever.
 */
int msg_receive(msg_t *m);
```