Wprowadzenie do Linux'a embedded

Współczesne oprogramowanie urządzeń powszechnego użytku staje się coraz bardziej skomplikowane. Nowoczesne urządzenia mają graficzny interfejs użytkownika, najczęściej dotykowy, coraz bardziej skomplikowane interfejsy komunikacyjne, takie jak WiFi, USB czy Ethernet. Dostępne funkcje oprogramowania stwarzają nam takie możliwości, o których kilka lat temu można było tylko marzyć. Urządzenia stają się coraz bardziej przyjazne dla użytkownika i coraz bardziej kolorowe. Kto by przypuszczał że kupując współczesny telefon np. z systemem Android, kupujemy urządzenie o wydajności komputera biurkowego sprzed kilkunastu lat, który możemy zmieścić w kieszeni? Niestety coraz większe skomplikowanie oprogramowania stawia coraz to nowe wyzwania programistom i konstruktorom, i dlatego Ci coraz chętniej sięgają po systemy operacyjne. Jednym z nich jest Linux embedded.

O ile niegdyś w ramach małej firmy napisanie od postaw skomplikowanego oprogramowania było całkiem realne, o tyle dziś już coraz trudniej temu podołać. Konieczność obsługi kolorowych wyświetlaczy LCD, paneli dotykowych i skomplikowanych interfejsów powoduje, że obsługa tych wszystkich peryferiów wykonywana na „piechotę" poprzez dostęp do rejestrów, bez żadnych dodatkowych bibliotek, jest w zasadzie nie do wykonania w sensownym czasie. W przypadku, gdy projekt jest stosunkowo prosty, możemy skorzystać z dodatkowych bibliotek coraz częściej dostarczanych przez producentów mikrokontrolerów. W przypadku większych projektów takie podejście również okazuje się niewystarczające. Brak jednolitego interfejsu w świecie mikrokontrolerów powoduje, że większość standardowego oprogramowania (na przykład serwer www z szyfrowaniem SSL) będziemy musieli utworzyć samodzielnie. Również brak ochrony pamięci pomiędzy zadaniami/procesami powoduje, że nawet mały błąd w kodzie powoduje zawieszenie się całego urządzenia, a o błędy w dużym i skomplikowanym programie nie jest trudno.

Rozwiązaniem wyżej wspomnianych problemów jest użycie systemu operacyjnego znanego z większych komputerów biurkowych. Dzięki temu zyskujemy mechanizm ochrony pamięci pomiędzy procesami, gdzie błędnie działająca aplikacja nie może zawiesić całego systemu. Dostajemy również dostęp do wielu standardowych bibliotek i programów, przez co możemy zaoszczędzić wiele czasu na tworzenie czegoś, co zostało wymyślone kiedyś. Obecnie najlepszym systemem operacyjnym przeznaczonym do zastosowania w urządzeniach wbudowanych jest system Linux, z uwagi na jego skalowalność, otwarty kod źródłowy, brak konieczności zakupu kosztownych licencji oraz dostępność na w zasadzie każdą platformę sprzętową. Linux potrafi działać na bardzo różnych mikrokomputerach, począwszy od małych systemów mikroprocesorowych, poprzez PC, do komputerów typu *mainframe*. Dzięki temu zdobywa on coraz większą popularność w urządzeniach wbudowanych i jest już używany w 25% telefonów komórkowych (Android).

Niestety, zastosowanie Linux'a wymaga użycia dużego procesora z jednostką zarządzania pamięcią MMU, oraz dużą pamięcią RAM. Z uwagi na znaczny spadek cen, komponenty niezbędne do budowy prostego systemu mikroprocesorowego dla systemu Linux możemy nabyć w cenie dużego mikrokontrolera jednoukładowego.

Tabela 1. Minimalne wymagania systemowe Linuxa z jądrem 2.6.x

Procesor: 32 – bitowy (ARM, MIPS, PPC)
Taktowanie procesora: 60MHz
Pamięć RAM: 4MB
Pamięć masowa (FLASH/Karta SD): 2MB

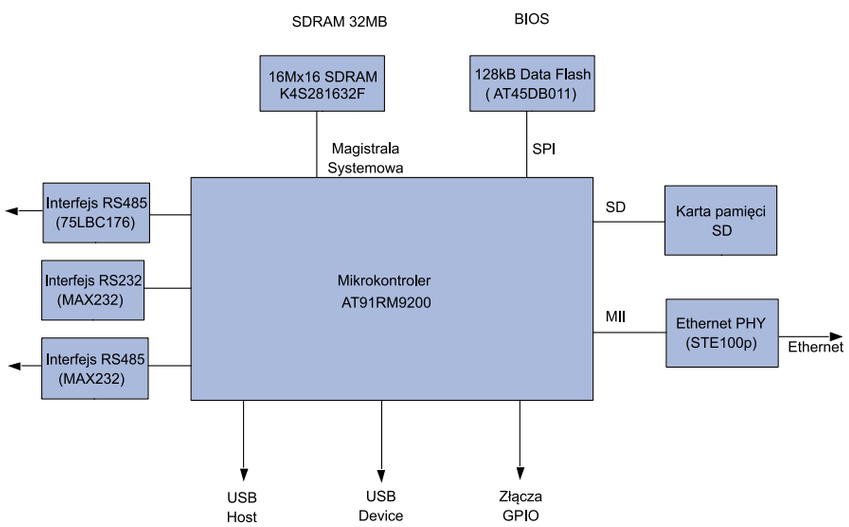
Tabela 2. Zalecane wymagania systemowe Linuxa z jądrem 2.6.x

Procesor: 32 – bitowy (ARM, MIPS, PPC)
Taktowanie procesora: 200MHz
Pamięć RAM: 32MB
Pamięć masowa (FLASH/Karta SD): 32MB

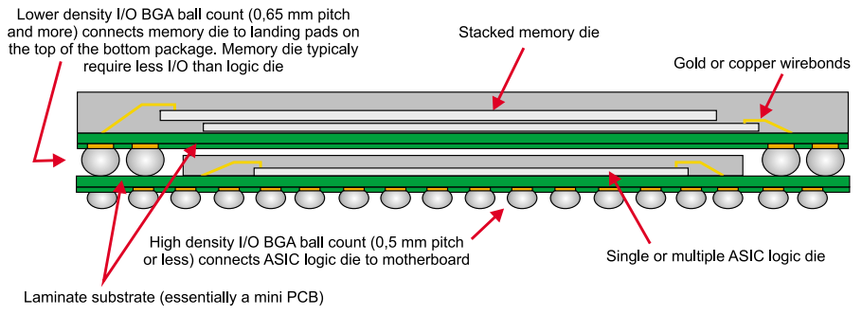
Minimalny system mikroprocesorowy potrzebny do uruchomienia Linuksa. Wymagania systemowe

W **tabeli 1** umieszczono minimalne wymagania systemowe które powinien spełniać system mikroprocesorowy, niezbędne do uruchomienia systemu Linux z jądrem w wersji 2.6.x, dla systemów wbudowanych.

Są wymagania minimalne, w przypadku których ciężko mówić o komfortowej pracy. Aby zapewnić w miarę komfortową pracę należy zbudować system mikroprocesorowy o parametrach co najmniej takich, jak wymienione w **tabeli 2**.



Rysunek 1. Schemat blokowy przykładowego, minimalnego systemu mikroprocesorowego dla Linux działającego na AT91RM9200



Rysunek 2. Pamięć RAM zamontowana mikrokontrolerze (źródło: *www.wikipedia.org*).

Kluczowym elementem do prawidłowego działania systemu, jest dostępność odpowiedniej ilości pamięci RAM, która musi być znacznie większa, niż w przypadku aplikacji dla mikrokontrolerów jednoukładowych. Podobnie wygląda sprawa z pamięcią, w której jest przechowywany *firmware* urządzenia. Samo jądro linuksa z włączoną obsługą sieci zajmuje około 1 MB. Dużo mniej krytycznym aspektem jest częstotliwość pracy procesora, i nawet przy taktowaniu rzędu 60 MHz, system będzie pracował w miarę wydajnie. Przy wyborze odpowiedniego procesora kluczowym znaczeniem jest jego dostępność oraz powszechność. Im bardziej układ jest popularny i używany przez większą liczbę osób , tym lepiej dopracowany i stabilny jest kod jądra przeznaczonego dla niego Linux'a. Również kluczowe znaczenie ma tutaj cena układu, a wiadomo – im układ bardziej powszechny, tym jego produkcja jest tańsza. Ponieważ obecnie na rynku systemów wbudowanych królują mikrokontrolery z rdzeniem ARM, w dalszej części tekstu będziemy skupiać się wyłącznie na tej architekturze. Na **rysunku 1** przedstawiono przykładowy, minimalny system mikroprocesorowy z AT91RM9200, na którym możemy w miarę komfortowo uruchomić system linux:

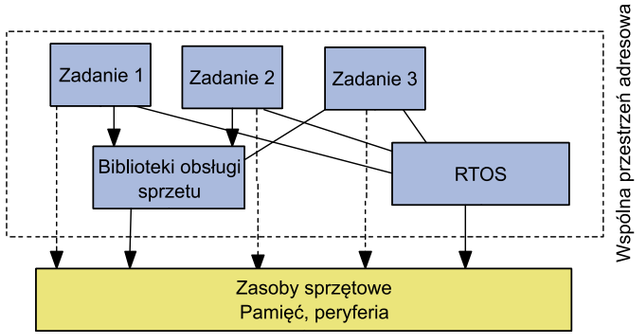
Sercem układu jest dość leciwy mikrokontroler AT91RM9200, z rdzeniem ARM920T do którego dołączono kluczowe elementy, czyli pamięć SDRAM o wielkości 32MB. Pamięć masową dla systemu stanowi pamięć Dataflash 128KB, w której przechowywany jest program ładujący (bootloader), oraz karta SD na której przechowywany jest właściwy system linux. Zamiast pamięci Dataflash i karty SDRAM w nowszych układach np. 9260, SAM-G20 możemy użyć pojedynczej pamięci typu NAND-FLASH. Pozostałe elementy stanowią interfejsy komunikacyjne i są w opcjonalne. Jak więc widzimy, w stosunku do klasycznego rozwiązania układowego jedyną różnicą jest konieczność dołączenia zewnętrznych pamięci do układu. Koszt procesora, oraz dodatkowych pamięci (SDRAM jest bardzo tani) , jest porównywalny do większych mikrokontrolerów jedno-układowych a możliwości są nieporównanie większe. Niestety problemem jest tutaj konieczność wyprowadzenia całej magistrali , co znacząco komplikuje konstrukcję. Jednak nadzieją są tutaj rozwiązania typu „Package on Package", umożliwiające umieszczenie pamięci nad procesorem, i złożenie z nich jakby jednego układu, np. rozwiązanie OMAP-DM510, zawierającą pamięć DDR umieszczoną na procesorze (**rysunek 2**).

Ochrona pamięci systemowej i co z tego wynika, czyli różnice pomiędzy pisaniem aplikacji dla mikrokontrolera a aplikacją przeznaczoną dla Linux'a

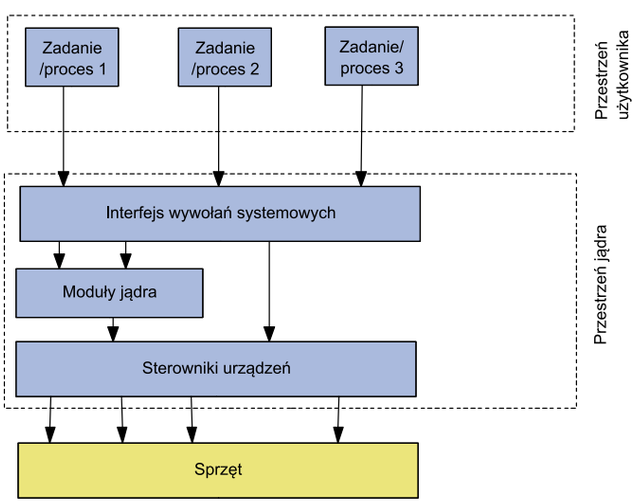
Oprogramowanie dla mikrokontrolerów jednoukładowych jest najczęściej pisane na dwa sposoby: bez systemu operacyjnego, gdzie poszczególne zadania kodowane są za pomocą maszyn stanów w sposób jawny lub z wykorzystaniem małych systemów RTOS, takich jak FreeRTOS, czy ISIX, które zapewniają nam wykonywanie zadań, bez konieczności tworzenia podatnych na popełnienie błędu maszyn stanu. Na **rysunku 3** przedstawiono model zależności pomiędzy sprzętem a przykładową aplikacją wielozadaniową dla mikrokontrolera jednoukładowego.

Istnieją również wersje linuksa, mogące pracować bez MMU np. na procesorach ARM7TDMI, czy CORTEX-M3, jednak są one dużo mniej wydajne ponieważ MMU jest emulowane, oraz nie zapewniają odpowiedniego poziomu bezpieczeństwa. Więc nie możemy go traktować jako pełnoprawną wersję systemu Linux.

Aplikacja ma trzy zadania realizujące różne funkcje. System RTOS jest odpowiedzialny za przełączanie zadań, synchronizację międzyzadaniową, która jest realizowana za pomocą semaforów i kolejek. Natomiast obsługa sprzętu realizowana jest za pomocą dodatkowej biblioteki dostarczonej przez producenta mikrokontrolera. Aplikacje, RTOS oraz biblioteki obsługi sprzętu działają w ramach jednej wspólnej przestrzeni adresowej, która jest określana podczas etapu linkowania. Każdy element aplikacji, czyli poszczególne zadania, RTOS, biblioteki obsługi sprzętu, mają dostęp do całej przestrzeni adresowej oraz systemu przerwań. W takim modelu jedno błędnie działające zadanie, które np. nadpisało pamięć w drugim zadaniu, jest w stanie spowodować zawieszenie działania całego programu. Jedynym ratunkiem jest wówczas zadziałanie sprzętowego układu watchdog. O ile w prostych programach możliwość wyszukania tego typu błędów jest w miarę łatwa, to wykrycie ich w przypadku rozbudowanych aplikacji jest dość problematyczne. Do niewątpliwych zalet modelu programowego ze wspólną przestrzenią adresową



Rysunek 3. Model zależności pomiędzy sprzętem a przykładową aplikacją wielozadaniową dla mikrokontrolera jednoukładowego



Rysunek 4. Model programowy dla projektów realizowanych z użyciem systemu Linux

możemy zaliczyć prostotę dostępu do zasobów. W zależności od potrzeb, z każdego miejsca aplikacji możemy bezpośrednio obsługiwać sprzęt poprzez bezpośredni dostęp do rejestrów układów peryferyjnych, z poziomu zadania, biblioteki, czy samego RTOS-a. Również przekazywanie danych pomiędzy zdaniami jest łatwe, ponieważ każde zadanie ma dostęp do tego samego obszaru pamięci, który może być zapisywany lub odczytywany. Podobnie wygląda sprawa z systemem przerwań, choć najczęściej jest ona realizowana przez oddzielny zestaw funkcji OS-a.

Na **rysunku 4** zaprezentowano model programowy dla projektów realizowanych z wykorzystaniem systemu Linux. Dla systemu Linux działającego na procesorze z jednostką MMU, jest realizowana wzajemna ochrona zasobów jądra i przydzielonych poszczególnym procesom (zadaniom). Każdy proces ma niezależną, wirtualną przestrzeń adresową, co zapewnia wzajemną ochronę aplikacji oraz samego jądra systemu przed błędnym działaniem innych procesów, jak również umożliwia optymalne wykorzystanie pamięci poprzez mapowanie adresów wirtualnych na adresy fizyczne za pomocą mechanizmu stronicowania. Każde odwołanie do nieprawidłowego adresu wirtualnego powoduje zatrzymanie procesu poprzez sygnał *SEGFAULT* informujący o błędnym odwołaniu do pamięci. Z poziomu aplikacji nie ma możliwości bezpośredniego odwoływania się do sprzętu, a komunikacja z nim musi odbywać się za pomocą wywołań systemowych jądra, które przekazują żądania obsługi sprzętu do sterowników urządzeń. Wywołanie systemowe (w przypadku architektury ARM instrukcja SWI) powoduje przejście aplikacji z trybu użytkownika do trybu jądra i wykonanie określonych czynności przez jądro na rzecz procesu (mówimy wówczas że mamy do czynienia z wykonywaniem kodu jądra w kontekście procesu). Natomiast samo jądro wraz ze wszystkimi sterownikami urządzeń działa w ramach wspólnej pojedynczej przestrzeni adresowej. Odizolowanie aplikacji od urządzeń zapewnia również dostęp do danego urządzenia przez wiele procesów oraz chroni przed błędnym działaniem poszczególnych aplikacji.

Dostęp do urządzeń z procesu użytkownika jest realizowany poprzez otwarcie pliku urządzenia a następnie pisanie i czytanie za pomocą standardowych wywołań systemowych *read*, *write*, *select* itp. Wywołania systemowe w aplikacji realizowane są za pośrednictwem standardowej biblioteki *libc* (b*iblioteka języka C*), z której wywoływane są funkcje realizujące wywołania systemowe jądra. Przygotowanie projektu z wykorzystaniem systemu Linux intensywnie wykorzystującego zewnętrzne układy peryferyjne, jest bardziej skomplikowane, niż przy pisaniu aplikacji dla mikrokontrolera jednoukładowego, ponieważ będziemy musieli napisać (lub wykorzystać gotowe) sterowniki dla urządzeń oraz osobno napisać aplikację, która będzie komunikowała się ze sterownikami. Przy pisaniu sterowników musimy zachować szczególną ostrożność, tak jak w przypadku pisania na mikrokontrolery bez MMU, ponieważ sterownik zawsze ma dostęp do wszystkich zasobów, więc głównie od jakości sterowników zależeć będzie stabilność systemu.

Kiedy będziemy potrzebować Linux'a?

Spróbujmy odpowiedzieć na pytanie, w jakich przypadkach będziemy potrzebować systemu mikroprocesorowego z systemem Linux, a kiedy szybsze i tańsze będzie zastosowanie standardowego rozwiązania opartego o zwykły mikrokontroler jednoukładowy i mini system operacyjny np. ISIX.

Systemu mikroprocesorowego z Linukx'em będziemy potrzebować, gdy:

- Istnieje konieczność używania skomplikowanych interfejsów komunikacyjnych WiFi, Bluetooth, Ethernet itp. (Linux ma wbudowane sterowniki dla większości typowych urządzeń sieciowych oraz stosy WiFi, TCP/IP, BT, USB HOST itp.).
- Potrzebujemy skomplikowanego interfejsu użytkownika z dużym, kolorowym wyświetlaczem TFT oraz panelem dotykowym. Korzystając z zewnętrznych bibliotek graficznych, serwera X oraz Qt czy GTK, możemy projektować bardzo skomplikowane interfejsy graficzne niewielkim nakładem środków.
- Potrzebujemy aplikacji multimedialnej wyświetlającej filmy, odtwarzającej pliki MP3 itp. Możemy wykorzystać bardzo bogate biblioteki kodeków, np. *ffmpeg* itp.
- Jest wymagana komunikacja z wykorzystaniem szyfrowania danych SSL.
- Potrzebujemy wykonać urządzenie z interfejsem sterowanym przez przeglądarkę WWW z bezpiecznym protokołem HTTPS oraz dużą liczbą dynamicznie generowanych stron. Możemy zainstalować standardowy serwer www+PHP i pisać aplikację, tak jak w przypadku pracy z dużym komputerem PC.
- Potrzebujemy wykonać bardziej skomplikowane zadanie, a znaleźliśmy biblioteki OpenSource pozwalające w łatwy sposób osiągnąć zamierzony efekt.
- Potrzebujemy urządzenia sterującego niezależnymi procesami, z gwarancją prawidłowego działania nawet w przypadku uszkodzenia jednego z procesów.

Lepszym rozwiązaniem będzie wykorzystanie standardowego mikrokontrolera jednoukładowego, oraz ewentualnie systemu RTOS takiego jak ISIX gdy:

- Wykonujemy aplikację, która realizuje prosty algorytm niepotrzebujący dużych zasobów, skomplikowanych obliczeń ani dodatkowych bibliotek zewnętrznych i niewymagający dużego nakładu pracy na napisanie samej aplikacji. W Linux'ie napisanie samego sterownika urządzenia może się okazać zadaniem dużo bardziej skomplikowanym, niż dodatkowy nakład potrzebny na napisanie aplikacji samodzielnie.
- Wykonujemy urządzenie produkowane masowo o niezbyt skomplikowanym działaniu, gdzie koszt dodatkowej pracy programistycznej jest niewspółmiernie mały do kosztów całego projektu i konieczności minimalizacji kosztów jednostkowych urządzenia. Takim przykładem może być np. sterownik z interfejsem ETHERNET, gdzie napisanie aplikacji pod Linux'em jest bardzo proste, natomiast w przypadku mikrokontrolera jednoukładowego jest bardziej skomplikowane, ale wykonalne w sensownym czasie np. z użyciem bibliotek udostępnianych przez producentów.

Jak więc widzimy użycie systemu mikroprocesorowego z systemem Linux nie jest lekarstwem na wszystko. W przypadku prostych układów sterujących typu załącz/wyłącz/steruj, lepszym rozwiązaniem będzie wykorzystanie klasycznego rozwiązania opartego o mikrokontroler jednoukładowy.

Krótki kurs pisania aplikacji dla Linux'a.

W literaturze, Internecie oraz na wykładach prowadzonych na wyższych uczelniach, możemy znaleźć wiele informacji na temat programowania w systemie Linux. Kursy te najczęściej są poświęcone ogólnym zagadnieniom dotyczącym programowania systemowego, najczęściej z wykorzystaniem komputerów PC. Niestety, brakuje literatury poświęconej Linux'owi w zastosowaniach embedded. W tym kursie postaram się wypełnić tę lukę pokazując, w jaki sposób do przykładowej płytki z Linux'em dołączyć standardowe układy typu wyświetlacz LCD czy klawiatura oraz w jaki sposób odwołać się do nich za pomocą aplikacji systemu operacyjnego. Jako platformę testową wybrano zestaw ewaluacyjny BF210 (*http://www.boff.pl/?shop=1&p_id=5*) zawierający mikrokontroler AT91RM9200, dla którego będą prezentowane gotowe obrazy karty SD pozwalające bez dodatkowych czynności uruchomić prezentowane przykłady.

Lucjan Bryndza, EP
lucjan.bryndza@boff.pl