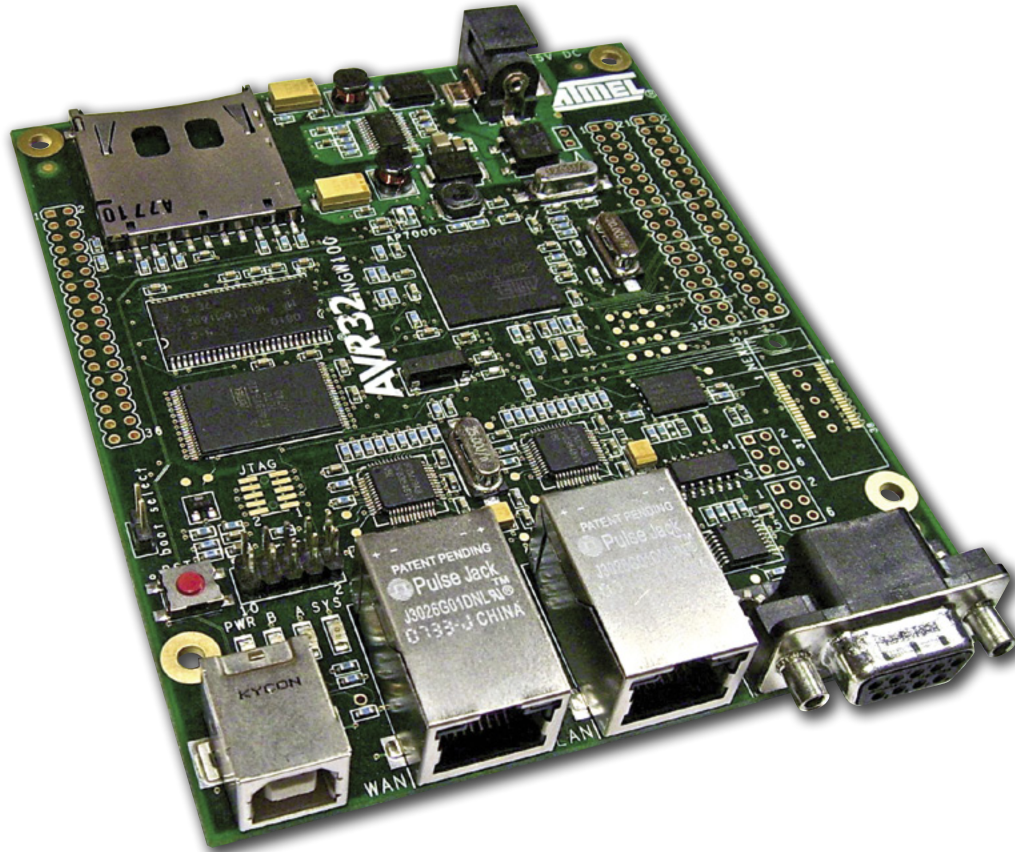
Linux w systemie z AVR32

Zapotrzebowanie na coraz większe moce obliczeniowe i równocześnie coraz większe oczekiwania w stosunku do urządzeń przenośnych przestają nas, konstruktorów, zaskakiwać. Duże częstotliwości taktowania muszą iść w parze z rozwojem oprogramowania, które zaczyna przypominać systemy rodem z komputerów PC. Jedyną możliwością oprogramowania takiego urządzenia w rozsądnym czasie staje się korzystanie z gotowych funkcji i bibliotek. Dlatego coraz częściej w urządzeniach embedded jest stosowany system operacyjny. W artykule opisano Linuksa pracującego na AVR32 AP7000, przy użyciu zestawu ewaluacyjnego ATNGW100.



AP7000 jest mikrokontrolerem z rdzeniem AP7 o architekturze AVR32 zaprojektowanej przez firmę Atmel. Ten 32-bitowy „scalak" może być taktowany z częstotliwością 150 MHz, co daje 210 DMIPS. Jest bogato wyposażony w układy peryferyjne. Umożliwia obsługę LCD (TFT, VGA), ma sprzętowe wsparcie dla wideo, interfejsy obsługi zewnętrznych pamięci (SDRAM, DataFlashTM, SRAM, MMC, SD, CF, Smart Media, NAND), Direct Memory Access, interfejs dla kamer CMOS, dwa interfejsy Ethernet, USB pracujący w trybie *device* oraz dwa 16-bitowe przetworniki A/C audio.

Istnieją dwie płyty ewaluacyjne z AP7000: NGW100 i STK1000. STK1000 należy traktować jako rozszerzenie NGW100, a co za tym idzie, często dostępne materiały (włącznie z tym artykułem) można stosować zamiennie dla obydwu płytek. Oprócz mikrokontrolera NGW100 ma złącze karty SD, 32 MB SDRAM, 8 MB Flash, 8 MB Data Flash (dostęp szeregowy), ATtiny24 jako Board Controller, RS232, dwa interfejsy Ethernet, USB-B i wyprowadzenia niemal wszystkich portów.

W dalszej części artykułu będę używał dwóch terminów: kernel (jądro) i system plików. Czym one się różnią? Kernel jest najważniejszą częścią Linuksa. Nosi nazwę uImage. System plików jest zbiorem plików, programów, plików konfiguracyjnych itp. Czasem zdarza się, że pod nazwą systemu plików kryje się również kernel. Jest tak tylko dlatego, gdyż oprogramowanie, którego będziemy używać, automatycznie kopiuje go do systemu plików. Inna kwestia wymagająca wyjaśnienia to pojawiające się dwa adresy IP. Adres 10.0.1.27 jest adresem komputera PC, natomiast 10.0.1.22 adresem płytki ewaluacyjnej NGW100.

Buildroot

Buildroot jest narzędziem automatyzującym tworzenie systemów embedded dzięki uproszczeniu pracy z cross-kompilatorami. Narzędzie to składa się z zestawu plików *Makefile* i *kconfig*, które ułatwiają generowanie kompletnego systemu embedded Linux. Używanie plików *kconfig* daje użytkownikowi prosty interfejs zapisywany do pliku. *Makefile* czyta wartości tych plików i konfigurują zasady kompilowania, tak aby były one odpowiednie do stosowanej platformy sprzętowej.

Buildroot generuje system plików root, jądro i bootloader. Rozpoczyna on pracę od kompilowania *Toolchaina* (gdy nie chcemy używać zewnętrznego), kernela, biblioteki i aplikacji użytkownika. Potem są dołączane biblioteki, aplikacje i obraz jądra do jednego systemu plików gotowego do zaprogramowania mikrokontrolera. W naszym przypadku jest to NGW100. Końcowy obraz systemu plików jest generowany jako *jffs2*. Często spotykany jest również format *yaffs2*. Obydwa systemy plików są przeznaczone do pracy w urządzeniach z pamięcią Flash. Umożliwiają one optymalizację użycia pamięci Flash.

Buildroot jest zestawieniem skryptów, które są zależne od systemu, na którym są uruchamiane. Dlatego jest istotne, by tworzony program był używany z wirtualnym lub natywnym Linuksem. W Internecie pod nazwą „AVR32 Ubuntu VMware image" jest gotowy obraz Ubuntu 8.04 specjalnie przygotowany do pracy z ATNGW100/STK1000. Użycie darmowej wersji programu *VMware-player-2.5.2-156735* pozwoli skutecznie rozpocząć pracę z tym systemem. Jednak tu użyjemy natywnego Linuksa Ubuntu 9.10 Karmic, dostępnego pod adresem *http://releases.ubuntu.com/karmic/*. Wszystkie kroki powinny także działać pod Ubuntu 8.04.

W celu rozpoczęcie pracy z buildrootem zalogujmy się jako *root.* Z poziomu konsoli wydajemy polecenie

```
su
```

Jeżeli to niemożliwe, należy najpierw ustawić hasło:

```
sudo passwd root
```

Buildroot jest dostępny na oficjalnej stronie Atmela *http://www.atmel.com* –> AVR32 –> Tools&Software –> *Buildroot* for AVR32 AP7 i z niego będziemy korzystać. Jest on również dostępny z innych źró-

deł, chociażby na oficjalnej stronie projektu http://buildroot.uclibc.org. Osobiście odradzam jednak korzystanie z takich wersji, a to ze względu na problemy podczas kompilacji. Do pracy *Buildroot* potrzebuje następujących programów:

```
C compiler (GCC)
C++ compiler (for Qtopia© (G++))
GNU make
sed
flex
bison
patch
gettext
libtool
texinfo
autoconf (version 2.13 and 2.61)
automake
ncurses library (development install)
zlib library (development install)
libacl library (development install)
lzo2 library (development install)
```

Dla przyspieszenia pierwszych kroków przygotowałem listę wszystkich niezbędnych nazw programów, które trzeba zainstalować, jednak przed instalacją należy zaktualizować repozytorium programu *apt* za pomoca poleceń:

```
apt-get update
apt-get -y install sed flex bison patch gettext texinfo libtool automake build-essential libcurses-ocaml-dev libtexttools-dev libvuurmuur-dev zlib1g-dev liblzo2-dev libacl1-dev
```

Teraz system jest już wyposażony we wszystkie niezbędne aplikacje do pracy z *Buildrootem*. Uruchamiamy komendę (z katalogu buildroota):

```
make menuconfig
```

Za pomocą interfejsu pseudograficznego pod konsolą jesteśmy w stanie ustawić wszystkie opcje związane z obrazem kernela, dodać obsługę różnych systemów plików, aplikacji użytkownika, opcje związane z bootloaderem i wiele innych. W zasadzie jesteśmy w stanie zmienić wszystko, by po kompilacji otrzymać gotowy plik, który zostanie wgrany na pytę NGW100. Jednak samodzielne konfigurowanie może przysporzyć wielu kłopotów, dlatego programiści z firmy Atmel przygotowali gotową konfigurację przeznaczoną tylko dla płyty NGW100:

```
make atngw100_defconfig
```

Zawiera ona większość aplikacji użytecznych dla tej płyty, zaś *atngw100-base_defconfig* stanowi tylko minimalne zestawienie aplikacji służące jako punkt startu pracy z tym sprzętem.

Teraz wszystkie ustawienia są odpowiednie. Nic nie stoi na przeszkodzie, by je zmieniać, lecz na razie zajmijmy się skompilowaniem kernela, zaznaczonych aplikacji użytkownika i uBoota. Proces uruchamia się jedną komendą (wymagane jest co najmniej 3 GB wolnej przestrzeni na dysku PC): `make`.

Po **kilku godzinach** powinniśmy otrzymać komunikaty końcowe, świadczące o procesie zakończonym sukcesem. Niestety, kompilowanie trwa bardzo długo i jest podatne na różne błędy. Dlatego, w celu zaoszczędzenia czasu, postaram się opisać możliwe przyczyny kilku z nich i ich rozwiązania.

Zdarza się, że proces kompilowania zatrzymuje się, ponieważ źródło, z którego ma być pobierany plik źródłowy, nie odpowiada. Rozwiązanie jest proste: należy przerwać program (Ctrl+C) i ponowić kompilację (make). Na szczęście program make sprawdza, które pliki są już skompilowane, a które nie, więc kompilacja zaczyna się od tego miejsca, w którym została przerwana.

Często występujące komunikaty na temat braku plików źródłowych są spowodowane tym, że niektóre linki źródłowe podane w buildroocie nie istnieją. Wskazany plik należy odszukać w Internecie i skopiować go do katalogu *dl/*.

Pojawienie się komunikatu *„scripts/unifdef.c:209: error: conflicting types for 'getline'/usr/include/stdio.h:651: note: previous declaration of 'getline' was here"* oznacza, że wersja kompilatora gcc jest niewłaściwa. Komunikaty te występują niezależnie od wersji buildroota (2.2.1 i 2.3.0). Rozwiązaniem jest zamiana funkcji *getline()* na *parseline()* we wskazanym przez kompilator pliku (prawdopodobnie będzie to *unifdef.c*). Inne rozwiązanie to kompilowanie pod systemem Ubuntu 9.04. Komunikat „

```
checking how to run the C++ preprocessor... /lib/cpp
configure: error: C++ preprocessor „/lib/cpp" fails sanity check
See `config.log' for more details.
make: *** [/home/darek/buildroot/buildroot-avr32-v2.3.0/toolchain_build_avr32/gmp-4.2.2-host/.configured] Error 1"
```

oznacza brak zainstalowanych wszystkich, niezbędnych kompilatorów. Rozwiązanie: `apt-get install build-essential`.

Komunikat oznajmiający brak pliku „avr_guide_doc" przerywa kompilowanie. Rozwiązaniem jest odznaczenie tejże instrukcji z płyty NGW za pomocą komendy:

```
make menuconfig->Packages Selection for the target->disable avr32-web-start and avr32-wiki-docs
```

Inne użyteczne komendy:

- `make source` powoduje pobranie plików źródłowych potrzebnych do skompilowania, zgodnie z ustawieniami zawartymi w buildroocie. Ułatwia to pracę offline.
- `make clean` usuwa wszystkie pliki wynikowe. Pliki źródłowe z katalogu *dl/* pozostają nieusunięte.

Minicom

Połączenie z płytą NGW100 można zrealizować na dwa sposoby. Poprzez połączenie ssh – kablem Ethernet bądź poprzez szeregowe złącze wyprowadzone na zewnątrz płyty – najlepiej użyć dowolnej przejściówki USB-serial. Drugie rozwiązanie niesie więcej korzyści, ponieważ można także kontrolować pracę programu bootloader i przeglądać na bieżąco logi podczas startu urządzenia. Dzięki czemu możliwe jest wgranie systemu plików na płytę czy też update uBoota. Do obsługi szeregowego interfejsu może posłużyć terminal Minicom. Instaluje się go poleceniem: *apt-get install minicom*.

Po podłączeniu przejściówki USB-serial warto sprawdzić jej nazwę w */dev/* za pomocą polecenia `ls /dev/tty*`.

Prawdopodobnie jej nazwą będzie ttyUSB0. Konfigurowanie Minicoma wykonuje się poleceniem: `minicom -s`. Ewentualnie można zmieniać port i szybkość transmisji danych:

```
Serial port setup->Serial Device: /dev/ttyUSB0
Serial port setup->Bps/Par/Bits: 115200 8N1
```

Po skonfigurowaniu należy zapisać nastawy. Minicom jest gotowy do pracy i można go uruchomić za pomocą polecenia `minicom`.

uBoot

uBoot jest bootloaderem open source dla systemów wbudowanych. Wspiera architektury wielu mikroprocesorów, między innymi PowerPC, ARM, AVR32 itp. Pozyskanie uBoota AVR32 jest możliwe z oficjalnej strony projektu, ze strony Atmela, a przede wszystkim generowany jest on przez buildroota (buildroot-avr/binaries/atngw100/u-boot.bin). Wejście do uBoota następuje po wciśnięciu spacji podczas uruchamiania się płyty NGW.

Podczas różnych prac z uBootem warto wiedzieć, jakiego rodzaju dane znajdują się w pamięci Flash, aby być pewnym, co się np. z niej usuwa. W **tabeli 1** zamieszczono ogólny podział przestrzeni adresowej pamięci Flash.

uBoot zawiera w sobie wiele komend, w dalszej części artykułu będą opisywane tylko te, które zostały użyte. Opis komend można znaleźć w Internecie pod adresem *http://www.avrfreaks.net/wiki/index.php/Documentation:AVR32_Linux_Development/U-boot_command_reference*

Aktualizacja uBoota

Niejednokrotnie może zajść potrzeba uaktualniania bootloadera. Jest to możliwe za pomocą programatorów ATJTAGICE2 lub ATAVRONE. Są to jednak profesjonalne, kosztowne programatory. Do zastosowań „mniej profesjonalnych" można użyć programu ***FlashUpgrade***. Jest on ładowany poprzez uBoota do pamięci SDRAM i stamtąd uruchamiany. Jego działanie polega na usunięciu istniejącego bootloadera z pamięci Flash i wgraniu w to miejsce nowego. Na etapie kompilowania *FlahUpgrade* możemy skopiować swojego uBoota, który będzie potem wgrany na płytę. Zatem, na początku instalujemy program wersjonujący git (*apt-get install git-core*) i pobieramy kody źródłowe *FlashUpgrade* (*git clone git://www.atmel.no/~hcegtvedt/flash-upgrade.git flash-upgrade*). Przed rozpoczęciem kompilowania programu w systemie powinien być prawidłowy buildroot, ponieważ w nim znajdują się niezbędne kompilatory. W celu umożliwienia ich automatycznego zlokalizowania należy do zmiennej PATH dodać ścieżkę dostępu np. w następujący sposób: *export PATH=$PATH:$HOME/buildroot/build_avr32/staging_dir/bin.* Należy również upewnić się, czy z każdej lokalizacji jest widoczny np. kompilator *avr32-linux-g++*.

Do katalogu programu kopiujemy obraz bootloadera dostępnego z buildroota *buildroot/binaries/atngw100/u-boot.bin* i kompilujemy komendą *make.* Po bezbłędnym skompilowaniu otrzymamy plik *flash-upgrade.uimg*.

Wygenerowany program może być załadowany z użyciem uBoota przez kartę SD/MMC, TFTP lub przez port szeregowy. W naszym przypadku nośnikiem danych będzie karta SD. Używając komputera stacjonarnego, formatujemy ją na format plików EXT2:

```
sudo umount /dev/sdb1
sudo mkfs.ext2 -L NGW100CARD /dev/sdb1
```

Nazwa sdb1 może być inna, zależna od komputera. Można to sprawdzić w folderze */dev.* Możesz także użyć graficznego programu partycjonującego gparted (*apt-get install gparted*).

Po sformatowaniu i zamontowaniu karty poleceniem `mount /dev/sdb1 /media/card` kopiujemy do jej katalogu głównego plik *flash-upgrade.uimg* (dostępny także na *http://www.atmel.no/buildroot/buildroot-u-boot.html*. Następnie uruchamiamy komendy:

```
sync
cd /
umount /dev/sdb1
```

i umieszczamy kartę SD w gnieździe płyty NGW100, po czym restartujemy płytę. Za pomocą programu *Minicom* wchodzimy do uBoota naciskając w odpowiednim czasie klawisz spacji. Uruchamiamy komendę uBoota inicjalizującą kartę SD `mmcinit` i ładujemy plik z karty do pamięci SDRAM (`ext2load mmc 0:1 0x10400000 /flash-upgrade.uimg`). Następnie uruchamiamy *FlashUpgrade* (`bootm 0x10400000`). Teraz trzeba stosować się do instrukcji pojawiających się w oknie konsoli. Podczas aktualizacji nie wolno wyłączać zasilania płyty lub naciskać klawisza Reset, ponieważ później jedyną metodą wgrania uBoota może okazać się użycie programatora.

Możesz także wgrać do pamięci SDRAM *flash-upgrade.uimg* innymi drogami, patrz na sposoby wgrywania systemu plików do flash.

Toolchain

Toolchain zawiera w sobie kompilator asemblera, linker, binutils (zestaw narzędzi w postaci binariów), kody źródłowe bibliotek (*Newlib* i *uClibc*), narzędzie do programowania (*avr32program*) i debugowania (*AVR32 GDB*, *avr32gdbproxy* i *avr32trace*).

Toolchain ma wsparcie dla wszystkich mikrokontrolerów AVR32. Jego instalacja jest możliwa na zarówno pod kontrolą Windows, jak i Linuksa. Tu zajmiemy się instalacją pod Linuksem.

Istnieje parę metod pozyskania Toolchaina. Można go pobrać ze strony firmy Atmel (*http://www.atmel.com* –> *AVR32* –> *Tools&Software* –> *Buildroot for AVR32 AP7*), gdzie pod nazwą *AVR32 GNU Toolchain 2.4.2* jest umieszczona wersja dla Linux Ubuntu 9.10. Dla innych dystrybucji, należy pobrać inną wersję Toolchaina.

Tabela 1. Ogólny podział przestrzeni adresowej pamięci Flash

0x000000–0x01FFFF	Partycja uboot. Jeżeli dane zostaną z niej usunięte, jedynym sposobem odzyskania komunikacji z płytą będzie użycie programatora JTAGICEmkII
0x020000–0x7EFFFF	Partycja Linux root. Podczas zmieniania systemu plików i kernela właśnie dane z tej partycji będą usuwane
0x7F0000–0x7FFFFF	W tej przestrzeni trzymane są wartości zmiennych środowiskowych uBoota. Jeżeli będziesz wykonywał update uBoota, wartości zmiennych pozostaną nienaruszone

Za pomocą komendy `unzip avr32_gnu_toolchain_2.4.2_ubuntu_910.zip` rozpakowujemy ściągnięty plik i instalujemy wszystkie repozytoria (`dpkg -i *.deb`). Druga metoda instalacji Toolchaina polega na użyciu programu Apt, zarządzającego pakietami. Pierwszy krok polega na dodaniu na końcu pliku */etc/apt/sources.list* linijki

```
deb http://distribute.atmel.no/tools/avr32/release/ubuntu/ karmic main
```

Jeśli jest używany inny Linuks niż Ubuntu 9.10 Karmic, to powyższy link do repozytorium należy zastąpić właściwym. Następnie odświeżamy listę dostępnych pakietów (`apt-get update`) i instalujemy Toolchaina (`apt-get install avr32-gnu-toolchain`). Jeżeli pojawią się nowsze wersje Toolchaina i chcielibyśmy je zainstalować, wystarczy wpisać komendy

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Jeśli program Apt wyświetla na konsoli komunikaty o błędach dotyczące zależności, które przerywają instalację, to prawdopodobnie jest nieprawidłowy link do plików źródłowych dla danej dystrybucji i wersji Linuksa.

Pierwszy program „Hello world!" wykonany za pomocą AVR32 Studio

AVR32 Studio jest zintegrowanym środowiskiem przeznaczonym do programowania mikrokontrolerów AVR32 napisanym w oparciu o popularny program Eclipse. Program umożliwia tworzenie i edycję wieloplikowych projektów, ma wbudowany edytor C/C++ z podświetlaniem składni, automatycznym dokańczaniem kodu, wersjonowaniem plików CVS, wsparciem dla debugowania i wiele innych ciekawych funkcji.

Przed rozpoczęciem pracy z AVR32 Studio należy zainstalować Java Runtime Environment 1.6 (nowsza może nie działać prawidłowo): `apt-get -y install sun-java6-jre`. Zainstalowaną wersję Javy możemy sprawdzić, wydając komendę *java –version.*

AVR32 Studio jest dostępne na stronie Atmela. Po pobraniu odpowiedniej wersji pod Linuksa rozpakowujemy plik `unzip avr32studio-ide-2.6.0.753-linux.gtk.x86.zip`. W zasadzie „instalacja" dobiegła końca. Teraz ważnym elementem niezbędnym do prawidłowej pracy programu jest zestaw kompilatorów. By program „widział" je, należy przed jego uruchomieniem dodać ścieżkę „`export PATH=$PATH:/$HOME/buildroot-avr32/build_avr32/staging_dir/bin`" do zmiennej PATH. Toolchain nie zawiera już w sobie kompilatora *avr32-linux-g++*, którego będziemy używali, ponieważ Atmel zaprzestał wpierania niektórych kompilatorów w toolchainie i są one dostępne po kompilacji buidroota.

Po zainstalowaniu AVR32 Studio i kompilatorów, można przystąpić do napisania pierwszego programu.

1. Uruchamiamy program z linii komend *./avr32studio*, z menu *File* wybieramy *New* –> *AVR C++ Project*.
2. Podajemy nazwę projektu, *Target device* – AT32AP7000, *Project type* – AVR32 Linux Executable.
3. Naciskamy prawym przyciskiem myszy na nowo utworzonym projekcie (po lewej stronie) *New* –> *Source File*, *Source file* – main.cpp
4. Przepisujemy program z **listingu 1** i kompilujemy, wybierając z menu *Project* –> *Build All.*

W wyniku bezbłędnej kompilacji będzie wygenerowany plik *main.elf*, który zostanie wgrany do płyty NGW i uruchomiony.

Napisanie programu „Hello world!" bez użycia AVR Studia

Teraz napiszemy program demonstracyjny „Hello world!" bez użycia AVR32 Studia. Ma to na celu zaznajomienie się z kompilatorem oraz z nowym sposobem dostarczenia pliku wynikowego na NGW100.

By móc kompilować aplikacje na Linuksa, należy mieć prawidłowo zbudowanego buildroota i dodaną wcześniej do zmiennej *PATH* ścieżkę dostępu do kompilatorów. Tworzymy na komputerze stacjonarnym tworzymy plik *main.c* i przepisujemy program z list. 2. Kompilujemy go, wydając komendy:

```
avr32-linux-g++ -pipe -O2 -g -Wall -D_GNU_SOURCE  -c -o main.o main.c
avr32-linux-gcc -pipe -O2 -g -Wall -o main.elf main.o
```

Użycie parametru `-Wall` włącza wyświetlanie wszystkich ostrzeżeń podczas kompilowania.

Powstały plik *main.elf* kopiujemy do katalogu /nfs/tftp. Do tego celu jest potrzebny zainstalowany, opisany w dalej, serwer tftp (*tftp-d-hpa*). Uruchamiamy płytę NGW100, a gdy pojawi się znak zachęty, konfigurujemy adres IP i maskę sieciową: `ifconfig eth0 inet 10.0.1.22 netmask 255.255.255.0`.

Dla pewności można spróbować można użyć komendy *ping,* jako docelowy podając adres komputera PC. Jeżeli jest odpowiedź, ściągamy plik *main.elf* za pomocą tftp i uruchamiamy go:

```
tftp -r main.elf -g 10.0.1.27
chmod 777 main.elf
./main.elf
```

Jeśli w oknie konsoli pojawi się komunikat „*Hello World!*", to oznacza, że jesteśmy gotowi na tworzenie większych programów.

Upgrade Linuks – TFTP

Wgranie nowego systemu plików na płytę NGW100 jest możliwe między innymi za pomocą sieci Ethernet z użyciem protokołu TFTP. Płyta NGW100 musi być podłączona do sieci za pomocą gniazda *LAN*. Na początku instalujemy jeden z serwerów TFTP na komputerze PC:

```
sudo su -
apt-get install tftpd-hpa
mkdir -p /nfs/tftp
pico /etc/default/tftpd-hpa
```

Następnie zmieniamy plik edytowalny, jak niżej:

```
RUN_DAEMON="yes"
OPTIONS="-l -s /nfs/tftp"
```

Za pomocą komendy „`/etc/init.d/tftpd-hpa start`" uruchamiamy program *tftpd-hpa.*

Kopiujemy plik *rootfs.avr32.jffs2-root*, który wygenerowaliśmy za pomocą buildroota (`buildroot-avr32/binaries/atngw100`) do katalogu */nfs/tftp*. Problemem podczas ściągania może być brak dostępu do pliku, więc warto wcześniej użyć komendy *chmod 777 /nfs/tftp/** pozwalającej wszystkim korzystać z plików w tym katalogu. Przechodzimy do linii komend uBoota za pomocą HyperTerminalu i wydajemy polecenia:

```
Uboot> askenv bootcmd
Please enter ,bootcmd': fsload boot/uImage; bootm
Uboot> set bootargs ,console=ttyS0 root=/dev/mtdblock1 rootfstype=jffs2'
Uboot> set ipaddr 10.0.1.22
Uboot> set serverip 10.0.1.27
Uboot> set tftpip 10.0.1.27
Uboot> protect off 0x20000 0x7EFFFF
Uboot> erase 0x20000 0x7EFFFF
Uboot> tftp 0x90000000 rootfs.avr32.jffs2-root
Uboot> cp.b 0x90000000 0x20000 $(filesize)
Uboot> protect on all
Uboot> saveenv
Uboot> boot
```

Na początku ustalamy komendy odseparowane średnikiem, które mają się uruchomić podczas startu. Z racji tego, że nasz system plików wraz z kernelem będą znajdować się w pamięci Flash, pierwsza komenda ładuje kernel z systemu plików jffs2 pod standardowy adres.

Bootm uruchamia kernela załadowanego pod wskazanym adresem. U nas z racji braku argumentu adresu oznacza to, iż jest on zdefiniowany w zmiennych środowiskowych uBoota.

Bootargs przekazuje argumenty kernelowi na jakim porcie znajduje się konsola, gdzie jest system plików root i jakiego jest on typu. Potem ustawiamy adres IP płyty NGW100, adres serwera, na którym jest zainstalowany serwer TFTP (czyli adres komputera PC). Następnie odblokowujemy dostęp do wybranego zakresu adresów i czyścimy je. W podanym przedziale mieści się partycja *root* Linuksa.

Komenda *tftp* ładuje plik z serwera TFTP i kopiuje go na wskazany adres. U nas jest to pamięć SDRAM. Kolejnie *cp.b* kopiuje zawartość pamięci, począwszy od adresu SDRAM 0x90000000 do pamięci Flash pod adres 0x20000. Długość danych do skopiowania definiuje zmienna *filesize*, którą równie dobrze moglibyśmy zastąpić liczbą. Zmienna ta jest zapisana odpowiednią wartością podczas wykonania wcześniejszej komendy *tftp*. Zabezpieczenie zapisu pamięci Flash, zapisanie zmiennych uBoota i uruchomienie systemu stanowią ostanie komendy.

Upgrade Linuks – SD

Inną metodą przeniesienia systemu plików na płytę NGW jest użycie karty SD. W zasadzie niewiele różni się ona od poprzedniej. Należy na komputerze PC skopiować na kartę SD z systemem plików EXT2 plik *rootfs.avr32.jffs2-root*, przełożyć kartę do NGW100 i wejść do uBoota. Poniżej lista komend, jakie należy uruchomić:

```
Uboot> askenv bootcmd
Please enter ,bootcmd': fsload boot/uImage; bootm
Uboot> set bootargs ,console=ttyS0 root=/dev/mtdblock1 rootfstype=jffs2'
Uboot> protect off 0x20000 0x7EFFFF
Uboot> erase 0x20000 0x7EFFFF
Uboot> mmcinit
Uboot> ext2load mmc 0:1 0x90000000 rootfs.avr32.jffs2-root
Uboot> cp.b 0x90000000 0x20000 0x340000
Uboot> protect on all
Uboot> saveenv
Uboot> boot
```

M*mcinit* inicjalizuje kartę SD, a komenda *ext2load* ładuje plik z karty SD pod wskazany adres.

Uruchamianie z karty SD

Po rozpoczęciu eksperymentowania z systemem plików szybko okaże się, że wewnętrzna pamięć Flash o wielkości 8 MB okazuje się stanowczo za mała. Dlatego warto pracować na systemie plików umieszczonych na karcie SD/MMC.

Na początku powinniśmy zdobyć system plików root. Można go pobrać np. ze strony *http://www.atmel.no/buildroot/buildroot-bin.html* z obrazem kernela włącznie, jednak lepszym rozwiązaniem jest użycie wygenerowanego przez siebie systemu plików z buildroota z obrazem kernela (`/buildroot-avr32/binaries/atngw100/rootfs.avr32.tar`).

Za pomocą komputera PC formatujemy kartę SD na system plików EXT2. Rozpakowujemy na kartę SD plik *rootfs.avr32.tar*. Na nowo sko-

Lisiting 1. Program demonstracyjny „Hello world!"

```
#include <stdio.h>
int main(int argc, char** argv){
  printf(„Hello World!\n");
  return 0;
}
```

piowanym na kartę SD systemie plików, w katalogu *boot*, jest widoczny plik *uImage*. Jest to obraz kernela. Jeżeli chcemy użyć innego jądra, wystarczy podmienić plik. Dlaczego akurat w katalogu *boot*? Jest to umowny katalog, jednak ważne, aby w uBoot'cie ścieżka do kernela była podana prawidłowo:

```
root@user:/home/user# mkfs.ext2 -L NGW100 /dev/sdc1
...
root@user:/home/user# cp /home/user/buildroot-avr32/binaries/atngw100/rootfs.avr32.tar /media/NGW100/
root@user:/# cd /media/NGW100/
root@user:/media/NGW100# tar -xvf rootfs.avr32.tar
root@user:/media/NGW100# rm rootfs.avr32.tar
```

Uruchamiamy uBoota naciskając w odpowiednim momencie spację. Po pojawieniu się znaku zachęty *U-Boot>* wydajemy komendy jak niżej:

```
Uboot> askenv bootcmd
Please enter ‚bootcmd':mmcinit; ext2load mmc 0:1 0x10400000 /boot/uImage; bootm
Uboot> set bootargs ‚console=ttyS0 root=/dev/mmcblk0p1 rootwait'
Uboot> saveenv
Uboot> boot
```

Pierwszą komendą ustawiamy listę komend, które mają się uruchomić podczas startu urządzenia. *Mmcinit* inicjalizuje kartę SD. Komenda *ext2load mmc 0:1 0x10400000 /boot/uImage* ładuje plik binarny znajdujący się w podanej ścieżce na karcie SD z systemem plików EXT2 do adresu 0x10400000. *Bootm* uruchamia załadowanego wcześniej Linuksa z domyślnego adresu. Drugą komendą ustawiamy wartość zmiennej zawierającej parametry, które zostaną przekazane do kernela podczas uruchamiania. Następnie zapisujemy do pamięci Flash ustawienia uBoota i uruchamiamy system.

Moduł LED

Pierwszą kwestią wymagającą wyjaśnienia w temacie oprogramowania jest różnica pomiędzy modułem a aplikacją. Aplikacja wykonuje pewne zadanie, począwszy od jego uruchomienia aż do zakończenia. Moduł zaś rejestruje się, by obsłużyć pewne żądania. Aplikacje mają dostęp tylko do przestrzeni użytkownika, zaś moduły pracują w przestrzeni jądra. Oznacza to, że dzięki modułowi będziemy mogli zmieniać wartości rejestrów mikrokontrolera AP7000. Bezpośrednie odwołanie się do rejestrów przez aplikacje jest niemożliwe ze względów bezpieczeństwa.

Klasy urządzeń i modułów dzielimy na trzy rodzaje:
- urządzenia znakowe,
- urządzenia blokowe,
- interfejsy sieciowe.

Linuks potrafi załadować każdy z typów urządzeń jako moduł. Nas interesuje w tym momencie tylko urządzenie znakowe. Jest to urządzenie, do którego dostęp jest taki sam, jak do pliku, czyli można wykonać na nim wywołania systemowe, takie jak: *open*, *close*, *read*, *write*. Oczywiście, muszą one być zaimplementowane w danym module, aby można było jej wywołać. W naszym przypadku żadne z wymienionych nie będzie implementowane. Zaprezentowany moduł jest okrojony do minimum, tak by pokazać tylko funkcjonalność przy maksymalnej prostocie. Występują w nim tylko dwie funkcje. Rolą funkcji *init_module()* jest przygotowanie do późniejszego wywoływania funkcji modułu. W jej ciele powinny znajdować się procesy inicjalizujące. Funkcja ta wykonywana jest podczas ładowania modułu komendą *insmod*. Podczas usuwania modułu wykonywana jest funkcja *cleanup_module()*. W niej odwrotnie do wcześniejszej funkcji, powinny znajdować się kroki „sprzątające" po już nieistniejącym module. Wykonywana jest ona podczas komendy *rmmod*. Funkcja *printk* jest zdefiniowana w jądrze Linuksa i jej działanie jest podobne do *printf*. Na **listingu 2** umieszczono kod źródłowy omawianego modułu.

Jego jedyną rolą jest zapalanie i gaszenie równocześnie diod LED A i B dostępnych na płycie NGW100 podczas ładowania i deinstalacji modułu. Dołączany plik *port_defs.h* zawiera adres bazowy rejestrów portu A i przesunięcia pomiędzy kolejnymi rejestrami tego portu. Nad kompilacją modułu czuwa plik *makefile*. Za pomocą komendy *make* tworzy się plik *testmodule.ko*, który zostanie skonsolidowany z jądrem na płycie NGW100. Należy go skopiować na kartę SD płyty ewaluacyjnej, nadać prawa dostępu, załadować moduł do jądra, co powinno objawić się zaświeceniem się diod LED. Następnie należy odinstalować moduł, obserwując ich zgaszenie. Poniżej umieszczono komendy, których trzeba użyć dla NGW100:

```
~ # chmod 777 testmodule.ko
~ # insmod testmodule.ko
hello led
Led on A
Led on B
~ # rmmod testmodule.ko
LED off A
LED off B
~ #
```

```
# Makefile:
ARCH := avr32
CROSS_COMPILE := avr32-linux-
KDIR := /sciezka/do/buildroota/project_build_avr32/atngw100/linux-2.6.27.6
PWD := $(shell pwd)
.
obj-m := testmodule.o
default:
<------><------>$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) module
.
clean:
<------><------>rm -rf testmodule.ko testmodule.o* testmodule.mod* .testmodule* .tmp* Module*
```

Dariusz Rzędowski
batmanmen@tlen.pl

Listing 2. Kod źródłowy przykładowego modułu w języku C

```
/*
* testmodule.c
* Author: Dariusz Rzedowski
*/

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/io.h>
#include „port_defs.h"
#define LED_A 19
#define LED_B 19
#include <linux/delay.h>
#define _delay_ms(x) msleep(x)
int init_module(void){
  printk(„<1>hello led\n");
  //Enables the PIO to control the corresponding pin
  __raw_writel( (1<<(LED_A)), PORTA + PIO_PER);
  //Enables the output on the I/O line.
  __raw_writel( (1<<(LED_A)), PORTA + PIO_OER);
  //Clears the data to be driven on the I/O line.
  __raw_writel( (1<<(LED_A)), PORTA + PIO_CODR);

  //Enables the PIO to control the corresponding pin
  __raw_writel( (1<<(LED_B)), PORTE + PIO_PER);
  //Enables the output on the I/O line.
  __raw_writel( (1<<(LED_B)), PORTE + PIO_OER);
  //Clears the data to be driven on the I/O line.
  __raw_writel( (1<<(LED_B)), PORTE + PIO_CODR);
  printk(„<1>Led on A\n");
  printk(„<1>Led on B\n");
  return 0;
}

void cleanup_module(void)
{
  //Sets the data to be driven on the I/O line.
  __raw_writel( (1<<(LED_A)), PORTA + PIO_SODR);
  //Sets the data to be driven on the I/O line.
  __raw_writel( (1<<(LED_B)), PORTE + PIO_SODR);
  _delay_ms(1000);
  printk(„<1>LED off A\n");
  printk(„<1>LED off B\n");
}
```