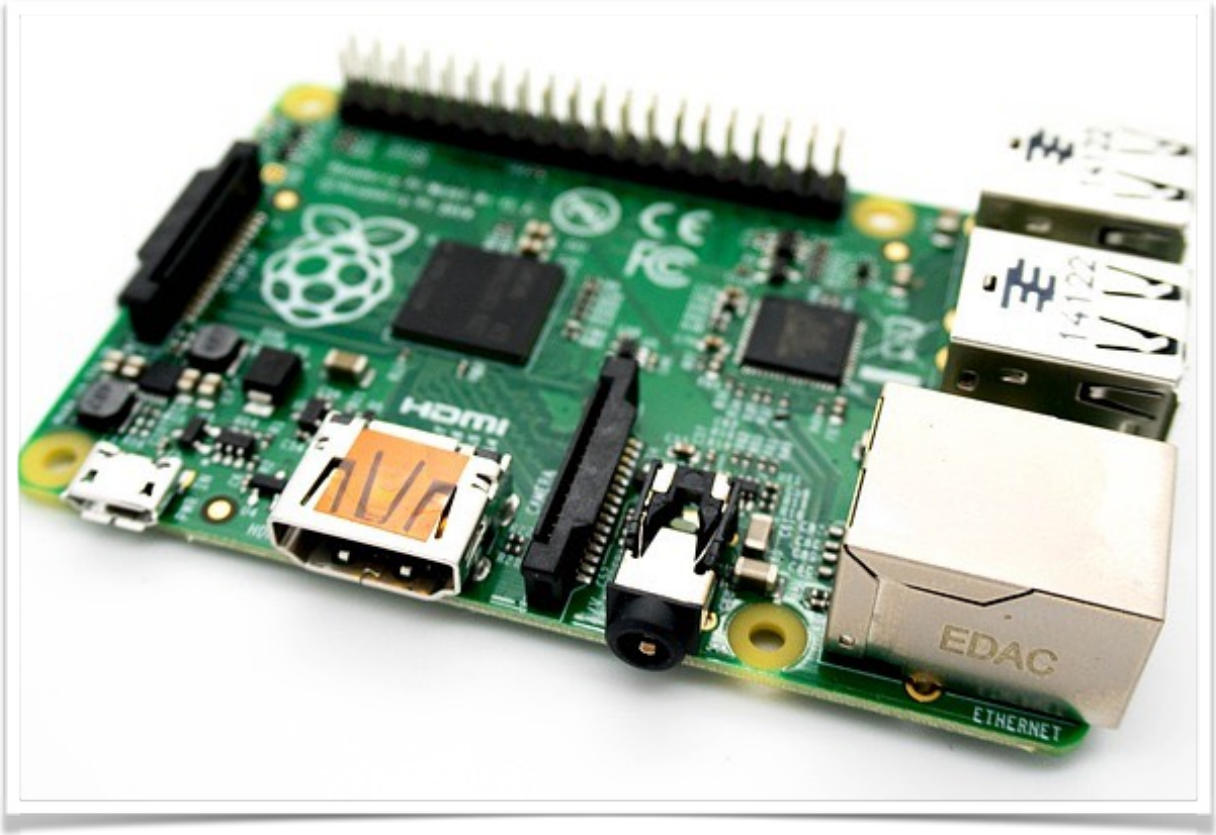
Linux w Systemach Wbudowanych



Laboratorium 2

Mateusz Sadowski

Lato 2018

1. Opis ćwiczenia

Główną częścią ćwiczenia było przygotowanie **aplikacji w języku C**, obsługującej przyciski i diody LED. Odpowiednio połączony układ 4 diod LED oraz 3 przycisków został dostarczony na laboratoriach.

Ważną częścią zadania było, aby zapewnić **brak aktywnego czekania**. Następnie aplikację należało odpowiednio skompilować na system docelowy i wgrać go na niego.

Pozostałe części zadania miały związek z napisaną aplikacją. Należało przekształcić ją w **pakiet Buildroota** oraz przetestować korzystanie z **debuggera gdb** łącząc się z maszyny źródłowej z maszyną docelową.

2. Rozwiązanie

2.1. Opis działania aplikacji

Przygotowałem program, który symuluje działanie świateł sygnalizacji w samochodzie. Umożliwia on obsługę kierunkowskazów w obie strony oraz świateł awaryjnych.

Sygnalizacja kierunku odbywa się przez sekwencyjne zapalanie jednej diody w jednym kierunku co SIGNAL_DELAY milisekund, aż do momentu zapalenia wszystkich diod. Po czym następuje zgaszenie wszystkich diod, opóźnienie o SIGNAL_DELAY milisekund i rozpoczęcie cyklu od nowa.

Dzieje się tak do momentu ponownego wciśnięcia przycisku wywołującego sygnalizację kierunku, co powoduje zgaszenie wszystkich diod i oczekiwanie na ponowne wprowadzenie danych, lub wciśnięcie przycisku wywołującego sygnalizację innego kierunku lub świateł awaryjnych, co powoduje bezzwłoczną sygnalizację nowego wyboru.

Sygnalizacja świateł awaryjnych odbywa się przez sekwencyjne zapalanie i gaszenie wszystkich diod co HAZARD_DELAY milisekund. Wciśnięcie przycisku podczas sygnalizacji świateł awaryjnych powoduje analogiczne zachowanie jak dla kierunkowskazów.

2.2. Implementacja

Do zaimplementowania wyżej opisanego programu wykorzystałem bibliotekę **wiringPi** pozwalającą na manipulację wtykami **GPIO** (General-Purpose Input/Output).

W pierwszym kroku napisałem skrypt *signal-lights.sh*, realizujący następujące funkcje:

1. export WIRINGPI_CODES=1 - powoduje, że funkcji biblioteczne wiringPi zwracają wartość ujemną w razie niepowodzenia, co pozwala na obsługę błędów

2. gpio export PIN_NUMBER out - przejęcie kontroli nad diodami LED i ustawienie ich jako wyjście
3. gpio export PIN_NUMBER in - przejęcie kontroli nad przyciskami i ustawienie ich jako wejście
4. gpio edge PIN_NUMBER both - ustawienie generowania sygnału zmiany stanu na przyciskach na zmianie stanu w obie strony
5. ./signal-lights - uruchomienie programu

Następnie napisałem program składający się z następujących kluczowych części:

1. Wywołanie funkcji wiringPiSetupSys()

   Funkcja ta przygotowuje środowisko pracy z wtykami GPIO w trybie Sys. Umożliwia to korzystanie z natywnej numeracji wtyków dla RaspberryPi, czyli tak jak na dostarczonej dokumentacji płytki LED oraz pozwala na uruchomienie programu zwykłemu użytkownikowi (nie wymaga uprawnień roota). Wymaga jednak wcześniejszego wyeksportowania odpowiednich wtyków, co zrobiłem w powyższym skrypcie.
2. Wywołania funkcji wiringPiISR(BUTTON_1, INT_EDGE_BOTH, &handler) dla każdego przycisku, gdzie BUTTON_1 to numer wtyku pierwszego przycisku

   Funkcja powoduje zarejestrowanie wywołania handlera na zdarzenie zmiany stanu przycisku pierwszego z *high* na *low* oraz z *low* na *high*.
3. Funkcje handler dla każdego przycisku

   Funkcje te wywołują debouncing dla odpowiedniego wtyku (opisane poniżej), a po ustabilizowaniu stanu odczytują go i ustawiają odpowiednie zmienne globalne sterujące logiką programu.
4. Funkcja deBounce(int pin)

   Realizuje **debouncing**, czyli opóźnienie odczytu stanu wtyku do momentu ustabilizowania się go. Jest to konieczne, ponieważ rzeczywiste urządzenia mają swoją bezwładność i styki bedą się od siebie odbijać przez pewien czas zanim złączą się i zamknął obwód.

Funkcja składa się z wywołania bibliotecznej funkcji waitForInterrupt(pin, DEBOUNCING_TIMEOUT) w pętli. Jeśli ta funkcja zostanie przerwana przez zmianę stanu na przycisku zostanie zwrócone 1. Jeśli funkcja skończy się przez timeout, zostanie zwrócone 0. Stąd funkcja będzie wywoływana w pętli do momentu zwrócenia wartości 0, czyli ustabilizowania stanu na przycisku.

5. Odczytanie stanu przycisku w handlerze

   Logika jest wykonywana tylko przy zmianie stanu z *high* na *low*. Stąd przy odczycie stanu *high* po ustabilizowania stanu, handler bezzwłocznie kończy wykonanie.
6. Główna pętla programu

   W funkcji main wykonywana jest pętla realizująca odpowiednią sygnalizację świetlną w zależności od zmiennych globalnych lub w razie braku sygnalizacji - śpi w oczekiwaniu na przerwanie zwracając zasoby do systemu.

6.1. Kompilacja i linkowanie

Aby odpowiednio zbudować program na maszynę docelową stworzyłem Makefile oraz skrypt budujący:

```
CC=$(CROSS_COMPILE)gcc
OBJS := signal-lights.o
signal-lights: $(OBJS)
    $(CC) -o signal-lights $(OBJS) -l wiringPi -lpthread -lrt
$(OBJS) : %.o : %.c
    $(CC) -c $(CFLAGS) $< -o $@
```

Makefile

```
BRPATH=/malina/sadowskim/buildroot-2018.02
(
export PATH=$BRPATH/output/host/usr/bin:$PATH
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- signal-lights
)
```

Skrypt budujący

Dzięki ustawieniu zmiennej środowiskowej BRPATH i określenia zmiennej CROSS_COMPILE kompilator sprawdzi ustawienia systemu w Buildroot oraz skompiluje odpowiednio na maszynę docelową.

6.2. Przekształcenie w pakiet Buildroota

Żeby dodać aplikację jako pakiet Buildroota trzeba w katalogu BRPATH/package/signal-lights stworzyć dwa pliki konfiguracyjne: Config.in oraz singal-lights.mk

W tych plikach należy określić zależności od bibliotek, skąd Buildroot ma pobrać źródła aplikacji oraz sposób pobierania.

Następnie aby nasz pakiet był widoczny w menu Buildroota trzeba dodać odpowiednią linię package/signal-lights/Config.in w pliku BRPATH/package/Config.in.

Po przebudowaniu systemu aplikacja powinna znajdować się w /usr/bin.

6.3. Korzystanie z debuggera

Aby móc debugować aplikację na maszynie docelowej ze stacji źródłowej, trzeba najpierw dodać odpowiednie ustawienia w systemie docelowym:

1. Target packages —> Debugging,... —> gdb —> gdbserver
2. Toolchain —> Build cross gdb for the host
3. Build options —> Build packages with debugging symbol

a następnie przebudować go.

Następnie należy skompilować aplikację z flagą -g3 umożliwiającą debuggowanie i na maszynie docelowej wykonać komendę: gdbserver host:7654 application.

Potem na maszynie źródłowej uruchamiamy:
BRPATH/output/host/usr/bin/arm-none-linux-gnueabi-gdb ścieżka-do-aplikacji/signal-lights
target remote xxx.yyy.zzz.vvv:7654, gdzie xxx.yyy.zzz.vvv - ip płytki

Pomoc na temat komend dostępnych w gdb możemy otrzymać:
gdb --help

Kilka podstawowych komendy to:
- break N, gdzie N to numer linii - aby ustawić breakpoint
- continue - aby kontynuować wykonanie programu
- step - aby przejść do kolejnej linii programu
- list - aby wyświetlić kod

3. Odtworzenie projektu

W paczce z plikami znajduje się plik źródłowy signal-lights-clean.c. Jest to plik z kodem aplikacji dodatkowo "posprzątanym" (na co nie było czasu na zajęciach). Z powodu, że nie mam możliwości przetestowania działania posprzątanego programu, w skrypcie uruchamiającym jest używana dla bezpieczeństwa wersja z laboratorium.

Proszę uruchomić skrypt build.sh z wnętrza katalogu, w którym się znajduje (uruchomić ./build.sh). W pliku signal-lights.mk należy ustawić zmienną SIGNAL-LIGHTS_SITE na ścieżkę do katalogu z plikami źródłowymi.

W razie problemów ze skryptem build.sh należy:
1. Skopiować ręcznie plik .config do katalogu głównego Buildroot
2. Ustawić ścieżkę do katalogu głównego Buildroot w zmiennej BRPATH w skrypcie make.sh
3. Wywołać skrypt make.sh
4. Przekopiować plik signal-lights oraz signal-lights.sh na system docelowy
5. Uruchomić skrypt signal-lights.sh