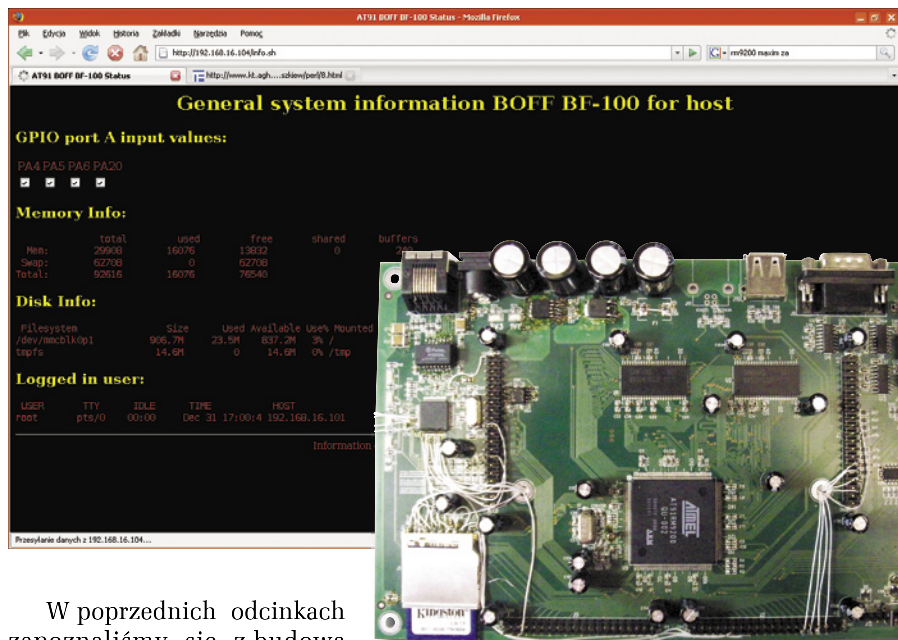
BF100 – linuksowy ARMputer, część 3

Przykładowa aplikacja

AVT–5128

Pomimo konstrukcyjnych niedociągnięć, ARMputer cieszy się dużym powodzeniem wśród naszych Czytelników. W drugiej (ostatniej) części artykułu przedstawiamy sposób jego montażu i uruchomienia, a także przygotowania instalacji Linuxa, która jest jednym z ważniejszych „smaczków" całego opracowania.

Rekomendacje:
opisany w artykule komputer jest dobrze wyposażoną platformą sprzętową, dla której przygotowaliśmy implementację systemu operacyjnego Linux. Jego wykorzystanie przenosi programistów w świat aplikacji innych, pisanych „wprost" na mikrokontrolery. Jeszcze kilka lat temu takie możliwości można było zakwalifikować do kategorii opowiadań science–fiction.



W poprzednich odcinkach zapoznaliśmy się z budową oraz procesem uruchamiania ARMputera. Jako zwieńczenie pierwszej części cyklu o „Linuxowym ARMputrze", pokażemy teraz krok po kroku, jak zmusić ten „piekielny ARMputer" do działania, na przykładzie prostego przykładu wizualizującego stan wybranych linii portów GPIO mikrokontrolera na stronie WWW. Do tego celu wykorzystamy zainstalowany w naszej dystrybucji serwer WWW – *lighthttpd*, skrypt w bashu do dynamicznego generowania strony WWW oraz prostą aplikację napisaną w języku C, która umożliwia odczytywanie i wyświetlanie stanu wybranej linii GPIO. Na **rys. 12** przedstawiono sposób współpracy poszczególnych komponentów projektu.

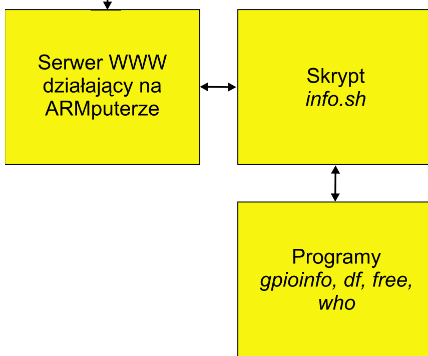
Użytkownik wpisując w przeglądarce internetowej swojego komputera adres *http://adres_armputera/info.sh* wysyła do serwera WWW działającego w ARMputerze żądanie odczytania strony *info.sh*. Serwer w tym momencie stwierdza, że nie jest to zwykły plik do przesłania, a skrypt CGI, więc go uruchamia. Skrypt ten następnie tworzy dynamicznie kod *html* zwracający informację o stanie portów linii GPIO, ilości wolnej pamięci RAM, ilości wolnego miejsca na dysku oraz informacji o zalogowanych użytkownikach i zwraca go do serwera, który następnie przesyła go do przeglądarki. W celu uzyskania informacji o stanie wybranych linii GPIO skrypt wywołuje program *gpioinfo*, który zwraca stan wybranej linii portu. Informacje systemowe o ilości wolnej pamięci RAM, pamięci na dysku oraz zalogowanych użytkowników uzyskuje natomiast za pomocą programów systemowych *free*, *df*, *who*.

Odczyt stanu wybranej linii portu (program *gpioinfo*)

Jak pamiętamy z poprzednich artykułów system linux pracujący na mikrokontrolerze AT91RM9200 wykorzystuje tryb ochrony pamięci, gdzie każdy proces użytkownika posiada własną wirtualną przestrzeń adresową. Próba dostępu przez proces do pamięci, która nie należy do niego powoduje natychmiastowe otrzymanie sygnału **SIGSEGV**, w efekcie czego aplikacja zostaje natychmiast zamknięta, a w terminalu pojawia się komu-

PODSTAWOWE PARAMETRY

- Mikroprocesor AT91RM9200 (ARM920T)
- 32 MB SDRAM
- 128 kB Data Flash SPI
- Interfejs Ethernet
- 2xRS232 (jeden z kompletem linii)
- Interfejs MMC/SD
- Interfejsy USB–device i USB–host
- Interfejs RS485
- Zasilanie 5 V/800 mA (w tym 500 mA rezerwy dla USB–host)



Rys. 12. Obieg danych w przykładowej aplikacji

nikat „Segmentation Fault" (błąd segmentacji pamięci). Aby odczytać stan linii GPIO wybranego portu musimy mieć dostęp do obszaru pamięci rejestrów SFR mikrokontrolera. Niestety tak się składa, że aplikacja działająca w przestrzeni użytkownika bezpośrednio takiego dostępu nie ma, a dostęp do tego obszaru pamięci jest możliwy jedynie za pośrednictwem jądra systemu linux. Aplikacja użytkownika, która chce uzyskać dostęp do urządzeń peryferyjnych mikrokontrolera, musi się z nimi komunikować za pomocą sterowników urządzeń, które są częścią jądra systemu linux. Komunikacja pomiędzy sterownikami a aplikacją użytkownika dla dużej części urządzeń blokowych i znakowych polega na otwarciu specjalnego pliku urządzenia, który znajduje się w katalogu /dev/. Polega to na wywołaniu systemowego *open()*, a następnie odczytywaniu i zapisywaniu danych do i z urządzenia za pomocą wywołań systemowych *read()* i *write()*. Są to te same wywołania systemowe, które są wykorzystywane do operacji na zwykłych plikach. W momencie, gdy aplikacja użytkownika wywoła którąś z wspomnianych funkcji systemowych, po stronie jądra wywoływana jest odpowiadająca mu funkcja sterownika, która ma dostęp do wszystkich zasobów sprzętowych mikrokontrolera i może w odpowiedni sposób przekazać dane do urządzenia peryferyjnego. Dzięki takiemu mechanizmowi mamy pewność, że poszczególne procesy nie będą sobie wchodzić w drogę, zyskujemy pewność, że obsługa urządzenia będzie wykonana prawidłowo, mamy również możliwość przydzielania praw dostępu do urządzenia dla poszczególnych użytkowników. Można sobie wyobrazić, co by było, gdyby na przykład błędnie działająca aplikacja źle obsłużyła kontroler przerwań. Zatem najlepszą drogą, aby uzyskać dostęp do portów GPIO, jest napisanie sterownika urządzenia, z którym aplikacja użytkownika będzie komunikować się z wykorzystaniem wywołań systemowych *read()*, *write()*. Przykładowo, aby odczytać stan linii portu GPIO_A aplikacja powinna otworzyć plik hipotetycznego urządzenia */dev/gpioA*, a następnie za pomocą funkcji *read()* odczytywać stan linii tego portu. Istnieje prostszy sposób na uzyskanie bezpośredniego dostępu do portów SFR, z wykorzystaniem specjalnego urządzenia */dev/mem*, które jest dostępne dla super–użytkownika (roota). Urządzenie */dev/mem*, działa w taki sposób, że pisząc i czytając dane z tego urządzenia mamy bezpośredni dostęp do fizycznej pamięci, a więc również do rejestrów SFR i właśnie ten mechanizm wykorzystamy w naszym projekcie. Niestety dostęp do rejestrów mikrokontrolera za pomocą tego mechanizmu posiada szereg wad, a jedyną zaletą tego rozwiązania jest jego prostota. Główne wady takiego podejścia to:

- dostęp do urządzenia */dev/mem* jest możliwy tylko przez użytkownika uprzywilejowanego (root), co wymaga wykonywania aplikacji odwołującej się do tego urządzenia na prawach roota,
- używając tego mechanmu musimy uważać, aby nie popełnić jakiegoś błędu, ponieważ może mieć on katastrofalne następstwa dla całego systemu,
- nie mamy możliwości obsługi przerwań,
- niezbyt duża wydajność, ponieważ system wymaga ciągłego przełączania z trybu jądra do trybu użytkownika i odwrotnie.

Aby uzyskać dostęp do pamięci fizycznej musimy najpierw otworzyć urządzenie */dev/mem* za pomocą wywołania systemowego *open()*, przesunąć się za pomocą wywołania systemowego *lseek* w odpowiednie miejsce w pamięci, a następnie pisać lub czytać zawartość tej pamięci za pomocą wywołań *read()*, oraz *write()*. Rozwiązanie to nie jest zbyt wygodnie, dużo lepiej było by mieć dostęp do pamięci fizycznej za pomocą zwykłego wskaźnika do obszaru pamięci. W linuksie istnieje specjalne wywołanie systemowe *mmap()*, które umożliwia zmapowanie pliku/urządzenia do wirtualnej przestrzeni adresowej procesu. Dzięki temu mechanizmowi możemy odwoływać się do pliku lub urządzenia za pomocą wskaźnika na obszar pamięci zwrócony przez *mmap()*. Właśnie ten mechanizm wykorzystano w bibliotece **at91gpio**, która umożliwia odczyt portów GPIO. Biblioteka ta składa się z następujących funkcji:

```
at91gpio_handle at91gpio_
open(unsigned long base)
{
 //Allocate memory for handle
 at91gpio_handle handle = (at91gpio_
 handle)calloc(sizeof(struct
 at91gpio_context),1);
 if(handle==NULL) return NULL;
 //Open file mem to map
 handle->file = open(„/dev/mem",O_
 RDWR|O_SYNC);
 if(handle->file<0)
{
        free(handle);
        return NULL;
    }
    //Map to process space and
    calculate address
    unsigned long map_addr = base &
    ~(getpagesize()-1);
    handle->mem = mmap(0,getpagesi
    ze(),PROT_READ|PROT_WRITE,MAP_
    SHARED,handle->file,map_addr);
    if(!handle->mem)
    {
     close(handle->file);
     free(handle);
     return NULL;
    }
 handle->gpio = (uint32_t*) handle-
 >mem + (base-map_addr)/4;
 return handle;
}
```

Funkcja ta służy do otwarcia dostępu do portów GPIO, jako parametr może ona przyjąć wartości: AT91PORT_A...AT91PORT_D, odpowiadające portom IO mikrokontrolera AT91RM9200, w wyniku wywołania funkcji w przypadku powodzenia zwraca uchwyt do portu, lub wartość NULL w przypadku wystąpienia błędu. Działanie tej funkcji polega na otwarciu urządzenia */dev/mem* za pomocą funkcji *open()*, do której przekazano dodatkowe flagi: O_RDWR mówiącej, że urządzenie zostało otwarte do zapisu/odczytu oraz O_SYNC nakazującej natychmiastowe wykonanie fizycznego zapisu

do pliku. Po otwarciu pliku dostajemy deskryptor pliku, który należy przekazać do funkcji *mmap*. Do funkcji *mmap()* jako pierwszy argument przekazujemy proponowany adres w pamięci wirtualnej procesu (u nas NULL), co spowoduje wybranie adresu automatycznie przez jądro. Drugi argument określa rozmiar obszaru mapowania w pamięci RAM, która musi być wielokrotnością fizycznej strony pamięci. W naszym przypadku obszar rejestrów SFR portu GPIO ma wielkość 512 bajtów, co jest wartością dużo mniejszą od rozmiaru strony, dlatego jako ten argument przekazujemy rozmiar jednej strony pamięci, który możemy odczytać za pomocą funkcji *getpagesize()*. Trzeci argument określa tryb dostępu do mapowanego obszaru. Jako że chcemy pisać i czytać z rejestrów SFR, ustawiamy flagi PROT_READ oraz PROT_WRITE. Do czwartego argumentu przekazujemy flagę MAP_SHARED, co oznacza, że ten obszar pamięci może być dzielony pomiędzy procesami. Jako kolejny argument przekazujemy deskryptor otwartego pliku (*/dev/mem*), a jako ostatni adres bazowy w pamięci fizycznej. Jednak jako adres bazowy nie podajemy tu pierwszego rejestru SFR portu GPIO, tylko adres początku strony fizycznej, tuż przed adresem fizycznym rejestrów GPIO. Funkcja *mmap()* w wyniku wywołania zwraca wskaźnik do pamięci wirtualnej, będącej odzwierciedleniem pamięci fizycznej. Aby uzyskać dostęp do rejestru bazowego portu, należy jeszcze przesunąć ten wskaźnik o offset jaki wynika z różnicy pomiędzy początkiem strony fizycznej a rejestrem bazowym GPIO.

```
void at91gpio_setup(at91gpio_handle handle,uint32_t enable, uint32_t dir)
{
 //Enable pin as GPIO
 handle->gpio[PIO_PER] = enable;

 //Disable output
 handle->gpio[PIO_ODR] = ~dir;

 //Enable output dir
 handle->gpio[PIO_OER] = dir;

 //Disable pullup resistor in output pins
 handle->gpio[PIO_PUDR] = dir;

 //Enable Pullup resistor
 handle->gpio[PIO_PUER] = ~dir;
}
```

Funkcja, ta służy do konfiguracji linii GPIO i umożliwia określenie kierunku oraz trybu pracy portu. Jako argument **handle** przyjmuje ona uchwyt, który powinien być zwrócony przez poprzednią funkcję. Argument **enable**, określa, które linie portu mają pracować jako porty GPIO, gdzie wybrana linia pracuje w trybie GPIO, gdy odpowiadający jej bit jest ustawiony w stan wysoki. Argument **dir** określa kierunek pracy portu. Jeżeli bit odpowiadający wybranej linii jest ustawiony, wówczas wybrana linia pracuje w kierunku wyjścia, jeżeli wyzerowany, w kierunku wejścia. Działanie tej funkcji jest bardzo proste, mianowicie odwołując się za pomocą wskaźnika do początku obszaru SFR wybranego portu GPIO, konfiguruje ona linię GPIO zgodnie ze specyfikacją, czyli: do rejestru PIO_PER wpisywana jest maska bitowa linii portu, które mają pracować w trybie GPIO; do rejestru PIO_ODR wpisywane są jedynki na bitach tych linii, które mają pracować jako wejściowe; do rejestru PIO_OER wpisywane są jedynki na bitach tych linii, które mają pracować jako wyjściowe; do rejestru PIO_PUDR wpisywane są jedynki na tych liniach które mają pracować jako wyjściowe, w efekcie czego zostają od nich odłączone rezystory podciągające; do rejestru PIO_PUER wpisywane są jedynki na bitach tych linii, które mają pracować jako wejściowe, w efekcie czego do tych linii zostają dołączone rezystory podciągające.

```
uint8_t at91gpio_set(at91gpio_handle handle,uint8_t bit,uint8_t value)
{
 if(value) handle->gpio[PIO_SODR] = 1 << bit;
 else handle->gpio[PIO_CODR] = 1 << bit;
}
```

Funkcja ta ustawia wybraną linię portu, która przekazana jest jako argument bit, w stan określony przez argument val. Jeżeli argument val przyjmuje wartość logiczną prawda, wówczas do rejestru PIO_SODR, wpisywana jest jedynka na bicie odpowiadającym za ten port, co spowoduje fizyczne ustawienie linii tego portu w stan wysoki. Jeżeli val przyjmuje wartość logiczną fałsz, wówczas do rejestru PIO_CODR wpisywana jest jedynka na bicie odpowiadającym za ten port, co spowoduje fizyczne ustawienie linii tego portu w stan niski.

```
uint8_t at91gpio_get(at91gpio_handle handle, uint8_t bit)
{
 return (handle->gpio[PIO_PDSR] & (1<<bit) )?1:0;
}
```

Funkcja ta służy do odczytania stanu linii portu o numerze przekazanym jako parametr bit, poprzez odczytanie stanu rejestru PIO_PDSR, odzwierciedlającego stan linii wejściowych, i zwróceniu wartości logicznej prawda lub fałsz w zależności od tego czy linia znajduje się w stanie niskim lub wysokim.

```
void at91gpio_close(at91gpio_handle handle)
{
 //Unmap memory from process space
 munmap(handle->mem,getpagesize());
 //Close file
 close(handle->file);
 //Free alocated memory
 free(handle);
}
```

Funkcja ta powinna być wywołana na koniec działania programu, zwalniając używane zasoby, poprzez wywołanie funkcji *munmap()*, zwalniającej mapowanie, funkcji *close()* zamykającej uchwyt pliku, oraz funkcji *free* zwalniającej pamięć dynamiczną używaną przez bibliotekę.

Opisaną przed chwilą bibliotekę wykorzystamy w programie **gpioinfo**, który będzie służył do odczytywania stanu wybranej linii GPIO. Aby odczytać stan wybranej linii GPIO musimy wywołać program **gpioinfo** z pięcioma parametrami, których znaczenie w kolejności jest następujące:

- 1: port GPIO, którego stan chcemy odczytać
- 2: maska określająca, które linie portu mają być włączone w postaci hexadecymalnej (na przykład 0xff)
- 3: numer bitu linii, której stan chcemy odczytać (od 0 do 31)
- 4: ciąg znaków zwracanych, gdy linia jest w stanie wysokim (dowolny tekst)
- 5: ciąg znaków zwracanych, gdy linia jest w stanie niskim (dowolny tekst)

Aby na przykład program zwrócił tekst „jedynka” w przypadku, gdy linia PA7 znajduje się w stanie wysokim, a tekst „zero”, gdy linia PA7 znajduje się w stanie niskim, program należy wywołać nastepująco: *gpioinfo A 0x80 7 jedynka zero*. W taki sposób właśnie program jest wywoływany przez skrypt *info.sh* Kod tego programu jest bardzo prosty i składa się dosłownie z kilkunastu linijek, gdzie najistotniejszy jest fragment odczytujący stan linii portu GPIO:

```
at91gpio_handle hwnd = at91gpio_ope-
```

```
n(port);


if(hwnd==NULL)
    {
      printf(„Error\n”);
      exit(-1);
    }

     //Setup gpio port as input
     at91gpio_setup(hwnd,mask,0);

uint8_t val = at91gpio_get(hwnd,ato-
i(argv[3]));

if(val) puts(argv[4]);
else puts(argv[5]);

//Zamknij port gpio
at91gpio_close(hwnd);
```

Najpierw tworzony jest uchwyt do biblioteki poprzez wywołanie funkcji *at91gpio_open* z argumentem port, który otrzymujemy w wyniku przetworzenia wiersza polecenia. Następnie wywoływana jest funkcja przełączająca wybrane linie w tryb wejściowy GPIO oraz odczytywany jest stan wybranej linii za pomocą funkcji *at91gpio_get()*. Na zakończenie w zależności od tego czy linia portu znajdowała się w stanie wysokim lub niskim, zwracany jest albo argument 3, albo 4 wywołania programu, a następnie zwalniane są zasoby biblioteki poprzez wywołanie funkcji *at91gpio_close()*.

Kilka słów o skrypcie

Skrypt *info.sh* jest odpowiedzialny za dynamiczne tworzenie kodu HTML. Jest to bardzo prosty skrypt powłoki systemowej składający się z ciągu wywołań poleceń echo wypisujących kod HTML. Najciekawszy jest fragment odpowiedzialny za wywołanie programu *gpioinfo*, generującego zaznaczone lub odznaczone „checkboxy” w zależności od stanu wybranej linii GPIO.

```
#wiersz 2
echo „<tr>”
# stan 1
echo ‚<td> <input type=”checkbox”
name=”nic0” value=”nic0”’
echo „$(./gpioinfo A 0x100070 4
‚checked=”checked”’ ‚ ‚) /></td>”
# stan 2
echo ‚<td> <input type=”checkbox”
name=”nic1” value=”nic1”’
echo „$(./gpioinfo A 0x100070 5
‚checked=”checked”’ ‚ ‚) /></td>”
# stan 3
echo ‚<td> <input type=”checkbox”
name=”nic2” value=”nic2”’
echo „$(./gpioinfo A 0x100070 6
‚checked=”checked”’ ‚ ‚) /></td>”
# stan 4
echo ‚<td> <input type=”checkbox”
name=”nic3” value=”nic3”’
echo „$(./gpioinfo A 0x100070 20
‚checked=”checked”’ ‚ ‚) /></td>”
echo „</table>”
echo „</form>”
```

Widzimy tutaj ciąg wywołań programu *gpioinfo* z argumentem na tak checked=checked, gdy stan portu jest wysoki oraz pustym ciągiem, gdy stan portu jest niski. W zależności od tego czy wybrana linia portu znajduje się w stanie wysokim czy niskim, wówczas generowany jest kod z zaznaczonym lub nie krzyżykiem przy odpowiednim „checkboxie”. W zależności od potrzeb możemy odczytywać stan innych linii portów GPIO, modyfikując argumenty programu *gpioinfo*.

Instalacja programu, kompilacja projektów

Program ten oraz wszystkie narzędzia niezbędne do wykonania tego projektu możemy pobrać ze strony *http://bryndza.ep.com.pl* Opiszemy teraz krok po kroku, jak zbudować cały projekt z plików źródłowych mając na komputerze do dyspozycji system linux. Na stronie dostępna jest również skompilowana wersja programu ***gpioinfo***, więc etap kompilacji programu *gpioinfo* możemy pominąć. Przed przystąpieniem do kompilacji musimy ściągnąć cały toolchain (kompilator+biblioteki), a następnie rozpakować go w katalogu domowym za pomocą polecenia tar *xvfz boff–toolchain.tgz*. W wyniku rozpakowania tego archiwum zostanie utworzony katalog *cross*. Teraz należy przygotować zmienne środowiskowe do kompilacji wydając w terminalu polecenie *source ./cross/compile.env*. Po wykonaniu tej czynności należy rozpakować źródła programu za pomocą polecenia *tar xvfz ioread.tgz*, a następnie przejść do katalogu źródłowego projektu wydając polecenie *cd ioread*. Gdy jesteśmy już w katalogu źródłowym, wydajemy polecenie *make*, w wyniku którego powstanie plik wykonywalny *gpioinfo* na platformę ARM.

Po skompilowaniu programu *gpioinfo* lub wzięciu gotowego skompilowanego programu z mojej strony domowej przystępujemy do skonfigurowania naszego środowiska na ARMputerze. W tym celu logujemy się do ARMputera za pomocą polecenia *ssh root@adres_ip_armputera*, a następnie edytujemy plik konfiguracyjny serwera *lighttpd* wydając polecenie *mcedit /etc/lighthttpd/lighttpd.conf*, gdzie w sekcji server.modules należy na końcu dopisać linię mod_fastcgi:

```
server.modules              = (
            „mod_access”,
            „mod_alias”,
            „mod_accesslog”,
            „mod_compress”,
#           „mod_rewrite”,
#           „mod_redirect”,
#           „mod_status”,
#           „mod_evhost”,
#           „mod_usertrack”,
#           „mod_rrdtool”,
#           „mod_webdav”,
#           „mod_expire”,
#           „mod_flv_streaming”,
#           „mod_evasive”
            „mod_cgi”,
            „mod_fastcgi”
 )
```

a następnie w sekcji cgi.assign dopisać linię umożliwiającą wykonanie skryptów powłoki:

```
cgi.assign = ( „.pl”  => „/usr/bin/
perl”,
„.cgi” => „/usr/bin/perl”,
„.sh” => „/bin/sh”)
```

Po skonfigurowaniu serwera WWW przystępujemy do przekopiowania skryptu oraz programu *gpioinfo* do katalogu głównego serwera WWW, co możemy zrobić wydając z komputera PC polecenie *scp ./gpioinfo root@ip_armputera:/var/www/htdocs* oraz *./gpioinfo root@ip_armputera:/var/www/htdocs*. Powyższe pliki możemy również przekopiować bezpośrednio do odpowiednich katalogów poprzez przełożenie karty SD do czytnika w komputerze PC, bez potrzeby przesyłania plików przez sieć. Po wykonaniu tej czynności mamy już w zasadzie wszystko gotowe, więc restartujemy, bądź uruchamiamy ARMputer, a następnie w pasku adresu przeglądarki wpisujemy *http://adres_armputera/info.sh*.

Strona ta jest automatycznie odświeżana co 5 sekund. Czas odświeżania oraz numery linii portów GPIO, które chcemy odczytywać możemy zmieniać poprzez bezpośrednią modyfikację skryptu *info.sh*.

Zakończenie

Zaprezentowany projekt, pokazuje jak w najprostszy sposób można zacząć tworzyć własne aplikacje z wykorzystaniem ARMputera. Aby tworzyć projekty wystarczy, że poznamy podstawowe tajniki programowania w systemie Linux. Redakcja sukcesywnie, w miarę potrzeb, będzie publikować sterowniki do najbardziej potrzebnych urządzeń dla naszego ARMputera. W najbardziej zaawansowanych przypadkach, będziemy zmuszeni do zapoznania się z tajnikami pisania sterowników dla jądra systemu, ale w podstawowym aspekcie nie jest to również zadanie bardzo skomplikowane.

Lucjan Bryndza, EP
lucjan.bryndza@ep.com.pl