Mikrokontrolery STR91x od podstaw, część 2

Narzędzia dla ARM9

W drugiej części artykułu przedstawiamy konfigurację bezpłatnych narzędzi programowych umożliwiających realizację projektów dla mikrokontrolerów STR91x. Okazuje się, że praktycznie wszystkie narzędzia są dostępne bezpłatnie. Za miesiąc opublikujemy w EP płytę CD–ROM zawierająca wszystkie niezbędne programy.

Mikrokontrolery z rdzeniem ARM są wykorzystywane w bardziej zaawansowanych projektach, dlatego do ich programowania będziemy używać najczęściej języków C lub C++. Na rynku istnieje wiele komercyjnych kompilatorów ze zintegrowanym środowiskiem IDE np. *Keil ARM* czy *Rowley Cross Works Studio for ARM*. Zawierają one rozbudowane edytory projektów, symulatory oraz wiele innych narzędzi ułatwiających pracę, jednak ich ceny sprawiają, że są one niejednokrotnie poza zasięgiem małych firm, nie mówiąc już o warsztacie przeciętnego elektronika. Możemy wykorzystać wersję ewaluacyjne komercyjnych narzędzi, ale wówczas musimy liczyć się z ograniczeniami generowanego kodu wynikowego (*Keil ARM – do 16 kB*), lub ograniczeniami czasowymi (*Rowley Cross Works* – 30 dni). Dużo lepszym rozwiązaniem jest wykorzystanie oprogramowania *open–source*, które jest pozbawione wszelkich ograniczeń, a swoją funkcjonalnością niejednokrotnie nie będzie odbiegać od rozwiązań komercyjnych.

Do projektowania aplikacji dla ARM–ów wykorzystać będziemy zintegrowane środowisko programistyczne (*IDE*) *Eclipse*. *Eclipse* pierwotnie zostało zaprojektowane do pisania aplikacji w języku Java, jednak po zainstalowaniu dodatkowego pluginu *CDT* umożliwia również pisanie oprogramowania w języku C/C++. Do kompilacji programów posłużymy się doskonałym kompilatorem języka C/C++ GCC, natomiast do programowania mikrokontrolerów STR91x oraz debugowania programów posłuży nam doskonałe narzędzie *openocd/gdb*. Kompilator, programator oraz debugger dają się doskonale zintegrować ze środowiskiem *Eclipse*, tak, więc po poświęceniu odrobiny czasu na zainstalowanie i skonfigurowanie wszystkich narzędzi staniemy się posiadaczami doskonałego środowiska dla mikrokontrolerów ARM, które nie posiada żadnych ograniczeń. Opisany zostanie tutaj sposób instalacji środowiska w Linuxie (dystrybucja Ubuntu 7.10) oraz w Windows.

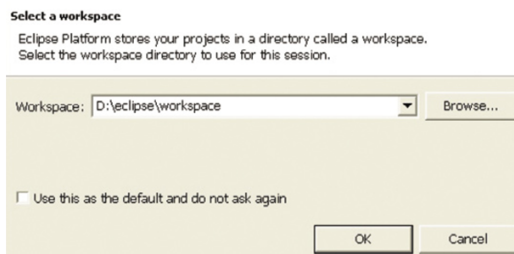
Przed rozpoczęciem instalacji programu *Eclipse* w Windows musimy upewnić się, że na komputerze została zainstalowana maszyna wirtualna Javy (*JRE – Java Runtime Environment*), która jest niezbędna do jego działania. Możemy to sprawdzić poprzez panel sterowania w zakładce *Dodaj/Usuń Programy*. W przypadku braku maszyny Java musimy pobrać jej najnowszą wersję ze strony *http://www.java.com/en/download/* i następnie zainstalować ją w systemie. W przypadku Linuxa wystarczy w managerze pakietów *Synaptic* odinstalować pakiet *gij* oraz zainstalować pakiet *sun–java6–jre* (można to zrobić również z konsoli za pomocą komend *sudo apt–get remove gij* oraz *sudo–apt get install sun–java5–jre*). Kolejną czynnością w przypadku instalacji Windows jest pobranie najnowszej wersji środowiska *Eclipse* ze strony: *http://www.eclipse.org/downloads/*.

Potrzebna będzie również nakładka CDT, umożliwiająca tworzenie projektów w języku C/C++. (pierwotnie środowisko *Eclipse* zostało zaprojektowanie do pisania aplikacji w języku Java) Nakładkę CDT w najnowszej pobieramy ze strony *http://www.eclipse.org/cdt/*. Instalację rozpoczynamy od rozpakowania pliku *Eclipse*, (który występuje w postaci archiwum zip) w dowolnie wybrane miejsce docelowe, np. *d:\programy.* Następnie należy rozpakować nakładkę CDT w to samo miejsce gdzie poprzednio rozpakowaliśmy *Eclipse* (np. *d:\programy*). Po rozpakowaniu plików w zależności od własnych preferencji tworzymy skrót do pliku *eclipse.exe* (plik ten znajduje się w podkatalogu *eclipse)* na pulpicie lub w menu *Start*.

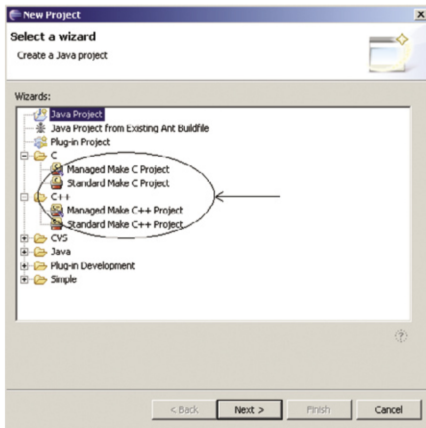
W przypadku *Ubuntu Linux* instalacja jest dużo prostsza wystarczy jedynie w managerze pakietów *Synaptic* wybrać do instalacji pakiety *eclipse* i *eclipse–cdt* oraz kliknąć przycisk *Zastosuj*. Gdy mamy już wstępnie zainstalowany edytor *Eclipse*, możemy go uruchomić w celu sprawdzenia poprawności jego działania. Przy pierwszym uruchomieniu pojawi się okno dialogowe przedstawione na **rys. 4**.

Wpisujemy w nim domyślny katalog, w którym będą przechowywane pliki wszystkich projektów. Po pojawieniu się okna głównego aplikacji z menu wybieramy opcję *File–>New–>Project*. Na ekranie pojawi się okno dialogowe służące do tworzenia nowego projektu przedstawione na **rys. 5**.

Do prawidłowej pracy narzędzi w środowisku Windows niezbędna jest instalacja środowiska *Cygwin*, natomiast użytkownicy Linuxa nie mają takiego problemu. Z ogromu dostępnych narzędzi interesować nas będzie *make* służące do automatyzacji procesu kompilacji projektów oraz kompilator *gcc* dla systemu Windows. Kompilator dla Windows wykorzy-



Rys. 4. Okno wyboru katalogu roboczego projektów w Eclipse



Rys. 5. Okno dialogowe tworzenia nowego projektu

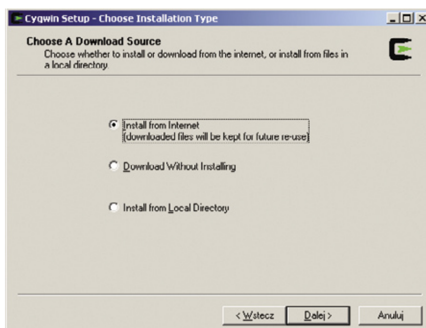
stywany będzie przez nakładkę CDT do wstępnego kompilowania plików źródłowych na potrzeby systemu inteligentnych podpowiedzi oraz eksploratora projektu.

Instalację Cygwina rozpoczynamy od ściągnięcia i uruchomienia pliku *setup.exe* ze strony: *www.cygwin.com* (link *install or update now*) Po uruchomieniu zobaczymy okno powitalne instalatora, w którym należy kliknąć przycisk *Dalej>* pokaże się wówczas okno wyboru źródła instalacji przedstawione na **rys. 6**.

Po zatwierdzeniu tej opcji wyświetli się dialog ustawień konfiguracyjnych środowiska przedstawiony na **rys. 7**.

W polu tekstowym *Root Directory*" wpisujemy katalog, gdzie będzie zainstalowane środowisko Cygwin, natomiast pozostałe opcje ustawiamy tak jak na rys. 7. Po zatwierdzeniu ustawień pojawi się ekran z oknem tekstowym służącym do wpisania ścieżki, gdzie będą zapisane pliki pobrane przez instalator, możemy tutaj pozostawić domyślną wartość.

Pozostałe pakiety możemy zostawić na domyślnym poziomie instalacji chyba, że zamierzamy w przyszłości pracować bezpośrednio w środowisku Cygwin, wówczas możemy zainstalować wszystkie pakiety wybierając: *All Install*. Po potwierdzeniu wyboru pakietów rozpocznie się procedura pobierania plików i instalacja, co przy modemie DSL może zająć kilkadziesiąt minut.



Rys. 6. Okno wyboru trybu instalacji

Po zakończeniu instalacji, musimy jeszcze upewnić się, że katalog *cygwin\bin* został dodany do zmiennej systemowej *PATH*. Można to sprawdzić na przykład poprzez uruchomienie wiersza polecenia i wpisania gcc. Jeżeli nie pojawi się odpowiedz *gcc: no input files* wówczas do zmiennej systemowej *PATH* musimy wpisać ścieżkę do katalogu *cygwin\bin*.

Po upewnieniu się, że środowisko *Cygwin* zostało prawidłowo zainstalowane przystępujemy do instalacji kompilatora *gcc*. Dla Windowsa można go pobrać ze strony *http://yagarto.de*, gdzie znajduje się binarna wersja kompilatora.

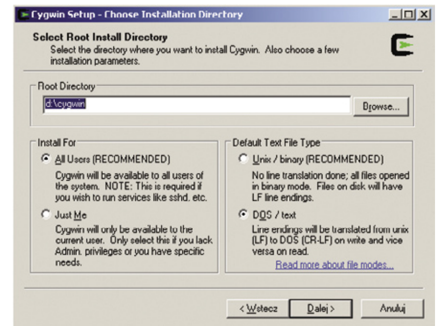
Po ściągnięciu tego pliku przystępujemy do instalacji kompilatora, która przebiega w taki sam sposób jak instalacja większości programów dla Windows. W przypadku linuxa kompilator możemy pobrać ze strony *http://bryndza.ep.com.pl/index.php?dz=rozne&id=linarm*, gdzie został umieszczony kompilator *gcc* w wersjach na systemy 32– (*i386*) oraz 64–bitowe (*amd64*).

Po ściągnięciu pakietu kompilatora i narzędzi należy je rozpakować w katalogu głównym za pomocą polecenia *tar xvfz arm–toolchain–*.tar.gz –C /*. Pliki zostaną rozpakowane do katalogu lokalnego */usr/local* dlatego po rozpakowaniu plików należy dopisać ścieżkę *PATH* dostępu do plików wykonywalnych kompilatora, na przykład na końcu pliku */etc/profile* dopisując w edytorze *export PATH=$PATH:/usr/local/arm–elf–gcc/bin*.

Do zakończenia konfiguracji całego środowiska pozostała nam jeszcze do instalacji oprogramowania *OpenOCD* służącego do obsługi JTAG–a. Dla Windows program ten możemy pobrać ze strony *http://www.yagarto.de* (w zakładce *Open On–Chip–Debugger*), a następnie uruchomić instalator.

Budowa projektu

Tworząc oprogramowanie z wykorzystaniem kompilatora *gcc* na platformę AVR, musimy stworzyć jedynie plik *makefile* opisujący przebieg procesu kompilacji i zależności pomiędzy plikami. Pozostałe elementy niezbędne do prawidłowego przebiegu kompilacji zawarte są w samym pakiecie *WinAVR* i podstawie opcji *–m* określającej typ procesora są one automatycznie dołączane do projektu. W przypadku ARM–ów sytuacja jest zdecydowanie bardziej skomplikowana, ponieważ istnieje wiele odmian mikrokontrolerów z rdzeniem ARM, które są produkowane przez różne firmy. Jedyną ich cechą wspólną jest rdzeń ARM, natomiast sposób inicjalizacji, peryferia, mapa pamięci mogą być zupełnie odmienne, więc nie jest możliwe stworzenie uniwersalnego pliku startowego dla wszystkich typów mikrokontrolerów. Ponadto kompilator *gcc* i biblioteka standardowa *libc* z kompilatora GCC powstała głównie z myślą o Linuxie, w którym są wykorzystywane wywołania systemowe i pliki wykonywalne ELF, niezależne od architektury. Z tych właśnie powodów poszczególne pliki startowe niezbędne do uruchomienia programu działające na mikrokontrolerach bez systemu operacyjnego musimy stworzyć sami. Do prawidłowego działania programu niezbędne będą następujące pliki:



Rys. 7. Ustawienia konfiguracyjne środowiska Cygwin

– Skrypt linkera zawierający informację o rozmiarze i adresach pamięci fizycznej układu. Skrypt musi ponadto zawierać informacje o rozmieszczeniu poszczególnych sekcji danych i kodu w pamięci.
– Plik startowy zawierający kod wykonywany zaraz po uruchomieniu mikrokontrolera, do zadań którego należy inicjalizacja urządzeń peryferyjnych niezbędnych do prawidłowej pracy (pętla PLL, kontroler pamięci Flash). Kod startowy musi także zainicjalizować poszczególne sekcje kodu w pamięci mikrokontrolera zgodnie ze standardem ANSI C/C++, wywołać konstruktory obiektów globalnych oraz oddać sterowanie do funkcji *main*. Wszystkie wspomniane tutaj czynności w normalnym przypadku wykonywane są przez system operacyjny.

– Plik reguł *makefile*, określający zależności pomiędzy plikami projektu, sposób budowania projektu, narzędzia użyte do kompilacji oraz opcje kompilatora.
– Opcjonalny plik *.gdbinit* zawierający polecenia inicjalizujące dla debugera *gdb*, który będzie przydatny podczas debugowania projektu.

Pisząc nowy projekt nie musimy tworzyć za każdym razem od nowa plików startowych, wystarczy stworzyć jednorazowo projekt wzorcowy, a następie wykorzystywać go jako bazę do tworzenia innych projektów.

W **tab. 2** przedstawiono wszystkie pliki, przykładowego projektu wraz z krótkim opisem funkcji jakie one pełnią.

Tab. 2. Przykładowe pliki projektu dla GNUARM

Nazwa pliku	Przeznaczenie
crt0_str912.S	Plik startowy inicjalizujący układy peryferyjne mikrokontrolerów STR91x oraz inicjalizujący pamięć RAM zgodnie z wymaganiami standardu ANSI C/C++
str912–rom.ld	Plik linkera określający, w sposób rozmieszczenia poszczególne segmentów w pamięci. Jest on napisany tak, aby program i dane stałe były umieszczone w pamięci FLASH, natomiast zmienne umieszczone zostaną w pamięci RAM.
Makefile	Plik konfiguracyjny dla narzędzia make służącego do określenia zależności pomiędzy plikami projektu
sysclock.c	Plik źródłowy przykładowego projektu

Automatyzacja kompilacji projektu za pomocą GNU Make

Budowania aplikacji składa się z trzech etapów. Najpierw wszystkie pliki źródłowe są kompilowane z postaci źródłowej do plików obiektowych **.o*, które zawierają skompilowany kod wynikowy, pozbawiony bezwzględnych odwołań do pamięci. Następnie tak powstałe pliki wynikowe są przetwarzane przez linker, który na podstawie skryptu łączy razem wszystkie pliki obiektowe oraz umieszcza je w odpowiednich obszarach pamięci. W wyniku procesu łączenia (linkowania) powstaje plik wykonywalny w formacie ELF, a następnie plik w postaci binarnej. W przypadku skompilowanych projektów zawierających wiele plików źródłowych, kompilacja projektu jest złożona i wymaga użycia specjalnego narzędzia zarządzającego procesem kompilacji. Takim narzędziem jest *GNU make*, który na podstawie pliku opisującego proces budowania projektu *Makefile* stwierdza, które pliki wymagają kompilacji. Zaoszczędza to wiele czasu przy tworzeniu programu, ponieważ w wyniku zmiany pliku źródłowego kompilowane są tylko te pliki, które są zależne od zmienionego pliku, a nie cały projekt. Tworząc plik *makefile* musimy zbudować odpowiednie zależności i reguły określające, w jaki sposób powstać ma plik wynikowy.

Reguły są tworzone w następujący sposób:

plik_docelowy:pliki_wejsciowy_1 plik_wejsciowy_2 ... plik_wejsciowy_n
  komenda
  ....
  ....

Wywołanie programu *make* bez dodatkowych parametrów powoduje wykonywanie czynności według reguły *all*, a w przypadku jej braku pierwszej napotkanej reguły. Jako parametr polecenia make możemy podać nazwę reguły jaka ma zostać wykonana, na przykład wpisanie *make clean* powoduje rozpoczęcie wykonywania reguły *clean*. Pliki *makefile* dają także możliwość używania zmiennych ułatwiających tworzenie skryptów. Zmienną deklarujemy w sposób następujący: *NAZWA_ZMIENNEJ = wartosc*, a następnie tak zadeklarowaną zmienną możemy wykorzystać w wielu miejscach pliku. Aby podstawić w miejsce zmiennej jej wartość, poprzedzamy ją znakiem $ na przykład *$(NAZWA_ZMIENNEJ)*. Niektóre nazwy zmiennych mają szczególne znaczenie i są wykorzystywane przez reguły domyślne. Do najważniejszych zmiennych należą:

– CC jest zmienną określającą nazwę kompilatora języka C, wykorzystywaną przez regułę domyślną tworzącą pliki wynikowe z plików **.c*,
– CFLAGS jest zmienną określającą opcje kompilatora C, które będą przekazane do kompilatora podczas tworzenia plików wynikowych,
– CXX jest zmienną określającą nazwę kompilatora języka C++ wykorzystywaną przez regułę domyślną tworzącą pliki wynikowe z plików **.cpp*,
– CXXFLAGS jest zmienną określającą opcje kompilatora C++, które będą przekazane do kompilatora podczas tworzenia plików wynikowych,
– ASFLAGS jest zmienną określającą dodatkowe opcje przekazywane do kompilatora podczas tworzenia plików wynikowych z plików asemblera,
– LDFLAGS jest zmienną zawierającą dodatkowe opcje dla programu linkującego przekazane podczas procesu linkowania.

Istnieją także specjalne rodzaje zmiennych, które nazwano zmiennymi automatycznymi, które oszczędzają użytkownikowi wielokrotnego wpisywania nazw plików przekazywanych do polecenia. Do najczęściej używanych zmiennych automatycznych należą:

$@ – jest zmienną zwracającą nazwę pliku docelowego, jaki ma zostać stworzony (reguły),

$^ – jest zmienną zwracającą nazwę wszystkich plików wymaganych do stworzenia pliku docelowego,

$< – jest zmienną zawierająca nazwę pierwszego pliku wymaganego do stworzenia pliku docelowego.

Dla przykładu, jeżeli chcemy stworzyć regułę opisującą sposób tworzenia pliku wykonywalnego *ledtst.elf*, który powstaje w wyniku linkowania plików *led.o* i *crt0_str912.S*, posługując się metodą tworzenia reguł bez wykorzystania zmiennych automatycznych musimy napisać *ledtst.elf: led.o crt0_str912.o*
  gcc led.o crt0_str912.o –o ledtst.elf

Jak widać, nazwy plików musieliśmy powtórzyć dwukrotnie, raz dla określenia reguł tworzenia pliku, a drugi raz jako nazwy przekazane kompilatorowi.

Reguła z wykorzystaniem zmiennych automatycznych wygląda następująco:

ledtst.elf: led.o crt0_str912.o
  gcc $^ –o $@

Nie musimy więc dwukrotnie wpisywać nazw plików, ponieważ zostaną one rozwinięte automatycznie przez narzędzie *make*. Po zapoznaniu się z podstawowymi wiadomościami na temat narzędzia *make* stworzymy przykładowy plik *makefile* automatyzujący projekt kompilacji najprostszego projektu złożonego z jednego pliku źródłowego *led.c* oraz plików startowych.

Budowanie projektu rozpoczyna się od skompilowania plików *led.c* oraz plików nagłówkowych *h*, w wyniku czego powstaje plik obiektowy *led.o*. Następnie kompilujemy asemblerowy plik *crt0_str912.S*, z którego

List. 1.

```
#Programy uzywane do kompilacji
CXX    = arm-elf-c++
CP     = arm-elf-objcopy
CC     = arm-elf-gcc

#Opcje kompilatora
FLAGS = -Os -mcpu=arm966-s -Wall -gstabs

#Opcje specyficzne dla C++
CXXFLAGS = $(FLAGS)

#Opcje specyficzne dla C
CFLAGS = $(FLAGS) --std-gnu99


#Opcje specyficzne dla assemblera
ASFLAGS = $(FLAGS) -Wa,-mapcs-32

#Opcje specyficzne dla linkera
LDFLAGS = $(FLAGS) -nostartfiles -Tstr912-rom.ld

#Format pliku wyjsciowego ihex lub binary
CPFLAGS = -O binary -S

#Nazwa docelowego pliku wyjsciowego
all: ledtest.bin

#Tworzenie projektu
ledtest.bin: ledtest.elf
  $(CP) $(CPFLAGS) $^ $@

ledtest.elf: led.o crt0_str912.o | str912-rom.ld
  $(CC) $(LDFLAGS) $^ -o $@

led.o: led.c

crt0_str912.o: crt0_str912.S

#Czyszczenie projektu
clean:
  rm -f *.o
  rm -f ledtest.elf
  rm -rf ledtest.bin
```

List. 2.

```
#Tworzenie pliku wynikowego ELF
przyklad.elf:  3.o 2.o 1.o asm.o | str912-rom.ld
  $(CXX) $(LDFLAGS) $^ -o $@

#Tworzenie plikow obiektowych (kompilacja)
boot.o:boot.S
3.o: 3.S
asm.o: asm.S
1.o: 1.cpp naglowek.h
2.o: 2.cpp naglowek.h
```

powstaje plik obiektowy *boot.o*. W kolejnym etapie, zwanym linkowaniem, z plików obiektowych *led.o* i *boot.S* na podstawie skryptu linkera *str912–rom.ld* powstaje wykonywalny plik wynikowy *ledtst.elf*. Ostatnim etapem kompilacji projektu jest konwersja pliku *ledtst.elf* z formatu *ELF* do pliku *ledtst.bin*, który może posłużyć do zaprogramowania pamięci Flash mikrokontrolera.

Na **list. 1** przedstawiono plik *makefile* automatyzujący proces kompilacji tego projektu.

Tworząc pliki *makefile* musimy pamiętać, aby poprawnie zapisać wszystkie zależności, gdyż w przeciwnym przypadku program wynikowy może się poprawnie nie kompilować. Tworzenie plików *makefile* dla bardziej rozbudowanych projektów nie jest dużo bardziej skomplikowane.

W przypadku projektu pokazanego na **list. 2** reguły są następujące: aby powstał *przykład.elf* muszą istnieć pliki *1.o*, *2.o* i *3.o* oraz *asm.o*. Plik *przyklad.elf* powstaje poprzez wywołanie linkera, którego nazwa jest określona przez zmienną $(CXX). Następnie plik *boot.o* jest tworzony w wyniku kompilacji pliku *boot.S* z wykorzystaniem polecenia zdefiniowanego przez domyślną regułę przypisaną do nazwy pliku. Podobnie plik *3.o* powstaje poprzez kompilację z pliku *3.S*. Plik *1.o* jest kompilowany na podstawie pliku *1.cpp* oraz *nagłówek.h*, natomiast plik *2.o* powstaje z pliku *2.cpp* oraz *naglowek.h*.

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

R E K L A M A

x3 CD
Wydanie specjalne 1/2008
ELEKTRONIKA PRAKTYCZNA plus
Programy narzędziowe | Przykładowe aplikacje | Dokumentacje | Katalogi | Oferty | Prezentacje
Międzynarodowy magazyn elektroników konstruktorów
Mini PLC
MicroSmart Pentra – PLC otwarty na przyszłość
Kinco – generacja jutra
Nowoczesne struktury i pamięci danych ze „świata" IT w przemysłowych PLC
CP1L – nowy miniaturowy sterownik firmy Omron
Sterowniki PLC do małych systemów sterowania
Komunikacja sterownika Simatic S7-200 z wykorzystaniem protokołu Modbus RTU
System Bus Terminal firmy Beckhoff
Panel operatorski HMI Vision350
MiniPLC – przegląd oferty rynkowej
Need – mamy coś do powiedzenia
Cena 26,00 zł (w tym 0% VAT) Nakład: 14 000 egz.

ELEKTRONIKA PRAKTYCZNA plus

Oto piąte już wydanie Elektroniki Praktycznej Plus - nieregularnika, którego każdy numer poświęcony jest w całości konkretnej tematyce (poprzednie dotyczyły mikrokontrolerów ARM, technologii M2M, wyświetlaczy i paneli HMI oraz diod PowerLED). EP+ MiniPLC jest poświęcona urządzeniom automatyki - sterownikom logicznym. Przedstawiamy możliwości małych i średnich urządzeń tego typu, dostępnych na rynku, osprzęt ułatwiający ich aplikowanie, oprogramowanie narzędziowe, a także stosowane w praktyce moduły rozszerzeń. W magazynie MiniPLC znajdują się trzy płyty CD, na których - poza materiałami technicznymi, związanymi z tematyką wydania, przygotowanymi przez redakcję - są także materiały nadesłane przez firmy, np. oprogramowanie, katalogi, filmy, prezentacje.

JUŻ W SPRZEDAŻY

www.sklep.avt.pl