Rok akademicki 2005/2006

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Praca dyplomowa inżynierska

Radosław F. Wawrzusiak

Zastosowanie systemu Phoenix-RTOS do budowy urządzeń wbudowanych na platformie ARM

Kierownik pracy:
mgr inż. Waldemar Grabski

Ocena.....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Warszawa - 2006

Specjalność: **Inżyniera Systemów Informatycznych**

Data urodzenia: **10-07-1983**

Data rozpoczęcia studiów: **1-10-2002**

Życiorys

Urodzony dnia 10 lipca 1983 w Stalowej Woli, województwo Podkarpackie. W latach 1990-1996 uczęszczał do Państwowej Podstawowej Szkoły Muzycznej w Stalowej Woli, a w latach 1996-1998 do Szkoły Podstawowej nr 6 im. Władysława Broniewskiego w Stalowej Woli. Szkoły podstawowe ukończył z wyróżnieniem. W roku 2002 ukończył z wyróżnieniem Liceum Ogólnokształcące im. Komisji Edukacji Narodowej w Stalowej Woli, w klasie o profilu matematyczno-fizycznym. Od października 2002 student Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej, członek koła naukowego *Wolne Oprogramowanie na Politechnice Warszawskiej*.

.....

Podpis studenta

Egzamin Dyplomowy

Złożył egzamin dyplomy w dniu.....

z wynikiem.....

Ogólny wynik studiów.....

Dodatkowe wnioski i uwagi Komisji.....

.....

.....

Streszczenie

Przedmiotem pracy było przygotowanie wersji systemu operacyjnego Phoenix-RTOS na platformę ARM. W pracy przedstawiono opracowanie dotyczące architektury ARM. Procesory oparte na tej architekturze to klasyczne, 32-bitowe procesory RISC. Omówiony została także przykładowa implementacja architektury ARMv4T w postaci mikrokontrolera Philips LPC2148 wyposażonego w rdzeń ARM7TDMI. Dalszej części pracy znajduje sie krótka charakterystyka systemu Phoenix-RTOS. Jest to system operacyjny czasu rzeczywistego, którego prototyp powstał na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej w 2001 roku w ramach pracy magisterskiej Pawła Pisarczyka. Zasadnicza część pracy opisuje szczegóły implementacji wersji systemu Phoenix-RTOS przygotowanej, przez autora, do pracy na platformie ARM. Głównym celem projektu było zapoznanie się z platformą ARM oraz zaprojektowanie i uruchomienie jądra systemu Phoenix-RTOS na platformie ARM.

Słowa kluczowe: systemy operacyjne, systemy czasu rzeczywistego, systemy rozproszone, Phoenix-RTOS, architektura ARM, mikrokontrolery, architektura procesora, Philips LPC2000.

Applying Phoenix-RTOS for Making Embedded Systems on ARM

The thesis describes ARM port of Phoenix-RTOS. There is brief reference of ARM architecture, which is classic, 32-bit RISC. Next there is brief description of ARMv4T based processor example – Philips LPC2148. This microcontroller is based on ARM7TDMI core. Following part describes briefly Phoenix-RTOS. This real-time operating system was made by Pawel Pisarczyk in year 2001, as a main part of his M.Sc. Thesis at Faculty of Electronics and Information Technology at Warsaw University of Technology. The main part consits description of Phoenix-RTOS ARM port implementation made by author as a final design project.

Key words: operating systems, real-time systems, distributed systems, Phoenix-RTOS, ARM architecture, micro-controller, processor architecture, Philips LPC2000.

Podziękowania

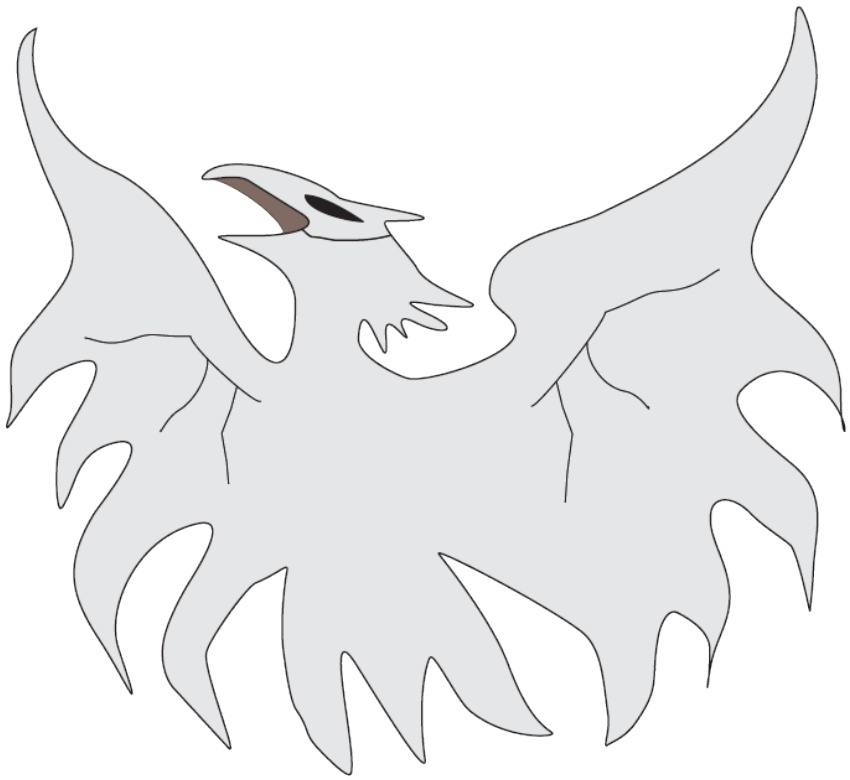
Szczególnie gorące podziękowania należą się autorowi systemu Phoenix-RTOS – Pawłowi Pisarczykowi. Po pierwsze za to, że zaraził mnie ideą tworzenia własnego systemu operacyjnego i wprowadził w świat systemów wbudowanych. Po drugie za jego nieocenioną pomoc w realizacji projektu. Za to, że poświęcił dużo własnego czasu na dyskusje i konsultacje.

Gorąco chciałbym podziękować mojemu opiekunowi – mgr inż. Waldemarowi Grabskiemu za cenne wskazówki i pomoc, a zwłaszcza za odwagę podjęcia się poprowadzenie tej nietypowej pracy.

Pragnę podziękować moim rodzicom za ich miłość, wsparcie i mobilizowanie mnie do systematycznej pracy, a także za umożliwienie mi studiowania na tak znamienitej uczelni jaką jest Politechnika Warszawska.

Podziękowania należą się firmie BTC, a zwłaszcza panu Piotrowi Zbysińskiemu, za przekazanie sprzętu dla celów projektu.

Na koniec chciałbym podziękować wszystkim moim przyjaciołom, a zwłaszcza Łukaszowi Krukowi, Marcinowi Plackowi oraz Tomaszowi Światowcowi, za ich duchowe wsparcie, wiarę w powodzenie projektu oraz wyrozumiałość dla chronicznego braku czasu, zwłaszcza przez ostatnie miesiące tworzenia pracy.



Spis treści

I. Wstęp.

Cele pracy.

Główny celem niniejszej pracy jest wykazanie możliwości rozwoju systemu operacyjnego Phoenix-RTOS. Pierwsza wersja systemu powstała w latach 1999-2001 na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. System został stworzony od podstaw przez Pawła Pisarczyka w ramach pracy magisterskiej[1]. Obecnie dostępna jest nowa, przebudowana i udoskonalona wersja systemu o nazwie Phoenix-RTOS. Wersja ta powstała w 2005 roku i jest sukcesywnie rozwijana i rozbudowywana. System został na pisany na platformę IA-32, lecz w samych założeniach ma on być wieloplatformowy. Wyraźnie wydzielona jest w nim warstwa zależna od sprzętu. Zgodnie z założeniami projektu Phoenix-RTOS, aby przystosować system do nowej platformy powinna wystarczyć modyfikacja tej warstwy.

Praca niniejsza ma na celu sprawdzenie założeń poczynionych przez autora systemu, w zakresie przenaszalności na inne platformy sprzętowe.

Kolejnym istotnym celem pracy jest analiza przydatności platformy ARM do budowy urządzeń wbudowanych. Platforma ta nie jest nowa, ale dopiero niedawno zaczęła wkraczać na rynek szerokim frontem. Mikrokontrolery oparte na rdzeniach o architekturze ARM powoli wypierają z rynku starsze układy 8-bitowe. Praca ta ma na celu przeanalizować budowę i możliwości tych mikrokontrolerów pod kątem wykorzystania ich w systemach wbudowanych wykorzystujących prosty system operacyjny czasu rzeczywistego.

Pobocznym celem pracy jest zapoznanie się z architekturą ARM, jako nowoczesnym i rozwojowym modelem programowym. Obecnie znaleźć można znaleźć nieliczne tylko dobre opracowania dotyczące tej architektury, zwłaszcza w języku polskim. Jest to dość duży problem przy popularyzacji tej, bardzo obiecującej platformy sprzętowej.

W ramach niniejszej pracy prowadzony jest przez autora podprojekt w ramach projektu Phoenix-RTOS[2]. Podprojekt ten ma na celu przygotowanie w pełni funkcjonalnej wersji systemu Phoenix-RTOS na platformę ARM. Postęp prac nad systemem można śledzić, na stronach internetowych projektu, pod adresem:

https://devel.phoenix-rtos.org/phoenix-rtos/

Układ pracy.

Praca podzielona jest na dwie zasadnicze części.

Część pierwsza, składająca się z rozdziałów II-IV, jest szczegółową analizą modelu programowego ARM, wraz z dokładną analizą przykładowej implementacji tego modelu programowego, w postaci szeroko stosowanego mikrokontrolera z rodziny Philips LPC2000. W części tej umieszczony jest także krótki opis systemu Phoenix-RTOS.

Część druga, na którą składa się rozdział V, zawiera szczegółowy opis implementacji niskopoziomowych mechanizmów systemu Phoenix-RTOS na platformie ARM oraz raport z testowania. Do pracy dołączone są kody źródłowe programu ładującego arm-plo oraz, działającej na platformie ARM, wersji jądra systemu Phoenix-RTOS.

1 Patrz [PM]
2 Strona projektu: [PH]

II. Architektura ARM

1. Architektura RISC

W początkowej fazie rozwoju procesorów dążono do skrócenia i uproszczenia kodu pisanego w języku maszynowym, ponieważ głównie w taki sposób tworzono oprogramowanie systemowe, a kompilatory języków wysokiego poziomu były niedoskonałe. Procesory wyposażano w rozbudowany zestaw instrukcji, służących do wykonywania bardzo skomplikowanych operacji. Interpretacja instrukcji procesora była implementowana jako mikroprogram. Każda instrukcja wykonywana była sekwencyjnie w wielu cyklach procesora. Procesor posiadał wiele, nieortogonalnych rozkazów (tzn. wiele z pojedynczych instrukcji procesora mogło zostać zastąpione przez kombinację innych), wiele trybów adresowania i małą liczbę specjalizowanych rejestrów. Architektura procesora o takich cechach nazwana została terminem CISC (Complex Instruction Set Computing).

Wraz z rozwojem języków programowania wysokiego poziomu i ich kompilatorów szybko zauważono, że kod wynikowy programów wykorzystuje jedynie mały podzbiór instrukcji procesora. Zaobserwowano, że tranzystory, przeznaczone do realizowania rzadko używanych instrukcji, można wykorzystać do budowy dodatkowych rejestrów lub do zwiększenia rozmiaru pamięci podręcznej. Zaproponowano uproszczenie instrukcji, zmniejszenie liczby trybów adresowania i podział procesora na jednostki wykonawcze, czyli na elementy logiczne, odpowiadające za wykonanie określonych grup instrukcji. Wprowadzono także potokowe wykonanie instrukcji, które polegało na wyodrębnieniu dobrze określonych faz wykonania instrukcji wewnątrz procesora. Dzięki przetwarzaniu potokowemu, sekwencyjne wykonanie pojedynczej instrukcji przestało blokować procesor przez wiele cykli. W pojedynczym cyklu procesor przetwarzał wiele instrukcji w różnych fazach wykonania. Procesory o takiej architekturze zostały określone mianem RISC (Reduced Instruction Set Computing). Za archetyp procesora RISC uważany jest procesor komputera CDC6600, zaprojektowany przez legendarnego twórcę superkomputerów Seymoura Craya.

Dzięki wyeliminowaniu złożonych instrukcji, wyraźnemu rozdzieleniu instrukcji arytmetyczno-logicznych od instrukcji, współpracujących z pamięcią i usystematyzowaniu przetwarzania, można było produkować szybsze procesory przy użyciu mniejszej liczby tranzystorów.

W obecnych czasach różnica pomiędzy mikroarchitekturą procesorów RISC i CISC uległa zatarciu. Ze względu na postępującą miniaturyzację i niewiarygodną gęstość upakowania tranzystorów w pojedynczych układach scalonych, model programowy przestał znacząco wpływać na mikroarchitekturę. W procesorach o modelu programowym CISC takich jak np. Intel Pentium, wprowadzono architekturę potokową, dodano wiele skomplikowanych jednostek wykonawczych i rozbudowano pamięć podręczną. Architektura tego typu została nazwana mianem Post-RISC.

2. Historia[3]

Procesory ARM wyrosły na bazie mikroprocesora MOS6502 firmy MOS Technology. Nazwa ARM wywodzi się od Acorn RISC Machine. Pierwszy procesor ARM został opracowana przez firmę Acorn Computers Ltd. w roku 1985, ale dopiero opracowany rok później ARM2 został wypuszczony na rynek. *"Był to w owym czasie najprostszy, szeroko stosowany mikroprocesor 32-bitowy."*[4] Do jego realizacji użyto 32000 tranzystorów, czyli o połowę mniej, niż do realizacji procesora Motorola 68000. Współpraca firmy Acorn z Apple Computer zaowocowała utworzeniem odrębnej firmy o nazwie Advanced RISC Machines (ARM Ltd.)[5] oraz wprowadzeniem na rynek w 1990 roku procesora ARM6, który implementował model programowy ARMv3. Firma ARM nie zajmowała się produkcją procesorów, a jedynie sprzedażą licencji na zaprojektowaną architekturę. Taką właśnie licencję kupiła firma Digital Equipment, tworząc na jej podstawie popularny procesor StrongARM. Prace nad mikroarchitekturą ARM, prowadzone przez Digital Equipment, firmę produkującą przez wiele lat najszybsze procesory na świecie, sprawiły, że StrongARM osiągnął wydajność 211 MIPS przy zużyciu mocy, wynoszącym zaledwie kilka Watów. StrongARM implementował czwartą generację modelu programowego (ARMv4). Projekt został przejęty przez firmę Intel, która na bazie StrongARM opracowała nowy, wydajny procesor do zastosowań wbudowanych o nazwie XScale. Procesor XScale implementuje model programowy ARMv5 i wprowadza kilka nowych rozszerzeń.

3. Model programowy[6]

Architektura ARM została zaprojektowana w taki sposób, by jej implementacje były możliwe jak najprostsze i dzięki temu energooszczędne. ARM jest klasycznym przykładem 32-bitowej architektury RISC. Posiada specyficzne dla tej architektury cechy, takie jak duża liczba uniwersalnych rejestrów, rozdzielenie instrukcji, operujących na danych od instrukcji, operujących na pamięci, proste tryby adresowania i stała długość instrukcji. Specyficzne dla tego procesora cechy to możliwość przesuwania bitowego argumentów każdej instrukcji arytmetyczno-logicznej, tryby adresowania z automatyczną inkrementacją/dekrementacją, blokowe instrukcje zapisu/odczytu oraz możliwość warunkowego wykonania każdej instrukcji. W dalszej części pracy skupiono się na wersji 4T[7], jako wersji użytej w projekcie.

W tabeli 1 zebrano podstawowe cechy, charakterystyczne dla każdej wersji modelu programowego ARM.

3 Na podstawie [WIKI]
4 cyt. [WIKI] Rozdz. 1
5 Patrz [ARL]
6 Na podstawie [ARM]
7 [ARM], Preface. Architecture versions and variants, strona 5.

1. Model	2. Cechy
ARMv1	Podstawowe instrukcje arytmetyczno-logiczne bez mnożenia oraz instrukcje zapisu/odczytu, pozwalające na zapis jednego bajtu lub słowa (32-bitowego), instrukcje skoku i przerwanie programowe. 26-bitowa przestrzeń adresowa. Ta wersja nigdy nie została użyta w komercyjnym produkcie.
ARMv2	Dodano instrukcje mnożenia i mnożenia z akumulacją wyniku oraz zapewniono wsparcie dla koprocesorów. Dodano dwa dodatkowe rejestry do wykorzystania w trybie obsługi szybkiego przerwania oraz (w wersji 2a) atomowe instrukcje typu test and set nazwane SWP i SWB.
ARMv3	Rozszerzono przestrzeń adresową do 32 bitów. Dodano rejestry stanu CPSR i SPSR. Wcześniej informacje o stanie procesora zapisywane były w rejestrze R15. Rozszerzono listę rozkazów o instrukcje manipulacji na słowach stanu. Wprowadzono także dwa dodatkowe tryby pracy do obsługi wyjątków ochrony pamięci oraz nielegalnej instrukcji. Wersja trzecia (za wyjątkiem 3G) zapewniała zgodność z 26-bitowymi architekturami.
ARMv4	Dodano instrukcje odczytu/zapisu słów 16-bitowych oraz instrukcje do zapisu/odczytu z uzupełnieniem bitem znaku. Wprowadzono także wariant modelu programowego nazwany Thumb[8]. Dodano także nowy tryb uprzywilejowany o nazwie System, który mógł wykorzystywać rejestry dostępne w trybie użytkownika (R0-R15). Uściślono, które instrukcje powinny powodować wyjątek „Undefined Instruction”, a wsteczna kompatybilność z architekturami 26-bitowymi przestała być obowiązkowa.
ARMv5	Zmodyfikowano definicje kilku istniejących rozkazów, aby zwiększyć efektywność trybu Thumb. Dodatkowo dodano instrukcję, zliczającą wiodące zera, która usprawnia dzielenie całkowitoliczbowe oraz procedury ustalania priorytetów przerwań. Dodano także instrukcję programowej pułapki oraz rozszerzono zestaw instrukcji dla projektantów koprocesorów. Zawężono także zakres ustawiania flag przez instrukcje mnożenia.

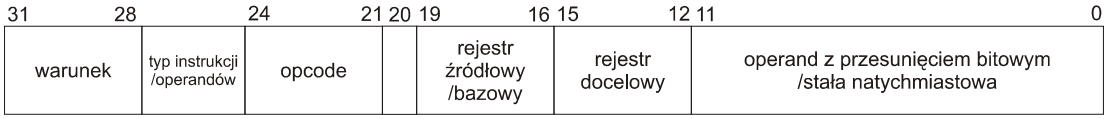
Tabela 1. Ewolucja modelu programowego ARM[9]

8 Patrz strona 11.

9 Na podstawie [ARM], Preface. Architecture versions and variants, strony 5-6.

Instrukcje[10]

Wszystkie rozkazy mają długość 32-bitów. Format typowego rozkazu przedstawiono na rys. 1.



Rys. 1. Format rozkazu ARM

Cztery najstarsze bity rozkazu określają warunek jego wykonania, czyli wymaganą kombinację flag w rejestrze stanu procesora. Jeżeli warunek wykonania nie jest spełniony, rozkaz traktowany jest jako NOP (instrukcja pusta). Kolejne elementy słowa rozkazowego to 3 bity, określające typy operandów (rejestry, stała natychmiastowa, offset w pamięci, przesunięcie bitowe), 4 bity, definiujące rozkaz (tzw. opcode) i bit, określający wariant danego rozkazu (np. odczyt lub zapis).

Rozkazy ARM można podzielić na cztery grupy: rozkazy arytmetyczno-logiczne, rozkazy operujące na pamięci, skoki i rozkazy dla koprocesorów. W architekturze ARM przez koprocesor rozumiany jest dodatkowy moduł procesora, implementujący specyficzne funkcje. Przykładami koprocesorów są jednostka do zarządzania układem MMU, koprocesor DSP do przetwarzania sygnałów (dodawany w układzie Intel PXA), koprocesor do monitorowania wydajności itp. Koprocesory zostały wprowadzone w ARMv2.

W skład rozkazów arytmetyczno-logicznych wchodzą instrukcje dodawania, odejmowania, mnożenia, mnożenia z akumulacją wyniku, przypisania między-rejestrowe, przesunięcia bitowe, iloczyn logiczny, suma logiczna, negacja i porównanie.

W skład rozkazów, operujących na pamięci, wchodzą instrukcje zapisu/odczytu i instrukcje odczytu/zapisu blokowego. Odczyt/zapis blokowy służy do jednoczesnego przesłania wartości wielu rejestrów. Adresy dostępu do pamięci muszą być wyrównane do 32-bitów. W przeciwnym wypadku wynik zapisu/odczytu może być nieprawidłowy. Standardowo procesor wykorzystuje kolejność bajtów little-endian, jednakże możliwe jest przełączenie w tryb big-endian.

Dostępne są trzy tryby adresowania: bezpośredni (adres obliczany jest jako suma/różnica stałej natychmiastowej i rejestru bazowego), rejestrowy (adres obliczany jest jako suma rejestru bazowego i rejestru przesunięcia) i rejestrowy-skalowany. Dostęp rejestrowy-skalowany wykorzystuje operandy typu przesunięcie bitowe. Operandy tego typu są ciekawą cechą ARM. Wykorzystywane są także w rozkazach arytmetyczno-logicznych. Pozwalają na określenie wartości drugiego operandu jako funkcji przesunięcia bitowego, wybranego rejestru procesora. Wartość przesunięcia może być podana przez stałą natychmiastową lub przez rejestr. Ten specyficzny sposób definiowania operandu może mieć zastosowanie przy optymalizacji przetwarzania liniowych zbiorów danych, znajdujących się w pamięci fizycznej. ARM posiada także instrukcje typu test and set, służące do atomowej wymiany zawartości rejestru i słowa pamięci.

Instrukcje skoku ARM to skok natychmiastowy i skok ze śladem. Operandem instrukcji skoku może być 24-o bitowy offset lub rejestr, który zawiera wartość bezwzględną adresu pamięci. Promień skoku z offsetem wynosi 32MB, ponieważ adres docelowy wyznaczany jest jako wartość offsetu przesunięta o 2 bity w lewo.

10 [ARM], A4. ARM Instructions, strona A4-1 (101)

Rozkazy skoku ze śladem służą do implementacji wywoływania funkcji. Przy wykonaniu tej instrukcji, aktualna zawartość licznika rozkazów zapisywana jest do rejestru R14 przed wykonaniem skoku. Można także wykonać skok bezpośrednio zapisując nową wartość w rejestrze R15, który jest licznikiem rozkazów.

W modelu programowym ARMv4T wprowadzono także instrukcje skrócone (tzw. **Thumb Instruction Set**), stąd litera T w nazwie modelu. W trybie skróconych instrukcji słowo rozkazowe skrócone zostało z 32-u do 16-u bitów. Pozwala to na zmniejszenie rozmiaru kodu programów. Instrukcje udało się skrócić poprzez usunięcie pola warunkowego wykonania i skróceniu adresu rejestru z czterech do trzech bitów. W skróconych instrukcjach pole opcode ma zróżnicowaną długość i zmienne położenie w obrębie słowa. Nie ma także możliwości przesuwania bitowego operandów rozkazu. Standardowo skrócone instrukcje operują na niskich rejestrach (R0 do R7). Istnieje także siedem instrukcji, operujących na wyższych rejestrach (R8 do R15), służących do przesyłania danych między rejestrami, dodawania, odejmowania oraz porównywania.

Rejestry[11]

Rejestry procesora zostały przedstawione na rys. 2.

Tryby pracy						
	Tryby uprzywilejowane					
		Tryby obsługi sytuacji wyjątkowych				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**IRQ**	**FIQ**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Rys. 2. Rejestry procesora ARM[12]

11 [ARM], A2.3. Registers, strona A2-4 (36)

12 Na podstawie [ARM], Fig. 2-1, strona A2-4 (36)

W procesorze dostępne jest trzydzieści jeden 32-bitowych rejestrów fizycznych ogólnego przeznaczenia.[13] Adres rejestru jest jednak 4-bitowy i programy mogą wykorzystywać tylko 16 rejestrów. Adres rejestru nie określa w sposób bezwzględny rejestru fizycznego tylko definiuje rejestr logiczny. Sposób odwzorowania rejestru logicznego w fizyczny zależy od trybu pracy procesora. Rejestry R0-R7 odwzorowane są tak samo we wszystkich trybach pracy procesora. Rejestry R8-R12 są powiązane z dwoma bankami rejestrów fizycznych. Jeden bank używany jest w trybie „Fast Interrupt", a drugi w pozostałych trybach (wymaga to dodatkowych 5 rejestrów fizycznych). Rejestry R13-R14 powiązane są z sześcioma bankami rejestrów fizycznych (po jednym dla każdego z 5 trybów obsługi wyjątków) i osobny dla tryby wykonania kodu (wymaga to 10-u dodatkowych rejestrów fizycznych). Tryby pracy procesora zostały opisane w dalszej części artykułu. Rejestry mogą być w zasadzie stosowane dowolnie, z tym zastrzeżeniem, że rejestr R15 służy jako licznik rozkazów (PC), a rejestr R14 używany jest przez funkcję skoku ze śladem jako rejestr adresu powrotu (LR). Rejestr R13 jest standardowo służy za rejestr wskaźnika stosu (SP).

Procesor ARM posiada 2 logiczne rejestry stanu (CPSR i SPSR).[14] W trybie wykonywania kodu możliwy jest dostęp wyłącznie do rejestru CPSR. Drugi rejestr logiczny o nazwie SPSR służy do zapamiętywania stanu procesora przy przechodzeniu do trybów obsługi wyjątków. Rejestr SPSR powiązany jest z pięcioma bankami rejestrów fizycznych, co oznacza, że każdy z trybów obsługi wyjątków operuje na własnej kopii rejestru.

W rejestrze stanu znajdują się flagi, wykorzystywane przy warunkowym wykonaniu kodu, ustawiane na ogół przez instrukcje porównujące. Dolne osiem bitów rejestru stanu to bity kontrolne, odpowiadające za blokowanie przerwań, przełączenie procesora w tryb Thumb i 5 bitów, określających tryb pracy procesora.

Tryby pracy procesora[15]

W opisie rejestrów wspomniano, że procesor może pracować w jednym z siedmiu trybów – User (usr), System (sys), FIQ (fiq), IRQ (irq), Supervisor (svc), Abor (abt) i Undefined (und).

Tryby User, System i Supervisor służą do wykonania programów w trybie użytkownika i w trybie jądra systemu operacyjnego. Pozostałe tryby służą do obsługi sytuacji wyjątkowych. Tryb User jest trybem nieuprzywilejowanym, co oznacza, że program nie może wykonywać wszystkich instrukcji procesora i nie ma dostępu do wszystkich jego zasobów oraz do wszystkich obszarów przestrzeni adresowej. Pozostałe tryby pracy są uprzywilejowane. Wprowadzenie dużej liczby trybów uzasadniane jest przez projektantów możliwością znacznej optymalizacji wydajności obsługi przerwań i wyjątków przez system operacyjny. Zmiana trybu pracy na inny może nastąpić po wystąpieniu przerwania lub wyjątku.

W tabeli 2 przedstawiono przeznaczenie i krótką charakterystykę trybów pracy procesora ARM.

13 [ARM], A.2.4. General-purpose registers, strona A2-5 (37)
14 [ARM], A2.5. Program status registers, strona A2-9 (41)
15 [ARM], A2.2. Processor modes, strona A2-3 (35)

3. Tryb pracy	4. Opis
User	Przeznaczony do pracy programów użytkowych. Ograniczona lista rozkazów, dostęp do pamięci i zasobów procesora.
System	Tryb uprzywilejowany dla systemu operacyjnego. Rejestry R0-R15 są takie same jak dla trybu User.
FIQ (Fast Interrupt)	Tryb obsługi przerwań, wymagających krótkiego czasu przystąpienia do obsługi.
IRQ (Interrupt)	Tryb obsługi przerwań zwykłych.
Supervisor	Tryb uprzywilejowany dla systemu operacyjnego. W tym trybie rejestry R13-R14 powiązane są z innym bankiem, niż w trybie User. Dostępny jest także rejestr SPSR.
Abort	Obsługa wyjątku nieuprawnionego dostępu do pamięci i próby wykonania niedozwolonej lub niepoprawnej instrukcji.
Undefined	Próba wykonania niezdefiniowanej instrukcji koprocesora.

Tabela 2. Tryby pracy procesora ARM

Obsługa przerwań i wyjątków[16]

W procesorze ARM przerwania sprzętowe zgłaszane są przy pomocy dwóch linii procesora. Przerwania typu IRQ zgłaszane są przy pomocy linii IRQ, natomiast przerwania szybkie tzw. FIQ zgłaszane są przy pomocy linii FIQ. Wyższy priorytet ma przerwanie na linii FIQ. Przerwania sprzętowe służą głównie do asynchronicznego raportowania o zmianie stanu urządzeń wejścia-wyjścia, nadzorowanych przez procesor. Procesor ARM implementuje także mechanizm przerwań programowych. Mechanizm ten polega na wprowadzeniu procesora w stan obsługi przerwania przy pomocy odpowiedniej instrukcji. Konsekwencją pojawienia się przerwania (zarówno sprzętowego jak i programowego) jest przełączenie trybu pracy procesora na uprzywilejowany.

Wyjątki procesora służą do obsługi sytuacji awaryjnych, wynikających z błędów wykonania programu. Przykładem sytuacji wyjątkowej, zgłaszanej przez procesor jako wyjątek, jest np. próba wykonania niezdefiniowanej instrukcji lub próba dostępu do chronionego obszaru pamięci.

Po nadejściu sygnału przerwania lub po wystąpieniu wyjątku, procesor przechodzi do trybu obsługi przerwania. W tym celu, w odpowiednim dla trybu obsługi rejestrze R14 (R14_fiq, R14_irq, R14_abt itd.) zapamiętywany jest aktualny stan licznika rozkazów, w odpowiednim dla trybu rejestrze SPSR zapamiętywana jest aktualna wartość rejestru stanu, a w rejestrze CPSR zapisywany jest identyfikator trybu obsługi. Jeżeli w momencie wystąpienia przerwania (wyjątku) procesor działał w trybie wykonania instrukcji Thumb, to przed przejściem do procedury obsługi następuje wyjście z tego trybu. Jeżeli przerwanie nadeszło na linii FIQ (szybkich przerwań) lub wystąpił sygnał Reset to blokowane są szybkie przerwania. W przypadku przerwań i wyjątków zawsze blokowane są przerwania na linii IRQ.

16 [ARM], A2.6. Exceptions, strona A2-13 (45)

Po zablokowaniu przerwań procesor przechodzi do wykonania kodu umieszczonego pod odpowiednim, ściśle określonym adresem w pamięci. Instrukcja do wykonania pobierana jest z wektora przerwań, który z reguły umieszczony jest na początku przestrzeni adresowej. W tabeli 3 przedstawiono adresy wektorów przerwań i wyjątków oraz docelowe tryby pracy procesora.

Przerwanie lub sytuacja wyjątkowa	Tryb pracy	Adres wektora
Reset	Supervisor	0x00000000
Nielegalna instrukcja (Undefined Instruction)	Undefined	0x00000004
Przerwanie programowe (Software Interrupt)	Supervisor	0x00000008
Błąd pobrania rozkazu (Prefetch Abort)	Abort	0x0000000C
Błąd dostępu do pamięci danych (Data Abort)	Abort	0x00000010
Normalne przerwanie (IRQ)	IRQ	0x00000018
Szybkie przerwanie (FIQ)	FIQ	0x0000001C

Tabela 3. Adresy wektorów przerwań[17]

Po wykonaniu procedury obsługi przerwania (wyjątku) wymagany jest powrót do wykonania kodu spod adresu, zapamiętanego w rejestrze R14 trybu. Przed powrotem należy odtworzyć rejestr stanu. Przywrócenie rejestru stanu następuje automatycznie, jeśli do powrotu z obsługi przerwania użyjemy odpowiednich instrukcji.

Interesującym mechanizmem ARM są przerwania szybkie tzw. FIQ. Przy obsłudze szybkiego przerwania (FIQ) procesor przełącza się do odrębnego trybu pracy, w którym dostępne są kopie rejestrów R8-R14, co pozwala na natychmiastowe rozpoczęcie obsługi bez potrzeby zapamiętywania rejestrów przed ich użyciem. Tryb ten służy do obsługi przerwań, wymagających szybkiej obsługi, np. do odbierania danych od urządzenia nie posiadającego buforowania.

Przestrzeń adresowa.[18]

W procesorach ARM dostępna jest wyłącznie pojedyncza, 32-bitowa, liniowa przestrzeń adresowa, która służy zarówno do dostępu do pamięci fizycznej, jak i do urządzeń wejścia-wyjścia. Procesory ARM mogą zaadresować 4GB pamięci. Z reguły adres w pamięci być wyrównany do 4 bajtów (z wyjątkiem instrukcji ładowania/składowania danych 16-to i 8-bitowych oraz pobierania instrukcji trybu *"Thumb"*). Kolejność bajtów jest zależna od implementacji. Zazwyczaj procesory ARM używają kolejności *little-endian* lub obydwu (istnieje możliwość przełączenia).

4. Zastosowania procesorów o architekturze ARM

W dzisiejszych czasach, procesory implementujące architekturę ARM stają się jednymi z najpopularniejszych w zastosowaniach wbudowanych. Wszelakie ich przykłady możemy znaleźć niemal na każdym kroku. Mikrokontrolery ze rdzeniami w wersji 4 znajdziemy w telefonach komórkowych, cyfrowych aparatach fotograficznych, urządzeniach GPS, odtwarzaczach MP3, PenDrive'ach i w wielu innych. W bardziej skomplikowanych urządzeniach jak Palmtopy, PDA, MDA, czy systemy nawigacji znajdziemy ARMv5 w tym rdzenie Intel XScale. Mikrokontrolery i układy SoC z rdzeniem ARM wykorzystywane są wszędzie tam, gdzie wymagane jest użycie taniego i wydajnego procesora 32-bitowe o możliwie małym zużyciu energii i pozwalającego na szybką i wydajną pracę wszelakiego typu urządzeń wbudowanych. Procesory te wypierają powoli z rynku wysłużone już procesory rodziny '51.

17 Na podstawie [ARM], tab. 2-3, strona A2-13

18 [ARM], A2.7. Mmemory and memory mapped I/O, strona A2-22 (54)

5. Przegląd mikrokontrolerów z rdzeniem ARM.

Na rynku dostępnych jest bardzo wiele mikrokontrolerów z rdzeniem ARM. Z punktu widzenia tego projektu interesujące są modele wyposażone w rdzeń ARM7 implementujący czwartą wersję architektury z trybem Thumb oraz bez jednostki zarządzania pamięcią. Architektura ta została wybrana z racji powszechności jej zastosowań. Jest to architektura stosowana w wielu systemach wbudowanych. Znajdujemy ją w telefonach komórkowych, odbiornikach GPS, modemach i routerach sieciowych, oraz w cyfrowych aparatach fotograficznych. Jest to klasyczna architektura ARM z trybem dekodowania instrukcji Thumb. Architekturę tego typu znajdziemy w mikrokontrolerach Philips serii LPC2000 oraz Atmel AT91.

Procesory firmy Atmel są ciekawą propozycją, gdyż niektóre wersje posiadają sporą ilość pamięci RAM (nawet 256 KB). Wersję z dużą ilością pamięci operacyjnej posiadają jednak bardzo skromne wyposażenie. Pozostałe modele Atmela mają parametry porównywalne do odpowiedników produkcji Philipsa.

Najszerszą ofertę posiada firma NXP Semiconductors (dawne Philips Semiconductors). Lista modeli procesorów serii LPC2000 jest długa, a do wyboru mamy przeróżne kombinacje pamięci SRAM i Flash. Wadą jest to, że maksymalnie układy te wyposażane są w 64 KB pamięci SRAM. W zamian za to można otrzymać modele posiadające nawet 512 KB pamięci Flash. Układy serii LPC są też bogato wyposażone.

W tabeli 4 zaprezentowane zostały parametry przykładowych układów:

Model	***SRAM [KB]***	***Flash [KB]***	***Freq. [Mhz]***	***UART***	***SPI***	***ADC***	***DAC***	***USB***	***GPIO***	***Timery*** *+watchdog/RTC*
LPC2106	64	128	60	2	1	-	-	-	32	2 (32b) +2
LPC2148	32	512	60	2	2	8+6	1	1	45	2 (32b) +2
AT91R40008	256	-	75	2	-	-	-	-	32	3 (16b) +1
AT91SAM7S128	32	128	55	3	1	8	-	1	32	3 (16b) +2

Tabela 4. Mikrokontrolery z rdzeniem ARM.[19]

Głównymi priorytetami przy wyborze mikrokontrolera były 3 aspekty:

- po pierwsze duża ilość pamięci nieulotnej,
- po drugie możliwie duża ilość pamięci SRAM,
- po trzecie zestaw urządzeń peryferyjnych; najistotniejsze są przetworniki analogowo-cyfrowe i cyfrowo-analogowe oraz kontroler USB,

Wersja jądra przygotowana w ramach projektu nie wykorzystuje co prawda wszystkich urządzeń peryferyjnych ani całej pamięci Flash, ale wybór został dokonany z myślą o późniejszym rozwoju projektu. Chodziło o wybranie platformy, która ze względu na spore możliwości umożliwi dalszy rozwój, po ukończeniu części objętej tym projektem.

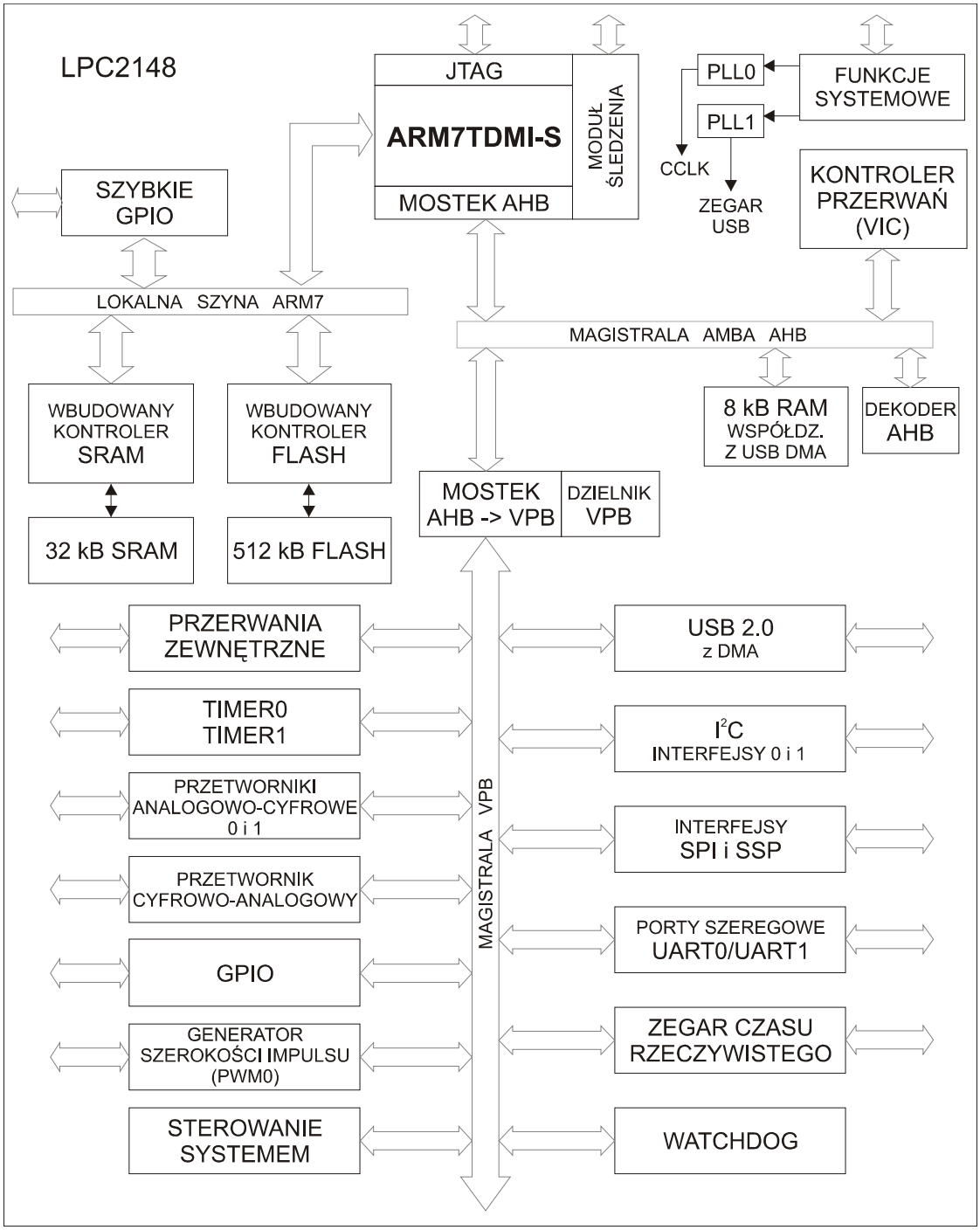
Wybrany został mikrokontroler Philips LPC2148, który najlepiej spełnił założenia.

19 Dane mikrokontrolerów Philips LPC za: [LPC]. Dane Atmel AT91 za: [AT]. LPC2148 posiada 2-kanałowy przetwornik analogowo-cyfrowy (kanały 8-pinowy i 6-pinowy)

III. Philips LPC2148

1. Charakterystyka mikrokontrolera[20]

Rysunek 3. przedstawia schemat blokowy mikrokontrolera:



Rys. 3. Schemat blokowy mikrokontrolera LPC2148.[21]

20 Na podstawie [UM]
21 Na podstawie [UM], Fig. 1 LPC2141/2/4/6/8 block diagram, strona 7.

Rodzina Philips LPC2100[22] to bardzo popularna seria mikrokontrolerów opartych na architekturze ARM. Mikrokontrolery tej serii wyposażone są w rdzeń ARM7TDMI-S[23], implementujący architekturę ARMv4T. Układ nie posiada jednostki zarządzania pamięcią (MMU, ang, Memory Management Unit). Obok procesora mamy szereg układów i urządzeń peryferyjnych:

- 8-40KB statycznej pamięci o dostępie swobodnym (SRAM, ang, Static Random Access Memory),
- 32-512KB pamięci nieulotnej FLASH,
- kontroler USB 2.0 wraz z pamięcią dla DMA,
- 1 lub 2 przetworniki analogowo-cyfrowe (6-o lub 8-pinowe),
- przetwornik cyfrowo analogowy,
- dwa 32-bitowe liczniki
- modulator szerokości impulsu (PWM, ang. Pulse Wide Modulator),
- watchdog,
- zegar czasy rzeczywistego (RTC, ang. Real Time Clock) z odrębnym zasilaniem i taktowaniem,
- dwa asynchroniczne porty szeregowe (UART, ang. Universal Asynchronous Receiver-Transmiter),
- dwa synchroniczne porty szeregowe (SPI, ang Serial Peripheral Interface; SSP, ang. Synchronus Serial Port),
- wektorowy kontroler przerwań (VIC, ang. Vectored Interrupt Controller).

Układ posiada także moduł akceleracji pamięci oraz w pętlę fazową (PLL, ang. Phase Lock Loop) generującą zegar dla układu, oraz odrębną dla kontrolera USB. Urządzenia peryferyjne połączone są z procesorem poprzez trzy magistrale:

1. **Lokalna magistrala ARM7 (ang. ARM7 Local Bus)**

   Przy pomocy 128-bitowej magistrali lokalnej układ łączy się z kontrolerami pamięci wewnętrznej (SRAM i Flash), oraz z układem obsługującym szybkie wejścia/wyjścia cyfrowe (Fast GPIO, ang. Fast General Purpose I/O)

2. **Magistrala AMBA AHB**

   Magistrala AHB (Advanced High-preformance Bus) służy do połączenia z kontrolerem przerwań oraz z magistralą VPB (VLSI Peripheral Bus). Układy peryferyjne szyny AHB zajmują 2MB na szczycie 4GB przestrzeni adresowej procesora ARM. Każdy układ peryferyjny magistrali AHB alokuje 16KB przestrzeni adresowej.

3. **Magistrala VPB**

   Do tej magistrali dołączone są pozostałe urządzenia peryferyjne, w które wyposażony jest mikrokontroler. Szyna VPB łączy się z magistralą z procesorem poprzez mostek AHB do VPB. Częstotliwość taktowania układów peryferyjnych ustala jest na podstawie zegara systemowego, przy pomocy dzielnika częstotliwości VPB (VPB Divider). Układy peryferyjne magistrali VPB zajmują 2MB przestrzeni adresowej ulokowanej powyżej 3.5GB. Każde urządzenie dołączone do magistrali zajmuje 16KB tej przestrzeni.

22 Patrz [NXP]
23 Patrz [A7].

Połączenie wbudowanych urządzeń peryferyjnych z wyprowadzeniami układu realizowane jest poprzez specjalny **Pin Connect Block**.

Pętla fazowa (Phase Locked Loop)

Moduły pętli fazowej umożliwiają zwielokrotnienie wejściowej częstotliwości (oscylatora kwarcowego), celem wygenerowania zegara systemowego. Częstotliwość wejściowa może zawierać się w zakresie od 10 do 25MHz. Układ pozwala podnieść tę częstotliwość do 60MHz (48MHz dla USB).

Moduł akceleracji pamięci (Memory Acceleration Module)

MAM pozwala zmaksymalizować wydajność procesora, przy wykonywaniu kodu bezpośrednio z pamięci Flash. Moduł zawiera trzy 128-bitowe bufory pozwalające zminimalizować zatrzymywanie procesora na czas odczytu kolejnych instrukcji z pamięci Flash.

Bufor instrukcji (ang. Prefetch Buffer) przechowuje linię pamięci Flash zawierającą wykonywaną instrukcję. Bufor soku (ang. Barnch Trial buffer) przechowuje linię pamięci zawierającą instrukcję wykonaną po ostatnim skoku. Bufor danych (ang. Data Buffer) przechowuje linię pamięci flash zawierającą daną, do której ostatnio następowało odwołanie.

Układ niweluje opóźnienia przy wykonywaniu sekwencyjnego kodu, jeżeli okres zegara systemowego jest większy od jednej czwartej czasu dostępu do pamięci Flash.

Kontroler przerwań (Vectored Interrupt Controller)

Układy serii LPC2100 wyposażone są w **ARM PrimeCell™ Vectored Interrupt Controller.**

VIC pozwala na używanie 32 źródeł przerwań. W programowalny sposób, każde z tych źródeł przydzielane jest do jednej z trzech kategorii: FIQ, wektorowane IRQ i niewektorowane IRQ. Programowalność oznacza, że priorytety przerwań mogą być przydzielane i ustawiane dynamicznie.

Najwyższy priorytet mają przerwania przydzielone do kategorii FIQ (szybkie przerwania, ang. Fast Interrupt reQuest). Jeśli więcej niż jedno przerwanie przypisane jest do kategorii FIQ, to kontroler wykorzystuje sumę żądań do wygenerowania sygnału FIQ dla procesora. Najkrótszy czas obsługi uzyskamy, gdy do kategorii FIQ przypisane będzie tylko jedno źródło przerwania. Jeśli do tej kategorii przypisane jest więcej przerwań, możliwe jest odczytanie z kontrolera, które z urządzeń zgłasza przerwanie.

Żądania wektorowane mają średni priorytet. Do tej kategorii można przypisać do 16 źródeł przerwań. Do każdego z 16 slotów można przypisać jedno z 32 źródeł przerwań, przy czym sloty te mają stałe priorytety. Najwyższy priorytet ma slot o numerze 0;

Najniższy priorytet mają przerwania niewektorowane.

Kontroler sumuje wszystkie żądania, zarówno z przerwań wektorowanych jak i niewektorowanych, produkując sygnał IRQ dla procesora. Funkcja obsługi przerwania IRQ może przeczytać z kontrolera adres odpowiedniej funkcji obsługi. Jeśli są oczekujące przerwania wektorowane, to zwracany jest adres funkcji obsługi przerwania o najwyższym priorytecie. Jeśli zgłaszane jest przerwanie z grupy niewektorowanych, to zwracany jest adres domyślnej funkcji obsługi.

Asynchroniczne porty szeregowe (UART)

Kontroler asynchronicznych portów szeregowych jest zgodny z układem 16550 National Semiconductors, który stał się standardem. Wyposażony jest w 16-bajtowe kolejki FIFO dla nadajnika i odbiornika. Posiada wbudowany, ułamkowy, generator taktowania z możliwością automatycznego ustawiania prędkości. Możliwe jest wykorzystanie sprzętowej bądź programowej kontroli przepływu. Drugi port szeregowy (UART1) w modelach LPC2144/6/8 dodatkowo posiada standardowe sygnały interfejsu modemowego, wraz z dołączoną kontrolą przepływu (auto-CTS/RTS) w pełni wspierane sprzętowo.

Kolejki FIFO odbiorników pozwalają ustawić próg zgłaszania przerwania na 1, 4, 8 lub 14 bajtów.

Zgodnie ze standardem kontrolerów '550 układ generuje 5 przerwań:

- Stan odbiornika (RLS, ang. Receive Line Status) – zgłaszane jest w przypadku wystąpienia jednego z 4 błedów w odbiorniku.
- Dane do odbioru (RDA, ang. Receive Data Available) – zgłaszane, gdy w buforze odbiorczym czeka odczytany bajt (przy wykorzystaniu kolejek FIFO – określona przez próg liczba bajtów)
- Timeout odbiornika (CTI, ang. Charackter Time-out Indicator) – zgłaszane jedynie przy stosowaniu kolejki FIFO i niezerowego progu. Oznacza konieczność odebrania bajtu z kolejki FIFO w przypadku, gdy w kolejce jest mniej bajtów niż określa próg, ale kontroler nie otrzymał żadnych danych od określonego czasu.
- Przerwanie nadajnika (THRE, ang. Transmit Holding Register Empty) – zgłaszane jest, kiedy posta jest kolejka FIFO nadajnika i można wysyłać kolejne bajty danych

Błędy przy których zgłaszane jest przerwanie RLS to: przepełnienie kolejki (OE, ang. Overrun Error), błąd parzystości (PE, ang. Parity Error), błąd ramki (FE, ang. Frame Error), przerwa w nadawaniu (BI, ang. Brake Interrupt)

Liczniki

Układy serii LPC2100 wyposażone są w dwa, 32-bitowe timery/liczniki. Każdy z liczników wyposażony jest w cztery 32-bitowe rejestry dla dopasowań oraz w cztery rejestry przypisywane do odpowiednich źródeł zewnętrznych. Liczniki mogą działać w trybie licznika (zliczając wystąpienia konkretnego zdarzenia), bądź timera (zliczając zbocza zegara systemowego).

2. Pamięć. Mapa pamięci.

Mikrokontrolery serii LPC2100 wyposażone są we wbudowaną pamięć RAM. Pojemność pamięci zależna jest od modelu i wacha się od 8 do 40KB. Wybrany do realizacji projektu model, LPC2148 wyposażono w 32+8KB pamięci SRAM. Na tą pamięć składa się 32KB pamięci ogólnego przeznaczenia oraz 8KB pamięci dedykowanej dla układu DMA kontrolera USB, nic jednak nie stoi na przeszkodzie w używaniu tej pamięci przez program użytkownika.

Układy wyposażone są także w nieulotną pamięć Flash o pojemności od 32 do 512KB (w zależności od modelu). LPC2148 zawiera 512KB pamięci. Pamięć flash podzielona jest na 27 sektorów. Z charakterystyki pamięci Flash wynika, że zapisów można

dokonywać sektorami. Przestrzeń pamięci podzielna jest na 14 sektorów 4KB oraz na 14 sektorów po 32KB. Małe sektory znajdują się na początku pamięci (pierwsze 32KB) oraz na końcu (ostatnie 32KB). Z przestrzeni tej wydzielone jest 12KB na końcu pamięci Flash. Obszar ten zajmowany jest przez Philips Bootloader, i jako taki nie może być zapisywany przy użyciu technologii ISP/IAP. Wymiana bootloadera możliwa jest tylko poprzez złącze JTAG.

Mapowanie obszarów pamięci

Mamy trzy tryby mapowania obszarów pamięci. Po każdym restarcie system wchodzi w tryb programu ładującego (ang. Boot Loader Mode). W tym trybie wektory przerwań przemapowane są na wektory dostarczone przez Philips Bootloader, czyli umieszczone na początku bootblocku. Program ładujący sprawdza, czy w pamięci Flash znajduje się prawidłowy kod programu użytkownika i przełącza w tryb User Flash. W tym trybie wektory przerwań nie są mapowane i wskazują bezpośrednio na pamięć Flash. Program użytkownika może przełączyć procesor w tryb User RAM, w którym wektory przerwań przemapowane są na początek pamięci SRAM. Zmiany trybu można dokonań poprzez modyfikację rejestru MEMMAP.

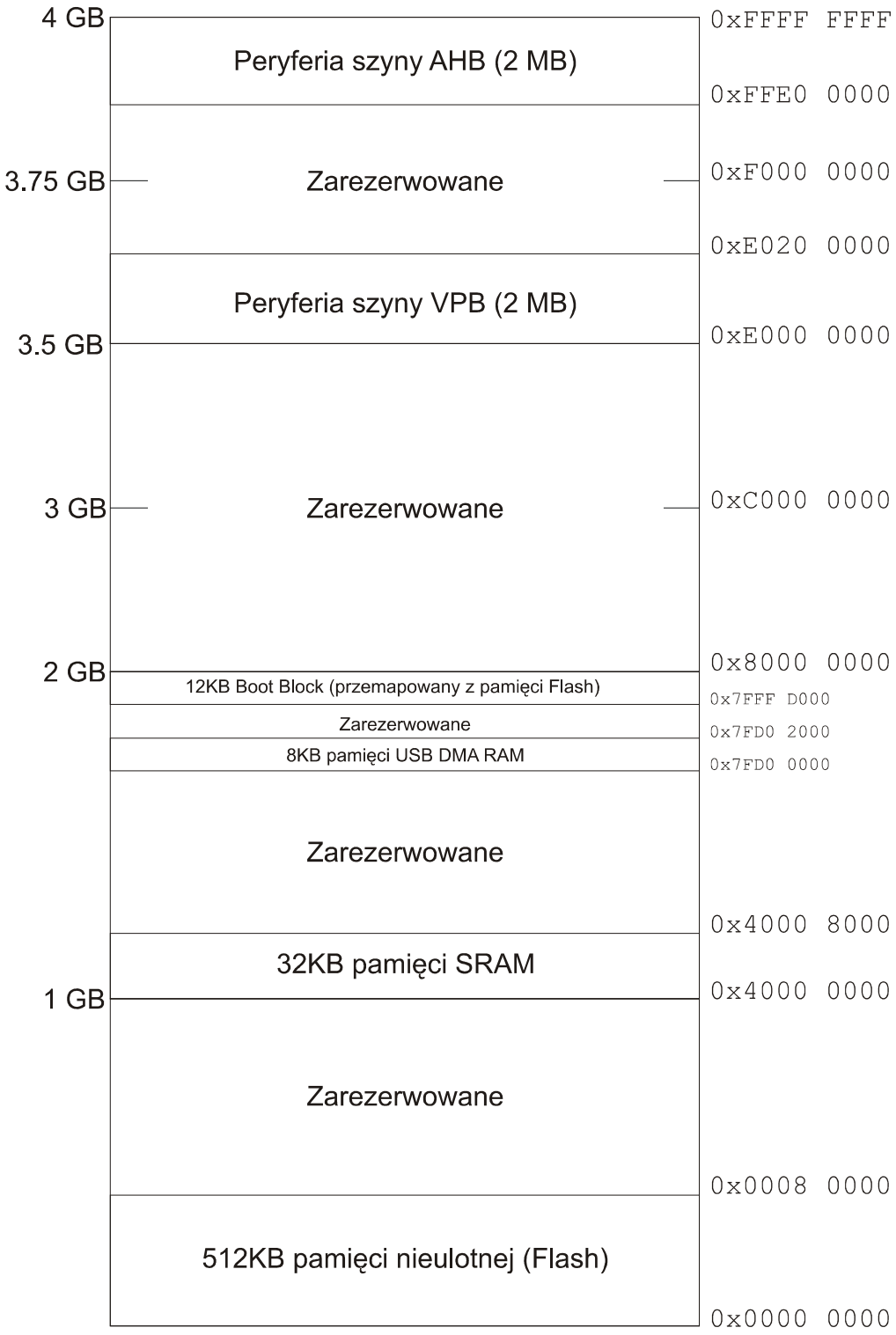
Programowanie pamięci Flash

Programowanie pamięci Flash mikrokontrolera może zostać przeprowadzone na trzy sposoby:

- ISP (ang. In-System Programming) – jest to sposób programowania pamięci udostępniany przez Philips Bootloader. Pozwala on załadować dane do pamięci Flash poprzez interfejs szeregowy. Wykorzystywany jest do tego celu port UART0. ISP wykorzystuje prosty, tekstowy interfejs komend. Do ładowania programu najlepiej posłużyć się dostarczanym przez producenta programem Philips LPC2000 Flash Utility[24]. Program ten implementuje język ISP i umożliwia załadowanie pliku w formacie Intel HEX, jak również odczytanie zawartości pamięci Flash oraz SRAM. ISP inicjowane jest, jeśli w trybie Reset zostanie zgłoszone przerwanie zewnętrzne na linii EINT1.
- IAP (ang. In-Application Programming) – jest to sposób pozwalający na programowanie pamięci Flash z aplikacji użytkownika. Udostępniany jest zestaw komend pozwalający na zapisywanie danych do pamięci Flash, jak również usuwanie zawartości. Jako, że podczas operacji na pamięci Flash, nie może ona być używana, IAP wykorzystuje 32B na końcu pamięci SRAM dla przechowywania kodu niezbędnego do wykonania poleceń. Programy użytkowe nie mogą używać tego fragmentu pamięci, jeśli IAP ma być wykorzystywane.
- Przez złącze JTAG – jeżeli niemożliwe jest korzystanie z portu szeregowego istnieje możliwość programowania pamięci Flash poprze złącze JTAG. Odpowiednie emulatory można zakupić u producentów zestawów uruchomieniowych. Metoda ta jest rzadko używana ze względu na prostotę wykorzystania ISP. Jedynym przypadkiem, w którym konieczne jest użycie tej metody, to wymiana bootloadera.

24 Program do pobrania z: [ISP]

Mapa pamięci



Rys. 4. Mapa pamięci LPC2148[25]

25 Na podstawie [UM], Fig. 2, strona 8.

3. Inicjalizacja[26]

W procesie inicjalizacji systemu możemy wyróżnić dwa zasadnicze aspekty. Pierwszy to odpowiednie przygotowanie obrazu programu ładowanego do pamięci mikrokontrolera. Drugi aspekt to inicjalizacja podstawowych modułów mikrokontrolera, niezbędnych do prawidłowej i wydajnej pracy oraz inicjalizacja struktur pamięci i stosu. Drugi aspekt inicjalizacji jest zależny od aplikacji. Z tego powodu, szczegółowe informacje na temat konfiguracji podane będą w rozdziałach opisujących implementację poszczególnych modułów.

Przygotowanie obrazu

Po starcie procesor rozpoczyna wykonywanie kodu od adresu 0x0000 0000, w trybie Supervisor. Z faktu tego wynika, że obraz umieszczony w pamięci Flash musi zaczynać się od adresu 0. Ze względu na właściwości architektury ARM, pierwsze 4 słowa programu powinny zawierać instrukcje skoku do funkcji obsługi sytuacji wyjątkowych. Pierwsze 32 bajty pamięci Flash są po starcie sprawdzane przez program ładujący pod kątem poprawności. Sprawdzana jest suma kontrolna umieszczona pod adresem 0x0000 0014. Jest to adres zarezerwowany przez definicję modelu programowego, a używany przez Philips Bootloader do weryfikacji poprawności kodu znajdującego się w pamięci Flash.

Specyficzne znaczenie ma także kolejne 32 bajty pamięci, gdyż pierwsze 64 bajty pamięci podlegają systemowi mapowania obszarów. Jeżeli kod programu lub tylko wektory przerwań, umieszczane są w pamięci RAM, istotne jest, aby instrukcje skoku wektorów były niezależne od położenia tychże wektorów w przestrzeni adresowej. W tym celu stosuje się dodatkowe 32 bajty remapowalnej pamięci, w których można przechować pełne, 32-bitowe, adresy funkcji obsługi przerwań i wyjątków.

Przykładowy kod rozpoczynający program powinien wyglądać następująco:

```
00000000 <_start>:
       0: e59ff018  ldr      pc, [pc, #24]      ; 20 <Reset_Addr>
       4: e59ff018  ldr      pc, [pc, #24]      ; 24 <PLLCFG_VALUE>
       8: e59ff018  ldr      pc, [pc, #24]      ; 28 <SWI_Addr>
       c: e59ff018  ldr      pc, [pc, #24]      ; 2c <PAbt_Addr>
      10: e59ff018  ldr      pc, [pc, #24]      ; 30 <DAbt_Addr>
      14: e1a00000  nop                ;(mov r0,r0)
      18: e59ff018  ldr      pc, [pc, #24]      ; 38 <irq_addr>
      1c: e59ff018  ldr      pc, [pc, #24]      ; 3c <FIQ_Addr>

00000020 <Reset_Addr>:
      20: 0000004c  ; <reset_handler>
00000024 <Undef_Addr>:
      24: 0000204c  ; <endef_handler>
00000028 <SWI_Addr>:
      28: 00002138  ; <swi_handler>
0000002c <PAbt_Addr>:
      2c: 0000215c  ; <pabt_handler>
00000030 <DAbt_Addr>:
      30: 0000223c  ; <dabt_handler>
      34: 00000000
00000038 <irq_addr>:
      38: 00000128  ; <irq_handler>
0000003c <FIQ_Addr>:
      3c: 0000235c  ; <fiq_handler>
```

Początek pliku _startup.lst

26 Na podstawie [UM] i [ARM]

Aby możliwe było używanie kodu napisanego w języku C konieczne jest przygotowanie segmentu zmiennych globalnych, zarówno inicjalizowanych wartością jak i zerowanych. Ze względu na brak możliwości zwykłych zapisów w pamięci Flash niezbędne jest, aby w czasie wykonanie programu segmenty te znajdowały się w pamięci ulotnej. Aby zapewnić trwałość zmiennych inicjalizowanych wartością, konsolidator, przygotowując plik wynikowy, umieszcza fizycznie segment danych tuż za segmentem kodu, przy czym w odwołaniach do danych używa wirtualnego adresu, który jest adresem docelowym tego segmentu. Podczas inijalizacji systemu konieczne jest przekopiowanie tych danych pod adresy docelowe, oraz wyzerowanie segmentu danych inicjalizowanych zerami. W tym celu wykorzystuje się fragment kodu w postaci:

```
# Relocate .data section (Copy from ROM to RAM)
                    LDR       R1, =_etext
                    LDR       R2, =_data
                    LDR       R3, =_edata
LoopRel:            CMP       R2, R3
                    LDRLO     R0, [R1], #4
                    STRLO     R0, [R2], #4
                    BLO       LoopRel


# Clear .bss section (Zero init)
                    MOV       R0, #0
                    LDR       R1, =_bss
                    LDR       R2, =_ebss
LoopZI:             CMP       R1, R2
                    STRLO     R0, [R1], #4
                    BLO       LoopZI
```

Fragment pliku _startup.S

Aby poprawnie zbudować plik wynikowy niezbędne jest posłużenie się specjalnie przygotowanym skryptem dla konsolidatora. Skrypt musi określać zakresy adresów pamięci Flash jak i SRAM, przypisać poszczególnym sekcjom pliku odpowiednie zakresy adresów fizycznych i wirtualnych, oraz ustawić wymagane przez kod inicjalizujący stałe: _etext, _data, _edata, _bss oraz _ebss. Przykładowy skrypt, wykorzystany w projekcie, przedstawiam poniżej:

```
/*
 * Phoenix-RTOS
 *
 * armplo - operating system loader for ARM7TDMI
 *
 * Linker script for Philips LPC2138 ARM microcontroller
 * applications that execute from Flash .
 *
 * Copyright 2006 Radoslaw F. Wawrzusiak
 *
 * Copyright 2006 D.W. Hawkins (dwh@ovro.caltech.edu)
 * Derived from: Philips 05: Project Number AR1803
 *
 * This file is part of Phoenix-RTOS.
 *
 * Phoenix-RTOS is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Phoenix-RTOS kernel is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Phoenix-RTOS kernel; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/* The LPC2138 has 512kB of Flash , and 32kB SRAM */
MEMORY
{
          flash (rx) : org = 0x00000000 , len = 0x00080000
          sram (rw)  : org = 0x40000000 , len = 0x00008000
}

SECTIONS
{
          /*
           * ----------------------------------------------------------
           * . text section (executable code)
           * ----------------------------------------------------------
           */
          .text :
          {
                    _startup.o (.text)
                    *(.text)
                    *(.glue_7t) *(.glue_7)
          } > flash
          . = ALIGN(4);

          /*
           * ----------------------------------------------------------
           * .rodata section (read/only (const) initialized variables)
           * ----------------------------------------------------------
           */
          .rodata :
          {
                    *(.rodata)
          } > flash
          . = ALIGN(4);

          /* End-of-text symbols */
          _etext = . ;
          PROVIDE (etext = .);

          /* ----------------------------------------------------------
           * . data section (read/write initialized variables )
           * ----------------------------------------------------------
           *
           * The values of the initialized variables are stored
           * in Flash , and the startup code copies them to SRAM.
           *
           * The variables are stored in Flash starting at _etext,
           * and are copied to SRAM address _data to _edata.
           */
          .data : AT (_etext)
          {
                    _data = . ;
                    *(.data)
          } > sram
          . = ALIGN(4);

          _edata = . ;
          PROVIDE (edata = .);

          /*
           * ----------------------------------------------------------
           * .bss section (uninitialized variables)
           * ----------------------------------------------------------
```

```
                 *
                 * These symbols define the range of addresses in SRAM that
                 * need to be zeroed.
                 */
                .bss :
                {
                          _bss = . ;
                          *(.bss)
                          *(COMMON)
                } > sram
                . = ALIGN(4);
                _ebss = . ;

                _end = . ;
                PROVIDE (end = .);

                /* Stabs debugging sections. */
                .stab                 0 : { *(.stab) }
                .stabstr              0 : { *(.stabstr) }
                .stab.excl            0 : { *(.stab.excl) }
                .stab.exclstr         0 : { *(.stab.exclstr) }
                .stab.index                     0 : { *(.stab.index) }
                .stab.indexstr        0 : { *(.stab.indexstr) }
                .comment              0 : { *(.comment) }
                /* DWARF debug sections.
                 * Symbols in the DWARF debugging sections are relative
                 * to the beginning of the section so we begin them at 0.
                 */
                /* DWARF 1 */
                .debug                0 : { *(.debug) }
                .line                 0 : { *(.line) }
                /* GNU DWARF 1 extensions */
                .debug_srcinfo        0 : { *(.debug_srcinfo) }
                .debug_sfnames        0 : { *(.debug_sfnames) }
                /* DWARF 1 . 1 and DWARF 2 */
                .debug_aranges        0 : { *(.debug_aranges) }
                .debug_pubnames       0 : { *(.debug_pubnames) }
                /* DWARF 2 */
                .debug_info           0 : { *(.debug_info .gnu.linkonce.wi.*) }
                .debug_abbrev         0 : { *(.debug_abbrev) }
                .debug_line                     0 : { *(.debug_line) }
                .debug_frame          0 : { *(.debug_frame) }
                .debug_str            0 : { *(.debug_str) }
                .debug_loc            0 : { *(.debug_loc) }
                .debug_macinfo        0 : { *(.debug_macinfo) }
                /* SGI/MIPS DWARF 2 extensions */
                .debug_weaknames      0 : { *(.debug_weaknames) }
                .debug_funcnames      0 : { *(.debug_funcnames) }
                .debug_typenames      0 : { *(.debug_typenames) }
                .debug_varnames       0 : { *(.debug_varnames) }
}
```

Plik lpc2148_flash.ld[27]

27 Zapożyczone z [AR]

IV. System operacyjny czasu rzeczywistego Phoenix-RTOS

Założenia projektu Phoenix

"Celem projektu Phoenix jest stworzenie w pełni samodzielnego mikrojądra systemu czasu rzeczywistego posiadającego wbudowany mechanizm tolerowania awarii. Jądro ma pozwolić na konstruowanie rozproszonych systemów czasu rzeczywistego"[28]

1. Systemy czasu rzeczywistego

Charakterystyczną cechą systemów czasu rzeczywistego jest krytyczność czasu reakcji. W systemach ogólnego przeznaczenia jedynym warunkiem poprawności wykonania programu jest właściwa kolejność wykonania instrukcji. W systemach czasu rzeczywistego dochodzi do tego czas wykonania poszczególnych grup instrukcji. Dzielimy je na systemy o łagodnych (miękkich) ograniczeniach czasowych (ang. soft real-time systems) oraz na systemy rygorystyczne (twarde, ang, hard real-time systems). W systemach łagodnych dopuszcza się sporadyczne przekroczenia limitu czasu wykonania, natomiast systemy rygorystyczne gwarantują zachowanie limitów czasowych w każdej sytuacji. Twarde systemy czasu rzeczywistego stosuje się tam, gdzie nawet drobne przekroczenia czasu reakcji mogą doprowadzić do awarii, lub wręcz katastrofy. Aby zagwarantować czas reakcji, rezygnuje się z wielu mechanizmów spotykanych w systemach ogólnego przeznaczenia.

Ze względu na rodzaj pobudzenia wyróżniamy dwa podstawowe typy systemów: pobudzane zdarzeniami oraz czasem. Systemy pobudzane zdarzeniami reagują asynchronicznie na zdarzenia w postaci przerwań sprzętowych. Systemy pobudzane czasem wykorzystują wewnętrzne przerwanie zegarowe i przeglądają zdarzenia okresowo, poprzez odpytywanie urządzeń. Metoda ta zapewnia z reguły większy determinizm czasowy, lecz może skutkować stratami danych, ze względu na synchroniczność działania. Współcześnie szeroko stosuje się systemy pobudzane zdarzeniami. Jest to metoda charakterystyczna dla systemów o łagodnych ograniczeniach.

System Phoenix powstał, aby zaspokoić zapotrzebowanie na prosty, rozproszony system operacyjny czasu rzeczywistego z wbudowanymi elementami tolerowania awarii. Awarie sprzętowe są niezwykle ważnym zagadnieniem w sterowaniu. Obecnie, aby zminimalizować prawdopodobieństwo awarii stosuje się aktywne zwielokrotnianie, bazujące na protokołach kontrolnych. System Phoenix ma w swoich założeniach wspierać algorytmy zapewniania niezawodności oraz wykrywania uszkodzeń sprzętowych.

2. Phoenix Loader

Aby uruchomić system operacyjny, niezbędne jest umieszczenie go w pamięci operacyjnej komputera. W tym właśnie celu, obok jądra systemu postał specjalny program ładujący, nazwany przez autora systemu Phoenix Loader (plo). Program ten pozwala na załadowanie jądra systemu Phoenix z jednego ze źródeł. Ostatnia wersja plo pozwala na załadowanie obrazu z komputera deweloperskiego (hosta) poprzez port

28 [PM], rozdział 3.3, strona 26

szeregowy, a także na załadowanie obrazu z dyskietki oraz dysku twardego (w tym z dysku CompactFlash).

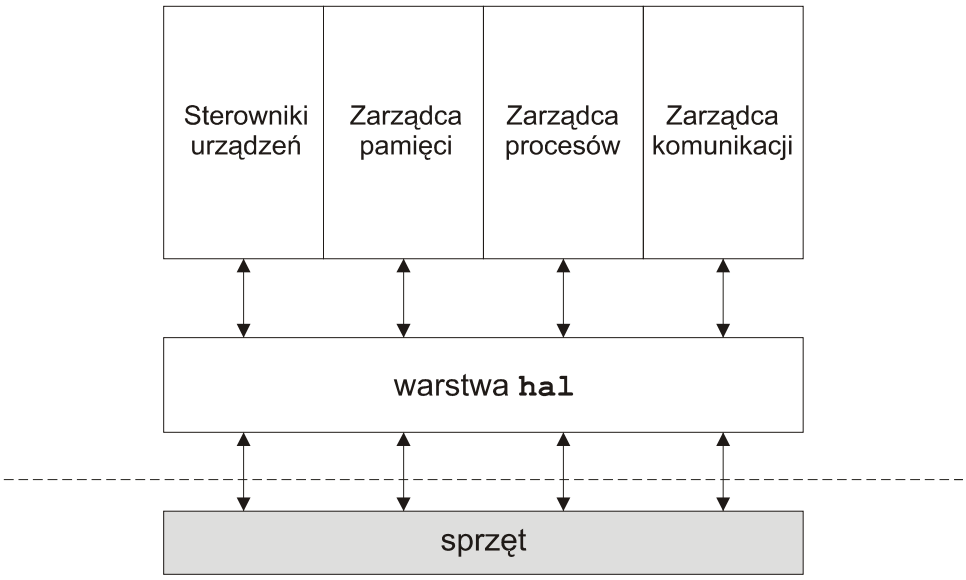
W celu ładowania systemu przez port szeregowy niezbędnym jest, po stronie plo, obsłużenie szeroko stosowanego kontrolera portu szeregowego 16550. Po stronie hosta wykorzystywana jest specjalna aplikacja rezydentna nazwana Phoenix Server (phoenixd). Do komunikacji między serwerem, a komputerem docelowym, wykorzystywany jest, oparty o przesyłanie wiadomości, prosty system plików nazwany Phoenix FileSystem (phfs). System wiadomości oraz phfs zaimplementowane są po obu stronach połączenia. Ten sam interfejs systemu plików wykorzystywany jest do pobierania pliku obrazu z dyskietki oraz z dysków twardych. Pozwala na skuteczne oddzielnie warstwy przetwarzania danych od warstwy fizycznej wymiany danych.

Program ładujący przyjmuje obraz jądra w formacie ELF. Jest on pobierany ze wskazanego źródła, a następnie ładowny w odpowiednie miejsce w pamięci. Następnie tak załadowane jądro można uruchomić z poziomu loadera.

Loader posiada wbudowaną funkcję autoboot pozwalającą na wykonanie zapisanych poleceń bez ingerencji operatora. Interaktywna praca z loaderem możliwa jest poprzez klasyczną klawiaturę, oraz klasyczną konsolę. Obsługa urządzeń peryferyjnych w loaderze realizowana jest głównie poprzez funkcje BIOSu.

3. Struktura mikrojądra i podsystemów Phoenix-RTOS[29]

Mikrojądro systemu Phoenix składa się z 4 podstawowych podsystemów – zarządcy pamięci, zarządcy procesów, zarządcy komunikacji oraz sterowników urządzeń. Podsystemu komunikują się z urządzeniami poprzez abstrakcyjną warstwę `hal` (ang. Hardware Abstraction Layer). Warstwa ta posiada ściśle zdefiniowany interfejs. Schemat struktury przedstawiony jest na rysunku 5:



Rys5. Struktura mikrojądra Phoenix[30]

W dalszej części scharakteryzuję krótko każdy z podsystemów.

29 Na podstawie [PM]

30 [PM], Rysunek 5.1, strona 36, uwzględniona nowa nazwa warstwy pośredniczącej.

a) Zarządzanie pamięcią.

Podsystem zarządzania pamięcią opracowany jest w oparciu o rozwiązania zastosowane w systemach Linux, Mach i BSD. Zrezygnowano tu jednak, między innymi, z wymiatania, kopiowania przy zapisie, czy mapowania plików w pamięci. System może zaalokować tyle, ile dostępnej jest pamięci fizycznej. Nie ma urządzenia wymiany. Było to konieczne ze względu na stawiane przed Phoenixem zadanie determinizmu czasowego. Mechanizmy zarządzania pamięcią są wielobieżne, ze względu na możliwość stosowania jądra na architekturach równoległych. W systemie zarządzania pamięcią możemy wyróżnić trzy główne warstwy:[31]

- część MD (ang. Machine Dependent), zrealizowana w postaci interfejsu `pmap` – udostępnia interfejs jednopoziomowej tablicy stron, umożliwia mapowanie pojedynczych stron pod konkretne adresy wirtualne,
- niskopoziomowe zarządzanie pamięcią, służące do przydzielania stron lub ich grup w oparciu o mapę stron, czyli liniową tablicę deskryptorów stron tworzoną w momencie uruchamiania jądra; grupa stron nazywana jest obszarem pamięci, natomiast sposób wyboru stron na rzecz obszaru jest zależny od jego przeznaczenia np. Alokacja na rzecz DMA, alokacja na rzecz jądra lub na rzecz procesu,
- warstwa dynamicznego przydziału pamięci w jądrze pozwalająca na alokowanie fragmentów mniejszych niż rozmiar strony – funkcje `kmalloc()` i `kfree()` – bardzo ważna warstwa z punktu widzenia używania dynamicznych struktur danych wewnątrz jądra i minimalizacji zużywanych przez nie zasobów.

Przestrzeń adresowa procesu

Przestrzeń adresowa procesu jest liniowa i składa się z segmentów użytkownika oraz z segmentu jądra. Segment definiowany jest przez autora jako spójny obszar pamięci wirtualnej, który charakteryzuje się określonym przeznaczeniem. Segmenty użytkownika mogą znajdować się na adresach poniżej 3GB, za wyjątkiem pierwszych 4KB, które zarezerwowane są celem obsługi odwołań do pamięci poprzez wskaźnik NULL.

Założono, że segment stosu zaczyna się od adresu 3GB. W przestrzeń adresową procesu wmapowany jest segment jądra, co umożliwia obsługę przerwań sprzętowych i wyjątków w kontekście procesu. Strony segmentu jądra chronione są przed dostępem z poziomu użytkownika. Strony jądra wmapowywane są jako globalne. Ze względu na wmapowywanie jądra pod adres 3GB, do takich właśnie adresów musi ono być skompilowane.

Zastosowane rozwiązanie układu przestrzeni adresowej znane jest z systemów Linux, BSD i Mach. Jest to ogólnie przyjęta tendencja, a rozwiązanie to jest optymalne i nie posiada znaczących ograniczeń.

Dynamiczna alokacja obszarów wolnych

Mechanizm ten pozwala zminimalizować zużycie pamięci przy alokacji obszarów drobnych, służących do przechowywania dynamicznych struktur danych jądra. Przydział realizowany jest za pomocą wywołania `kmalloc()`. Podobne rozwiązania wykorzystywane są w systemach Linux, BSD i Mach. Realizacja tego mechanizmu w

31 Opis warstw podsysytemu cytowany za [PM] (rozdział 6.3, strona 50)

systemie Phoenix zbliżona jest do realizacji z systemu Linux, która jest najbardziej przejrzysta ze wszystkich podanych przykładów. Mechanizm ten pozwala przydzielać obszary pamięci mniejsze od rozmiaru strony, co zapobiega marnowaniu całych stron przez przez małe struktury danych alokowane na rzecz jądra. Implementacja opiera się na tablicy `sizes[]`, która zawiera listy wolnych bloków o określonym rozmiarze (od 32 bajtów do 2 kilobajtów). Alokacja obszaru o zadanym rozmiarze sprowadza się do wybrania pierwszego elementu z listy wolnych bloków o określonym rozmiarze.

b) Obsługa przerwań i wyjątków

Procedury obsługi przerwań muszą spełniać wymagania narzucane przez główną cechę systemów czasu rzeczywistego, jaką jest determinizm czasowy:[32]

- procedury obsługi nie mogą blokować przerwań na długi czas – większa część procedury powinna wykonywać sie przy odblokowanych przerwaniach,
- nie jest dopuszczalne blokowanie przerwań na bliżej nieokreślony czas np. na czas przeglądania struktur o nieznanej długości,
- procedury obsługi powinny być napisane w sposób optymalny – w niektórych systemach (np. w systemie CMX) procedury obsługi pisane są całkowicie w języku maszynowym,
- w trakcie obsługi przerwania nie można wywoływać funkcji o nieznanym czasie działania np. Alokujących pamięć,
- należy unikać wywoływania jakichkolwiek funkcji, w większości przypadków procedura obsługi powinna przyjmować dane urządzenia do bufora lub informować system o jego gotowości.

Jądro udostępnia odpowiedni interfejs pozwalający na rejestrację funkcji obsługi, a także zapewnia komunikację z kontrolerem przerwań.

Wyjątki obsługiwane są analogicznie do przerwań, w tym jednak przypadku, w generacji nie bierze udziału kontroler przerwań, a jedynie procesor. Informuje on w ten sposób o zaistniałych sytuacjach błędnych, jak dostęp do chronionych obszarów pamięci, błędy dzielenia, czy braku strony. Wystąpienie wyjątku skutkuje wywołaniem odpowiadającej mu procedury obsługi.

Udostępniany interfejs obsługi przerwań współpracuje z zewnętrznym kontrolerem przerwań i8259. Dzisiejsze komputery PC wyposażone są w dwa kaskadowo połączone kontrolery, co pozwala na identyfikację 15 źródeł przerwań.

W obsłudze przerwań i wyjątków wydzielono dwie warstwy. Pierwszą część procedury stanowi niskopoziomowa namiastka przerwania lub wyjątku oraz, w przypadku przerwań procedura identyfikująca przerwanie (ang. Dispatcher)[33], drugą część natomiast sama funkcja obsługi przerwania, która nie jest zależna od procesora, a jedynie od rodzaju obsługiwanego urządzenia. Namiastka zrzuca kontekst i uruchamia dispatcher. Procedura ta przegląda funkcje obsługi przerwań, aby zaspokoić wszystkie oczekujące żądania. Przerwania zgłaszane są zboczem, co uniemożliwia ponowne wejście do funkcji obsługi, jeśli któreś z żądań nie zostało obsłużone.

32 Wymagania cytowane za [PM], rozdział 7, strona 58-59

33 Opis ten dotyczy aktualnej wersji systemu Phoenix-RTOS. W podsystemie tym zaszły istotne zmiany w porównaniu z wersją jądra z roku 2001.

Ze względy na zakładaną wielobieżność jądra konieczny jest podział na małe sekcje krytyczne. Przerwania wyłączane są tylko wtedy, kiedy jest to absolutnie konieczne. Blokowanie przerwań oraz w wejście do sekcji krytycznej realizowane jest poprzez mechanizm spinlocków.[34]

c) Zarządzenie procesami.[35]

W systemie Phoenix-RTOS zadania realizowane są w formie procesów i wątków. Struktura wątku składa się z części niezależnej od platformy oraz z zależnej od sprzętu części zawierającej kontekst procesora. Każdy proces i wątek w systemie posiadają unikalny identyfikator. Każdy proces podzielony jest na wątki.

Kontekst procesora pozwala powrócić do wykonywania danego zadania. Jest on umieszczany na stosie jądra zawsze po przejściu z trybu użytkownika w tryb systemowy.

Nowy proces ładowany jest do pamięci przy pomocy procedury `proc_load()`. Pozwala ona załadować ciało procesu z pliku w formacie ELF.

Szeregowanie zadań następuje na poziomie wątków. W podstawowej wersji systemu wykorzystywany jest algorytm planowania rotacyjnego[36] (RR, ang. round-robin).

Wszystkie procedury zostały napisane w taki sposób, aby możliwe było uruchamianie jądra na maszynach wieloporcesorowych.

d) Obsługa urządzeń[37]

Sterownik urządzenia powinien składać się z funkcji inicializującej dane urządzenie, z funkcji obsługi przerwania oraz funkcji usługowych danego urządzenia. Zrozumiałe jest, że sterownik powinien także alokować potrzebne struktury danych.

W aktualnej wersji system Phoenix-RTOS wyposażony jest w szereg sterowników. Są to między innymi sterowniki:

- klawiatury PC i konsoli,
- portu szeregowego 16550,
- kart sieciowych: RealTec RTL-8139, Intel EtherExpress 100, bazujące na stosie UDP/IP dla DPMI32, stworzonego w firmie IMMOS,
- graficzny dla Asilliant CT69000
- watchdogów firmy IEI oraz Advantech

Obsługiany jest interfejs szyny PCI.

34 Opis dotyczy aktualnej wersji systemu Phoenix-RTOS (patrz 33.)
35 Patrz 33.
36 Definicja algorytmu RR znajduje się w [PSO], rozdział 5.3.4, strona 161
37 Patrz 33.

V. Implementacja Phoenix-RTOS na platformie ARM

Implementacja została przeprowadzona przy użyciu zestawu uruchomieniowego ZL9ARM firmy BTC z układem mikrokontrolera dipARM wyposażonego w miktrokontroler Philips LPC2148. Zestaw uruchomieniowy zapewnia zasilanie, oscylatory kwarcowe, układy portów szeregowych w standardzie RS232, port USB, a także wyjście w postaci diod LED oraz wejście w postaci przycisków.

1. Systemy operacyjne dla procesorów ARM

Komputery oparte o procesor ARM mogą pracować pod kontrolą wielu systemów operacyjnych, w tym systemów typu Open Source i systemów eksperymentalnych. Ilość dostępnych systemów operacyjnych dla ARM prawdopodobnie przekracza liczbę systemów dla komputerów PC (IA32). Jednym z najbardziej popularnych systemów operacyjnych dla procesora ARM jest SymbianOS[38], używany w telefonach komórkowych firmy Nokia. Użycie systemu we własnych urządzeniach wymaga jednak zakupu odpowiedniej licencji. Inną popularna platformą systemową dla ARMa jest MS Windows CE[39]. System stał się w ostatnim czasie standardem dla komputerów typu handheld, wypierając powoli system PalmOS[40] (dzięki aplikacjom, wywodzącym się z komputerów PC). Licencja na ten system bardzo często jest dostarczana razem z platformami sprzętowymi.

W zastosowaniach niestandardowych i specjalizowanych bardzo popularną platformą systemową dla ARMa jest GNU/Linux[41]. Dostępny jest on na poziomie kodu źródłowego i jego używanie nie wymaga żadnych opłat licencyjnych. Przygotowanie zupełnie od podstaw (na podstawie kodu źródłowego systemu) własnej dystrybucji dla systemów jedno-układowych (SoC, ang. System-On-Chip) lub jedno-modułowych (SOM, ang. System-On-Module), wbrew pozorom, nie jest trudne. Dostępne jest szereg darmowych narzędzi, ułatwiających ten proces. Na uwagę zasługuje fakt, że używanie GNU/Linux nie wiąże się wcale z koniecznością kupowania drogich pakietów SDK i wsparcia od producenta danej platformy. Istnieją także wolne systemy BSD (NetBSD)[42] dla procesora ARM i systemów typu SoC. Na komputerach ARM działa także system czasu rzeczywistego QNX[43], modularny system operacyjny RedHat eCos[44], system do zastosowań krytycznych Green Hills Integrity[45] i szereg eksperymentalnych systemów operacyjnych.

W prostych mikrokontrolerach, jak wybrany do projektu Philips LPC2148, bardzo często stosowany jest właśnie system eCos.

2. Phoenix Loader (plo) na platformę ARM7TDMI

Program ładujący systemu Phoenix-RTOS wymagał gruntownej przebudowy, ze względu na jego niskopoziomowy charakter. Poczyniono założenie, że program ładujący niezbędny jest w stadium deweloperskim, ale ze względu na bardzo

38 Patrz [SYM]
39 Patrz [CE]
40 Patrz [PA]
41 Patrz [DEB]
42 Patrz [BSD]
43 Patrz [QNX]
44 Patrz [ECO]
45 Patrz [GHS]

ograniczone zasoby sprzętowe, w docelowym systemie nie należy umieszczać wydzielonego programu ładującego, a jedynie zredukować go do pewnych procedur inicjalizacyjnych jądra. Wersja programu ładującego dla platformy ARM przyjęła nazwę `arm-plo`.

a) Projekt.

Ze względu na specyfikę środowiska, czyli brak programu typu BIOS, loader w wersji dla architektury ARM musi przejąć, niezbędne do wykonywania powierzonych zadań, niektóre funkcje BIOSu. Co za tym idzie, program ładujący musi dokonać podstawowej inicjalizacji mikrokontrolera po starcie systemu.

Założenia projektu

Mikrokontrolery nie posiadają klasycznej konsoli, ani kontrolera klawiatury. Wszelka komunikacja musi zatem zachodzić za pośrednictwem portu szeregowe. W tym celu jeden z portów szeregowych mikrokontrolera musi zostać oddelegowany do obsługi interaktywnej komunikacji z operatorem.

Aby zagwarantować niezawodne działanie konsoli, zwłaszcza w fazie testowej, konieczne jest wykorzystywanie portu szeregowego w trybie aktywnego oczekiwania, aby zabezpieczyć się przed problemami z systemem przerwań.

Interfejs użytkownika programu ładującego musi być zgodny z interfejsem wersji dla IA-32. Dopuszcza się dodanie pewnych, specyficznych dla aplikacji funkcjonalności.

Interfejs programu ładującego zostanie rozszerzony o komendy `get` oraz `set` pozwalające na odczyt/zapis dowolnej komórki pamięci lub rejestru sterującego urządzenia. Interfejs ułatwi proces testowania i usuwanie problemów.

Program ładujący powinien zawierać komendę pozwalającą na zapis i ustawienie zegara czasu rzeczywistego.

Podstawowym zadaniem programu ładującego jest załadowanie jądra systemu Phoenix-RTOS do pamięci mikrokontrolera oraz jego uruchomienie. Interfejs ten musi być w pełni zgodny z aktualną wersją programu Phoenix Server, służącego do ładowania jądra poprzez port szeregowy.

Na potrzeby ładowania wykorzystany zostanie drugi port szeregowy. Port ten obsługiwany będzie z wykorzystaniem przerwań, przez sterownik portu szeregowego rodziny '550. Sterownik ten zostanie przygotowany na bazie istniejącego sterownika poprzez dokonanie niezbędnych modyfikacji.

Zakładamy, że jądro ładowane będzie pod ustalone w czasie jego kompilacji adresy. Po stronie przygotowania jądra pozostaje zapewnienie, aby pamięć używana przez obraz jądra nie pokrywała sie z pamięcią alokowaną przez program ładujący.

Dla celów metody autoboot konieczne jest oprogramowanie timera, aby generował wymagany przez funkcję timeoutu interwał czasowy.

Dla celów uruchomieniowych użyte zostanie wyjście na diody LED zestawu uruchomieniowego.

Loader musi przechowywać w pamięci nieulotnej dane konfiguracyjne funkcji autoboot. Przechowywany musi być czas oczekiwania, oraz komenda, mająca być po upływie tego czasu wykonana. W tym celu konieczne jest oprogramowanie systemu

IAP (ang. In-Application Programming) w celu umożliwienia zapisów w pamięci Flash.

Szczegółowe założenia konfiguracyjne

Zakładamy, że częstotliwość taktowania procesora zostanie ustawiona, przy pomocy układu PLL na 60MHz, a częstotliwość taktowania magistrali VPB na 15MHz.

Porty szeregowe działać będą z prędkością 115200 bodów, a przesyłane dane będą 8 bitowe, z jednym bitem stopu i bez kontroli parzystości. Taka prędkość oraz format ramki wymagane są przez Phoenix Server. Zakładamy wykorzystanie kolejki fifo z progiem ustawionym na 1 bajt aby zapewnić elementarne buforowanie i zabezpieczyć się przed gubieniem danych.

Interwał przerwania zegarowego ustala się na 10ms, aby zapewnić możliwie dużą rozdzielczość timerów programowych, przy jednoczesnym, znikomym obciążeniu systemu przerwaniami.

Założenia dotyczące kontrolera przerwań

Wykorzystany zostanie wbudowany kontroler przerwań mikrokontrolera (VIC, ang. Vectored Interrupt Controller). Kontroler będzie użyty w sposób opracowany przez autora pracy. Szczegóły sposobu użycia kontrolera zostaną podane rozdziale V.5, traktującym o obsłudze przerwań i wyjątków w jądrze systemu Phoenix-RTOS. Zasada wykorzystania VIC w loaderze jest analogiczna jak w jądrze. Różnice występują jedynie w podziale między warstwą niskopoziomową a wysokopoziomową.

W arm-plo mamy dwie warstwy funkcji obsługi. Pierwsza, napisana w języku maszynowym, warstwa odpowiedzialna jest za identyfikację źródła przerwania i wywołanie odpowiedniej, zarejestrowanej funkcji obsługi, która stanowi drugą, zależną tylko od obsługiwanego urządzenia, warstwę funkcji obsługi przerwania.

Obsługa wyjątków

W programie ładującym dla platformy ARM wykorzystywany jest prosty interfejs obsługi wyjątków sprzętowych. Składa się on ze statycznie rejestrowanych funkcji obsługi. Nie jest obsługiwane usuwanie przyczyn zaistniałego wyjątku, a jedynie dostarczana jest szczegółowa informacja o zaistniałej sytuacji wraz ze zrzutem kontekstu procesora.

b) Implementacja[46]

- ***Inicjalizacja podstawowych modułów***

Po starcie systemu sterowanie powinno zostać przekazane do funkcji obsługi restartu systemu. Funkcja ta powinna wykonać szereg podstawowych czynności inicjalizacyjnych. Tutaj powinien znaleźć się wspomniany wyżej kod przenoszący segment danych do pamięci SRAM. Po zainicjowaniu podstawowych podzespołów należy przejść do inicjalizacji pamięci. W pierwszej kolejności należy zainicjować stos dla wszystkich trybów pracy (tryby User i System używają jednego stosu). Należy przy tym pamiętać, że zgodnie z przyjętymi dla architektury AMR standardami wołania procedur, wykorzystywany jest stos rosnący w kierunku malejących adresów, wskaźnik stosu wskazuje na ostatni element, a adresy muszą być wyrównane do 4 bajtów. Po inicjalizacji stosu powinien znaleźć się, wspomniany wyżej, kod

46 Z wykorzystaniem informacji zawartych w [UM], [ARM], [PCS] oraz [GCC]

przenoszący segment danych do pamięci SRAM. Po wykonaniu tych czynności można przekazać sterowanie do funkcji `plo_init()`.

Pętla fazowa (PLL, ang. Phase Lock Loop)

Pierwszą czynnością jaką trzeba wykonać po starcie systemu jest inicjalizacja pętli fazowej. Odpowiednio ustawiony układ PLL pozwala na zwielokrotnienie częstotliwości zegara. Układ PLL konfigurowany jest przy pomocy zestawu rejestrów, do którego zaliczamy:

- rejestr sterujący (PLLCON, ang. PLL Control Register) – zawiera bity sterujące pracą pętli fazowej. Wszelkie zapisy do tego rejestru nie będą mieć wpływu na stan modułu przed odpowiednim wysterowaniem rejestru wprowadzającego;
- rejestr konfiguracyjny (PLLCFG, ang. PLL Configuration Register) – pozwala na wprowadzenie mnożnika i dzielnika dla pętli fazowej; podobnie jak przy rejestrze sterującym, zmiany muszą być potwierdzone
- rejestr statusu (PLLSTAT, ang. PLL Status Register) – jest to rejestr tylko do odczytu, pozwalający sprawdzić aktualny stan modułu PLL
- rejestr wprowadzający (PLLFEED, ang. PLL Feed Register) – jest to specjalny rejestr, na którym należy wykonać ściśle określoną sekwencję zapisów, aby wprowadzić w życie zmiany w rejestrach konfiguracyjnych.

Aby zmiany zostały wpisane do układu konieczne jest wykonanie sekwencji dwóch zapisów do rejestru PLLFEED. Najpierw należy zapisać wartość 0xAAh a następnie 0x55h. Te dwa zapisy muszą zostać wykonane w odpowiedniej kolejności, a co więcej, muszą być wykonane w dwóch, bezpośrednio następujących po sobie cyklach magistrali VPB. Drugie wymaganie sprawia, iż konieczne jest wyłączenie przerwań na czas wprowadzania zmian w pętli fazowej.

Przystępując do konfiguracji układu PLL musimy wykonać cztery podstawowe kroki. Po pierwsze wybieramy docelową częstotliwość taktowania procesora (CCLK). Wybór ten zależy w dużej mierze od wymagań używanych przez nas podzespołów, jak na przykład wymaganej szybkości pracy portu szeregowego. Należy pamiętać, że magistrala VPB może działać (i zazwyczaj działa) z mniejszą częstotliwością niż procesor. CCLK musi być większe od 10MHz i nie może przekroczyć maksymalnej częstotliwości dopuszczonej dla na danego modelu mikrokontrolera.

Następnie należy wybrać częstotliwość oscylatora kwarcowego (F_{OSC}). Częstotliwość taktowania (CCLK) musi być całkowitą wielokrotnością częstotliwości oscylatora. Na oscylator narzucone są także ścisłe ograniczenia. Częstotliwość drgań musi być w zakresie do 10MHz do 25MHz.

Aby przejść do kolejnego kroku konieczne jest zdefiniowanie dodatkowych symboli oraz zapoznanie się ze wzorami opisującymi zależności między zdefiniowanymi wielkościami. Obok wyjściowej częstotliwości taktowania (CCLK) i częstotliwości oscylatora kwarcowego (F_{OSC}), musimy brać pod uwagę zakres częstotliwości wewnętrznego, starowanego prądem, oscylatora pętli fazowej. Częstotliwość ta (określana dalej jako F_{CCO}), musi mieścić się w zakresie od 156MHz do 320MHz.

Do pracy pętli fazowej konieczne jest ustawienie modułowi dwóch wartości liczbowych: mnożnika częstotliwości wejściowej (M) oraz dzielnika częstotliwości (P).

Częstotliwość taktowania określona jest wzorami:

$$CCLK = M \times F_{OSC} \quad (1) \quad \text{oraz} \quad CCLK = \frac{F_{CCO}}{2 \times P} \quad (2)$$

Natomiast częstotliwość CCO można wyznaczyć ze wzorów:

$$F_{CCO} = CCLK \times 2 \times P \quad (3) \text{ oraz } \quad F_{CCO} = F_{OSC} \times M \times P \quad (4).$$

Znając powyższe wzory można przystąpić do do obliczenia mnożnika. Wartość M musi być dodatnia i nie większa od 32. Mnożnik obliczamy ze wzoru (1), natomiast w rejestrze PLLCFG, jako wartość pola mnożnika (MSEL) przyjmujemy M-1.

Następnie należy znaleźć wartość dzielnika, aby skonfigurować bity PSEL rejestru konfiguracyjnego. Podstawowym kryterium wyboru podzielnika (P) jest fakt, że częstotliwość F_{CCO} musi zawierać się w ustalonych granicach. Podzielnik może przyjmować tylko ściśle określone wartości: 1, 2, 4 lub 8. Wartości te zakodowane są na 2 bitach pola PSEL. Do wyznaczenia podzielnika wykorzystujemy wzór (3) podstawiając skrajne wartości F_{CCO}.

Dla przykładu układ którym dysponuję posiada oscylator kwarcowy o częstotliwości 12MHz. Założona przeze mnie częstotliwość taktowania wynosi 60MHz. Przystępując do konfiguracji układu PLL wyznaczamy mnożnik (M) ze wzoru (1):

$M = \frac{CCLK}{F_{OSC}} = \frac{60}{12} = 5$. Oznacza to, że na bity MSEL wpisujemy wartość 5-1=4. Następnie wyznaczamy P ze wzoru (3) podstawiając za F_{CCO} wartości odpowiednio 156MHz i 320MHz:

$$P > \frac{F_{CCO}^{MIN}}{2 \times CCLK} = \frac{156}{2 \times 60} = 1.3 \quad \text{i} \quad P < \frac{F_{CCO}^{MAX}}{2 \times CCLK} = \frac{320}{2 \times 60} = 2.67 \quad .$$

Jedyna dopuszczalna wartość P z tego zakresu to 2, czyli na bity PSEL wpisujemy wartość 01b.

Cały proces najlepiej zilustrować fragmentem kodu:

```
# INITIALISE PLL
pll_init:
                    /* Use r0 for indirect addressing */
                    ldr       r0, PLLBASE

                    /* PLLCFG = PLLCFG_VALUE */
                    mov       r3, #PLLCFG_VALUE
                    str       r3, [r0, #PLLCFG_OFFSET]

                    /* PLLCON = PLLCON_PLLE */
                    mov       r3, #PLLCON_PLLE
                    str       r3, [r0, #PLLCON_OFFSET]

                    /* PLLFEED = PLLFEED1, PLLFEED2 */
                    mov       r1, #PLLFEED1
                    mov       r2, #PLLFEED2
                    str       r1, [r0, #PLLFEED_OFFSET]
                    str       r2, [r0, #PLLFEED_OFFSET]

                    /* while ((PLLSTAT & PLLSTAT_PLOCK) == 0); */
pll_loop:
                    ldr       r3, [r0, #PLLSTAT_OFFSET]
                    tst       r3, #PLLSTAT_PLOCK
```

```
beq       pll_loop

/* PLLCON = PLLCON_PLLC|PLLCON_PLLE */
mov       r3, #PLLCON_PLLC|PLLCON_PLLE
str       r3, [r0, #PLLCON_OFFSET]

/* PLLFEED = PLLFEED1, PLLFEED2 */
str       r1, [r0, #PLLFEED_OFFSET]
str       r2, [r0, #PLLFEED_OFFSET]
```

Fragment pliku _startup.S

W załączonym fragmencie widzimy proces ustawiania pętli fazowej. W pierwszej kolejności ustawiane są odpowiednie wartości mnożnika i dzielnika, a układ PLL jest aktywowany. Następnie przeprowadzane są zapisy do rejestru PLLFEED, aby zaktualizować moduł PLL. Po wprowadzeniu zmian oczekujemy aktywnie, aż układ PLL ustabilizuje częstotliwość wyjściową na założonym poziomie. Kiedy to się stanie, podłączamy układ PLL, aby stał sie źródłem sygnału taktowania. Zmianę tę także potwierdzamy zapisami do rejestru PLLFEED.

Moduł akceleracji pamięci (MAM, ang. Memory Acceleration Module)

Moduł może pracować w jednym z trzech trybów:

- Tryb 0 – akceleracja wyłączona. Wszystkie odwołania do pamięci Flash prowadzą do odczytu. Instrukcje nie są pobierane z wyprzedzeniem
- Tryb 1 – akceleracja częściowo włączona. Odwołania sekwencyjne do kodu przyspieszane są przy pomocy zatrzasków modułu. Wszystkie skoki powodują odczyt z pamięci Flash. Wszystkie dostępy do danych powodują odczyt.
- Tryb 2 – pełna akceleracja. Wszystkie odwołania do kodu i danych przechodzą przez układ akceleracji

Do konfiguracji modułu służą dwa rejestry: rejestr sterujący (CR, ang. Control Register), oraz rejestr konfigurujący zależności czasowe (TIM, ang. Timing Register).

Przy użyciu rejestru MAMTIM możemy ustawić szacowaną ilość cykli zegara potrzebnych na odczyt z pamięci Flash. Pozwala to modułowi akceleracji dopasować swoje działanie do szybkości procesora. Możliwe jest ustawienie od 1 do 7 cykli zegara. Zalecana ilość cykli zależna jest od częstotliwości taktowania procesora. Przy taktowaniu rzędu 20MHz zaleca się jeden cykl, między 20MHz a 40MHz – 2 cykle, powyżej 40MHz – trzy. Przy 60MHz zdecydowałem się na ustawienie 4 cykli.

Rejestr MAMCR pozwala na przełączenie trybu pracy. Po włączeniu akceleracji moduł staje się niewrażliwy na zmiany rejestru MAMTIM. Aby ponownie skonfigurować moduł, należy go wcześniej zatrzymać wpisując 0 do rejestru MAMCR.

Poniżej zamieszczam przykładowy kod inicjujący MAM:

```
mam_init:
                  /* Use r0 for indirect addressing */
                  ldr       r0, MAMBASE
                  /* MAMTIM = MAMTIM_VALUE */
                  mov       r1, #MAMTIM_VALUE
                  str       r1, [r0, #MAMTIM_OFFSET]
                  /* MAMCR = MAMCR_VALUE */
                  mov       r1, #MAMCR_VALUE
                  str       r1, [r0, #MAMCR_OFFSET]
```

Fragment pliku _startup.S

- ***Kontroler przerwań (VIC, ang. Vectored Interrupt Controller)***

Zgodnie z przedstawionym powyżej opisem, kontroler przerwań wykorzystywany jest do identyfikacji źródła przerwania, a także do maskowania i konfiguracji przerwań.

Struktury danych

Podstawowym elementem jest globalna struktura typu `vic_data_t`:

```
typedef struct {
        vic_cntl_t vic_cntl;
        vu32 *irq_no;
        vu32 *irq_cntl;
        int (*irq_isr[16])(u16, void*);
        void* irq_data[16];
} vic_data_t;
```

Struktura ta zawiera wskaźnik na strukturę rejestrów sterujących kontrolera, wskaźnik na tablicę numerów przerwań (tablica VICVectAddrX), wskaźnik na tablicę kontrolną przerwań oraz tablicę funkcji obsługi i zarejestrowanych argumentów tych funkcji.

Do sterowania kontrolerem używana jest specjalna struktura `vic_cntl_t`:

```
typedef volatile struct{
        const vu32 irq_stat;
        const vu32 fiq_stat;
        const vu32 int_stat;
        vu32 int_sel;
        vu32 int_enable;
        vu32 int_enclr;
        vu32 soft_int;
        vu32 soft_intclr;
        vu32 protection;
} *vic_cntl_t;
```

Struktura ta pozwala przejść na wyższy poziom abstrakcji. Zasłania fizyczne adresy rejestrów kontrolnych. Wykorzystywana jest w ten sposób, że wskaźnikowi na strukturę tego typu przypisujemy adres bazowy kontrolera. Dalsze odwołania do rejestrów sterujących odbywają się poprzez odwołania do pól struktury. Takie rozwiązanie jest możliwe do zastosowania dzięki temu, że procesory ARM mają pojedynczą przestrzeń adresową, czyli odwołania do rejestrów sterujących urządzeń niczym nie różnią sie od innych odwołań do pamięci. Metoda ta, ze względu na funkcjonalność i znaczące zwiększenie czytelności kodu, stosowana jest powszechnie w tym projekcie dla sterowania urządzeniami wbudowanymi w mikrokontroler.

Implementacja funkcji sterujących kontrolerem

Interfejs kontrolera nie wymaga aktywnej inicjalizacji. Jedyną wymaganą formą jest zainicjowanie wartości wskaźników w globalnej strukturze `vic_data`. Interfejs kontrolera przerwań składa się z trzech czterech funkcji. Najważniejszą i centralną dla systemu funkcją jest niskopoziomowa procedura obsługi przerwania:

```
#
# IRQ handler
#
          .func irq_handler
irq_handler:
          sub       lr, lr, #4
          stmfd     sp!, {r0-r12, lr}

          mvn       r11, #4032          /*prepare VICVectAddr address*/
          sub       r11, r11, #15       /* 0xffff ff030 */
          ldr       r2, ih_itab         /* get isr table pointer */
          ldr       r1, ih_dtab         /*get isr data table pointer*/
          ldr       r0, [r11]           /*get irq number (isr arg 1)*/
          ldr       r3, [r2, r0, lsl #2] /* get isr address */
          ldr       r1, [r1, r0, lsl #2] /* get isr data (isr arg 2)*/

          add       lr, pc, #4          /*call isr(irq_no, irq_data)*/
          bx        r3


          mov       r2, #255            /* VICVectAddr = 0xff */
          str       r2, [r11]           /* update priority hardware  */
          ldmfd     sp!, {r0-r12, pc}^ /* return from interrupt */
ih_itab:            .word     vic_data+4+4+4
ih_dtab:            .word     vic_data+4+4+4+64

          .endfunc
```

Fragment pliku _startup.S

Funkcja ta pobiera z kontrolera numer przerwania. Uzyskany numer służy do indeksowania tablicy funkcji obsługi i argumentów. Następnie wywoływana jest odpowiednia funkcja obsługi z przygotowanymi parametrami. Po powrocie z funkcji obsługi, uaktualniany jest sprzętowy układ ustalania priorytetów i następuje powrót z przerwania.

Do konfiguracji kontrolera stosowane są funkcje `low_irqinst()` oraz `low_irquninst()`. Funkcje te, odpowiednio, rejestrują i wyrejestrowuję obsługę przerwania od określonego urządzenia. Aby poprawnie skonfigurować obsługę konkretnego przerwania należy przekazać funkcji `low_irqinst()` informacje o numerze źródła przerwania oraz o wybranym numerze przerwania. Przekazać też należy wskaźnik do funkcji obsługi oraz w wskaźnik do argumentu tej funkcji.

Po rejestracji przerwania identyfikowane są po wybranym numerze. Numer określa także priorytet przerwania. Najwyższy priorytet ma przerwanie o najniższym numerze.

Interfejs kontrolera wyposażony jest także w funkcję `low_irqen()`, która pozwala włączać bądź wyłączać przerwanie o danym numerze.

Prototyp funkcji obsługi wygląda następująco:

```
int irq_isr(u16 no, void* data);
```

- ***Diody LED***

Diody LED sterowane są z pierwszego portu wejść/wyjść cyfrowych ogólnego przeznaczenia. Do obsługi tego portu przygotowana została specjalna struktura danych `gpio_t`:

```
typedef struct {
        vu32 pin;
        vu32 set;
        vu32 dir;
        vu32 clr;
}* gpio_t;
```

Diody sterowane są za pomocą funkcji `low_dbg()`. Pozwala ona wyświetlić na diodach wartość 8-bitową. Inicjalizacja wymaga ustawienia pinów sterujących diodami jako wyjść, poprzez ustawienie składowej (`gpio_t)->dir` wartością 0x00FF0000.

- ***Programowanie pamięci Flash***

Program użytkowy może zapisywać pamięć Flash wykorzystując do tego celu interfejs IAP (ang. In-Application Programming). Dla celów projektu przygotowany został prosty interfejs korzystający z procedur IAP w celu zapisywania danych w pamięci Flash. IAP udostępnia także możliwość odczytania przechowywanych przez bootloader danych identyfikujących sprzęt. Interfejs modułu `iap` składa się z trzech funkcji. Dwóch służących do zapisywania i kasowania zawartości pamięci: `iap_copyto()` i `iap_erase()`, oraz z funkcji pobierającej identyfikator urządzenia: `iap_partId()`.

Funkcja `iap_erase()` przyjmuje dwa parametry: początkowy adres pamięci Flash oraz długość obszaru do skasowania. Kasowanie pamięci odbywa się sektorami, w związku z czym funkcja `iap_erase()` wylicza na podstawie argumentów w którym sektorze (lub sektorach) leży wskazywany obszar i kasuje ten sektor (sektory). Zostało przyjęte, że długość obszaru do skasowania lub zapisu nie może przekraczać 4KB.

Funkcja `iap_copyto()` przyjmuje trzy argumenty: adres docelowy w pamięci Flash, adres źródłowy bufora w pamięci RAM, oraz długość bufora do skopiowania. Wymagane jest aby adres w pamięci Flash był wyrównany do 256 bajtów. Długość kopiowanego obszaru może wynosić 256, 512, 1024 lub 4096 bajtów. Ograniczenia te wprowadzane są przez procedury IAP i ze względu na maksymalne uproszczenie implementacji, propagują do interfejsu użytkownika. Aby zapis do pamięci powiódł się wymagane jest wcześniejsze wyczyszczenie sektorów które są zapisywane. Skutek nadpisania danych w pamięci Flash jest nieprzewidywalny. Kasowanie należy wykonać przy pomocy funkcji `iap_erase()`. Nie zostało ono włączone do funkcji kopiującej aby umożliwić dopisywanie, do wcześniej skasowanego sektora, kolejnych danych.

Funckja `iap_partId()` jest bezargumentowa. Zwraca pobrany przez IAP identyfikator mikrokontrolera, który pozwala określić wersję z jaką mamy do czynienia.

Wszystkie funkcje interfejsu `iap` zwracają kod błędu. Kod ten jest sumą logiczną błędów zwracanych przez poszczególne, używane w funkcji, polecenia `IAP`.

Funkcje modułu `iap` zaimplementowane są z wykorzystaniem wspominanego już interfejsu IAP. Interfejs ten opiera się na znajdującej się pod adresem `0x7ffffff1` procedurze wejściowej (`IAP_ENTRY`). Procedura ta przyjmuje dwa argumetny, będące wskaźnikami na tablice argumentów (wejściowych i wyjściowych). Procedura przyjmuje maksymalnie 5 paramntrów wejściowych i zwraca maksymalnie 3

parametry wyjściowe. Dla zwiększenia czytelności kodu zostały zdefiniowane struktury określające dane wejściowe i wyjściowe procedury `IAP_ENTRY`:

```
typedef struct {
        u32     code;
        union {
                u32   start;
                void* dst;
        };
        union {
                u32   end;
                void* src;
        };
        u32     count;
        u32     cclk;

} iap_input_t;

typedef struct {
        u32 status;
        union {
                u32 offset;
                u32 partId;
                u32 version;
        };
        u32 content;
} iap_return_t;
```

Moduł `iap` ceduje większość ograniczeń interfejsu `IAP` na użytkownika. Jest to uzasadnione, gdyż moduł nie jest szeroko stosowany w loaderze i nie było powodu, aby opracowywać wygodny i wydajny interfejs do jego obsługi. Powstały moduł ma za zadanie jedynie przykryć niskopoziomowe algorytmy współpracy z `IAP` jak skomplikowane struktury danych czy ściśle określone sekwencje poleceń.

- ***Timer***

Przy ustawianiu timera kluczową rolę odgrywa główny rejestr sterujący (TCR, ang. Timer Control Register). Jest on odpowiedzialny za uruchamianie liczników oraz za ich zerowanie. Przed przystąpieniem do konfiguracji należy timer zatrzymać. Tryb pracy licznika określa rejestr sterujący zliczaniem (CTCR, ang. Count Control Register). Pozwala on ustawić jeden z czterech trybów:

- tryb timera (ang. Timer Mode), w którym zliczane są wszystkie narastające zbocza zegara magistrali
- trzy tryby licznika (ang. Counter Mode), w których zliczane są odpowiednio narastające, opadające lub obydwa zbocza wybranego wejścia. Wybór wejścia dokonywany w tym samym rejestrze. Pole wyboru wejścia zliczania (ang. Count Input Select) pozwala wybrać jedno z czterech wejść.

W dalszej części skoncentruję się na pracy w trybie timera, jako na bardziej interesującej z punktu widzenia projektowanego oprogramowania. Kolejnym elementem konfiguracji jest uzyskanie interesującego nas interwału czasowego. Timer mikrokontrolera działa dwustopniowo. Wyposażony jest w preskaler dzielący

częstotliwość zegara magistrali, a sam licznik timera inkrementowany jest przez ten preskaler. Do konfiguracji preskalera służy rejestr preskalera (PR, ang. Prescale Register), w którym ustawiamy graniczną wartość, po osiągnięciu której następuje inkrementacja głównego licznika i zerowanie preskalera. Pozwala to uzyskać zadowalający nas interwał czasowy inkrementacji licznika.

Podstawowym sposobem wykorzystania timera jest używanie generowanych przez ten układ przerwań. Aby uzyskać przerwania w określonych odstępach czasu, co jest potrzebne i pożądane w każdym systemie operacyjnym, wykorzystuje się mechanizm rejestrów zgodności. Każdy timer posiada 4 rejestry zgodności (MR, ang. Match Register) pozwalające na wykonanie określonej akcji w momencie, gdy wartość licznika (TC, ang. Timer Counter) równa jest wartości danego rejestru MR. Sposób reakcji na zrównanie określa się w rejestrze sterującym zgodnością (MCR, ang. Match Control Register). Poprzez ustawienie odpowiednich bitów, dla danego rejestru MR, możemy uzyskać jedną z trzech akcji: zgłoszenia przerwania, zerowania licznika, lub zatrzymania licznika. Pozwala to na łatwą konfigurację interwału czasowego przerwań od timera. Rejestr prescalera dostarcza nam pewnej jednostkowej wartości skoku (na przykład 1ms), a jeden z rejestrów MR pełni rolę mnożnika tej wartości. W celu uzyskania określonej częstotliwości przerwań, zerujemy timer wartością naszego rejestru mnożnika, a przerwanie zgłaszamy bądź tym samym rejestrem, bądź innym rejestrem MR ustawionym najlepiej na 0.

Po dokonaniu ustawień możemy uruchomić timer wpisując wartość 1 do rejestru TCR.

Identyfikację źródła przerwania zapewnia rejestr przerwań (IR, ang. Interrupt Register). Pozwala on stwierdzić który kanał (MR0-3, CR0-3) zgłasza przerwanie. Rejestr ten służy jednocześnie do zdejmowanie obsłużonego przerwania. Aby wyzerować przerwanie pochodzące z danego kanału, należy ustawić odpowiadający mu bit rejestru IR. Przy inicjalizacji należy pamiętać o wyczyszczeniu rejestru identyfikacji przerwań (poprzez zapisanie do niego wartości 0xFFh).

Inicjalizację timera przeprowadzamy już z poziomu języka C, gdyż nie wymaga ona żadnych specjalnych operacji. Aby obliczyć, potrzebną do uzyskania określonego interwału bazowego, wartość rejestru preskalera należy wziąć pod uwagę fakt, że timer zlicza zbocza zegara magistrali VPB (PCLK), który jest z reguły podwielokrotnością zegara procesora. Wartość przeskalowania określona jest w rejestrze sterującym dzielnikiem magistrali VPB (VPBDIV, ang. VPB Divider Control). Rejestr ten może przyjmować jedną z trzech możliwych wartości: 00b – zegar procesora dzielony przez 4, 01b – zegar magistrali taki sam jak procesora, 10b – zegar procesora dzielony przez 8. Z reguły zegar magistrali ustawia się na jedną czwartą zegara procesora.

Dla przykładu wyznaczmy wartość preskalera dla timera użytego w projekcie. Częstotliwość bazowa jaką chcemy uzyskać to 1kHz (okres 1ms). Częstotliwość zegara systemowego wynosi 60MHz, jak wcześniej zostało ustalone. Jeżeli podzielnik ustawiony jest na jedną czwartą, to częstotliwość magistrali VPB wynosi 15MHz. Wartość preskalera obliczamy z prostego wzoru:

$PR = \frac{PCLK}{F_{BASE}} = \frac{15\,MHz}{1\,kHz} = 15000 = 0x3A98h$. W tym momencie uzyskujemy timer, którego główny licznik inkrementowany jest dokładnie co 1ms. Następnie ustawiamy rejestr MR na wartość oczekiwane interwału w ms i mamy gotowy system zgłaszania przerwania zegarowego. Przykładowo, jeżeli chcemy uzyskać częstotliwość 100Hz,

jaka z reguły stosowana jest systemach operacyjnych, to ustawiamy odpowiedni rejestr MR na 10.

Struktury danych

Dla wygody oraz poprawienia czytelności kodu, do operacji na rejestrach sterujących timera używana jest specjalnie przygotowana struktura `timer_regs_t`. Implementacja funkcji wysokopoziomowych timera opiera się na oryginalnej implementacji z `plo`. Po oryginale odziedziczona została struktura `timer_t` używana do przechowywania licznika "momentów" (ang. jiffies). Dodatkowo struktura ta przechowuje wskaźnik na rejestry sterujące timera. Struktury te zdefiniowane zostały następująco:

```
typedef struct {
        vu32    ir;
        vu32    tcr;
        vu32    tc;
        vu32    pr;
        vu32    pc;
        vu32    mcr;
        vu32    mr[4];
        vu32    ccr;
        vu32    cr[4];
        vu32    emr;
        vu32    ctcr;
}* timer_regs_t;

typedef struct {
        timer_regs_t cntl;
        u16 jiffies;
} timer_t;
```

Fragment pliku timer.c

Implementacja funkcji

Interfejs timera składa się z czterech funkcji:

`timer_init()` to bezargumentowa funkcja inicjalizująca timer. W funkcji tej ustawiane są odpowiednie rejestru sterujące oraz instalowana jest funkcja obsługi przerwania. Interwał przerwań ustawiany jest na 1ms. Przerwanie timera trzymało numer 0 dla zapewnienia najwyższego priorytetu. W loaderze nie ma to dużego znaczenia, ale w jądrze systemu jest to istotne. W tym miejscu zero zostało wybrany raczej ze względu na konwencję jak na rzeczywiste wymogi priorytetów.

Funkcja `timer_wait()`[47] została w całości odziedziczona z oryginalnego loadera i nie wymagała żadnych modyfikacji. Przyjmuje cztery argumenty: czas oczekiwania (ms), flagi konfiguracyjne, wskaźnik na monitorowaną zmienną, wartość początkowa zmiennej. Funkcja ta pozwala wstrzymać wykonanie programu na czas przekazany w pierwszym argumencie lub do wystąpienia jednego z monitorowanych zdarzeń. Zakończenie działania może nastąpić w jednym z trzech przypadków:

- skończył się okres oczekiwania podany w argumencie pierwszym,
- wciśnięto klawisz przy ustawionej fladze `TIMER_KEYB`,

47 Funkcja zapożyczona z wersji dla IA-32

- zaobserwowano zmianę wartości wskazywanej przez trzeci argument, przy ustawione fladze `TIMER_VALCHG`

`timer_isr()` to procedura obsługi przerwania timera. Jako argument funkcja pobiera otrzymuje wskaźnik na strukturę `timer_t` przechowującą dane obsługiwanego timera. Jej zadaniem inkrementacja licznika "momentów" oraz zerowanie przerwania.

Funkcja `timer_done` służy do zakończenia działania, czyli zatrzymania licznika i wyrejestrowania przerwania od timera.

• *Port szeregowy*

Kontroler portów szeregowych LPC2100 zgodny jest z kontrolerami rodziny '550, jak znany z komputerów PC 16550. Jedyna istotna różnica to fakt, że adresu rejestrów kontrolera LPC wyrównane są do 4 bajtów. Dla zaoszczędzenia przestrzeni adresowej, rejestry sterujące kontrolera współdzielą między sobą adresy. Rozróżnienie, o który rejestr chodzi, dokonywane jest na dwa sposoby, albo poprzez charakter operacji (zapis/odczyt), jak w przypadku rejestrów wysyłania i odbioru, albo poprzez sprawdzenie stanu bitu DLAB (ang. Divisor Latch Access Bit) znajdującego się w rejestrze sterowania linią. Ustawienie bitu DLAB pozwala na dostęp do rejestrów dzielników częstotliwości, blokując jednocześnie dostęp do rejestrów wysyłania/odbioru oraz włączania przerwań.

Do konfiguracji kontrolera używane jest 5 rejestrów:

- Rejestr sterowania linią (LCR, ang. Line Control Register) – odpowiedzialny jest za ustawianie bitu DLAB, jak i za ustawianie parametrów transmisji: długości znaku (5, 6, 7 lub 8 bitów), ilości bitów stopu (1 lub 2), sprawdzania bitu parzystości, rodzaju parzystości, przerywania transmisji.
- Para rejestrów podzielnika częstotliwości (DLM:DLL, ang. Divisor Latch MSB:LSB) – odpowiedzialne są za szybkość transmisji danych.
- Rejestr sterowania kolejką FIFO (FCR, ang, FIFO Control Register) – pozwala włączyć kolejki nadajnika i odbiornika, wyczyścić ich zawartość, oraz ustawić próg zgłaszania przerwania odbioru danych.
- Rejestr maski przerwań (IER, ang. Interrupt Enable Register) – umożliwia włączenie lub wyłączenie poszczególnych źródeł przerwań kontrolera.

Domyślnie linie RxD i TxD portów szeregowych nie są połączone z wyprowadzeniami mikrokontrolera. Konieczne jest ustawienie funkcji alternatywnych dla pinów odpowiednio: P0.0 i P0.1 dla UART0 oraz P0.8 i P0.9 dla UART1. Każde wyprowadzenie może przyjąć jedną z maksymalnie czterech funkcji. W przypadku portu szeregowego jest to funkcja 01b. Ustawienie funkcji alternatywnych dla pierwszych 16 pinów portu P0 dokonywane jest poprzez rejestr rejestr wyboru funkcji 0 (PINSEL0, ang. Pin Function Select Register 0).

Przy konfiguracji portu szeregowego kluczową rolę odgrywa poprawne ustawienie prędkości. Służą do tego wspomniane wyżej rejestry dzielników częstotliwości. Wymaganą wartość podzielnika możemy otrzymać ze wzoru:

$$UART_{BAUDRATE} = \frac{PCLK}{16 \times UDLM : UDLL} \text{ , czyli: } UDLM : UDLL = \frac{PCLK}{16 \times UART_{BAUDRATE}} ,$$

gdzie PCLK – częstotliwość taktowania magistrali VPB (w Hz), $UART_{BAUDRATE}$ – oczekiwana prędkość (w bodach).

Po ustawieniu prędkości można wyczyścić bit DLAB i ustawić pożądany format transmisji. Przeważnie interesować nas będzie znak 8-bitowy, z jednym bitem stopu i bez sprawdzania parzystości, gdyż dla większości standardowych transmisji takie ustawienia są najlepsze.

Należy także rozważyć włączenie kolejek FIFO. Buforowanie jest z reguły pożądane, gdyż zabezpiecza nas przed ewentualnymi stratami nieodebranych danych oraz pozwala zwiększyć wydajność systemu minimalizując ilość przerwań.

Pozostaje jeszcze ustawienie maski przerwań. Przeważnie interesować nas będzie przerwanie dostępności danych do odczytu (ang. Receive Data Available) oraz możliwości zapisu do bufora (kolejki) nadajnika (ang. Transmit Holding Register Empty).

Jak przy inicjalizacji wszystkich urządzeń wykorzystujących przerwania, tak i przy porcie szeregowym niezbędne jest wyczyszczenie rejestru identyfikacji przerwań (IIR, ang. Interrupt Identification Register), aby umożliwić start całego mechanizmu w przypadku rekonfiguracji w trakcie działania systemu. Czyszczenie rejestru identyfikacji przerwań następuje poprze odczyt jego zawartości.

Korzystanie z prostego dzielnika częstotliwości sprawia, że dokładność ustalenia prędkości pracy kontrolera, zależy silnie od częstotliwości taktowania magistrali. Jeżeli szybkość transmisji nie jest całkowitą podwielokrotnością częstotliwości taktowania magistrali podzielonej przez 16, to nieunikniony jest błąd. Z reguły błąd ten jest niewielki i nie ma wpływu na jakość transmisji, jednak, jeśli uznamy, że błąd jest zbyt duży, możemy go zminimalizować stosując ułamkowy dzielnik częstotliwości, w jaki wyposażony jest kontroler. Dzielnik ten wprowadza dwie dodatkowe wartości: dzielnik (DivAddVal) i mnożnik (MulVal). Jeżeli dzielnik równy jest 0, to korekcja nie jest wprowadzana. W innym przypadku, do powyższego wzoru na prędkość transmisji, należy dołożyć czynnik korygujący. Wzór ten przyjmuje wtedy postać: $UART_{BAUDRATE} = \frac{PCLK}{16 \times DLM:DLL} \times \frac{MulVal}{MulVal + DivAddVal}$.

Obliczenie na podstawie tego wzoru wartości DLM:DLL, MulVal i DivAddVal nie jest oczywiste. Wymaga stosowania metody prób i błędów aby znaleźć rozwiązanie minimalizujące błąd.

Dla przykładu, jeżeli mam częstotliwość PCLK równą 15MHz i planujemy użyć kontrolera z prędkością 115200 bodów, to bez użycia korekcji ułamkowej, wyliczamy ze wzoru wartość DLM:DLL: $DLM:DLL = \frac{15 \cdot 10^6}{16 \times 115200} = 8.138 \approx 0x00:08h$.

Wpisujemy do rejestru DLM – 0, a do rejestru DLL – 8, ale otrzymujemy błąd równy 1.73%. Jeżeli przeprowadzimy analizę metodą prób i błędów, to możemy znaleźć zestaw danych: DLL = 6, MulVal = 14, DivAddVal = 5. Przy takich ustawieniach błąd maleje do 0.06%. Korekcja jest trudna do stosowania, gdyż wartości nie zmieniają się liniowo wraz ze zmianami prędkości, jak to ma miejsce przy prostym dzielniku całkowitoliczbowym.

Z faktu zgodności z układami '550 wynika możliwość bazowania na sterowniku portu szeregowego z loadera dla IA-32. Jedyne wymagane zmiany dotyczą konfiguracji.

Trzeba uwzględnić wspomniany wyżej fakt różnic w adresowaniu rejestrów sterujących oraz uwzględnić konieczność konfiguracji wyjść mikrokontrolera.

Struktury danych

Podobnie jak w poprzednich przypadkach, dla zwiększenia czytelności kodu i wygody użytkowania, została opracowana struktura danych rejestrów sterujących:

```
typedef struct{
        union{                  /* 0x00 */
                vu32    rbr;
                vu32    thr;
                vu32    lsb;
        };
        union{                  /* 0x04 */
                vu32    imr;
                vu32    msb;
        };
        union{                  /* 0x08 */
                const vu32 iir;
                vu32    fcr;
        };
        vu32            lcr;    /* 0x0C */
        const vu32      pad1;
        vu32            lsr;    /* 0x14 */
        const vu32      pad2[4];
        vu32            fdr;    /* 0x28 */
}* serial_regs_t;
```

Ze względów praktycznych nie zostały w tej strukturze ujęte, nieużywane, rejestry kontroli automatycznego strojenia prędkości.

Część niezależna od konfiguracji została przejęta z wersji dla IA-32. Wykorzystuje on strukturę danych `serial_t`. Jedyną modyfikacją tej struktury jest zamiana adresu bazowego na wskaźnik struktury rejestrów sterujących:

```
typedef struct {
        u16 active;
        serial_regs_t cntl;
        u16 irq;
        u8      src;
        u8 rbuff[RBUFFSZ];
        u16 rb;
        u16 rp;
        u8 sbuff[SBUFFSZ];
        u16 sp;
        u16 se;
} serial_t;
```

Rozmiary buforów zostały ustalone na 256 bajtów (oryginalnie rozmiar wynosił 2048 bajtów).

Funkcje

Jak w przypadku innych urządzeń możemy wyróżnić funkcje inicjalizujące, procedurę obsługi przerwania oraz funkcje usługowe interfejsu użytkownika.

Inicjalizacja została przeprowadzona zgodnie z opisem znajdującym się powyżej. Procedura obsługi przerwania oraz funkcje usługowe zostały zaadaptowane ze sterownika dla maszyn z IA-32. Jedyną konieczną zmianą była zmiana sposobu odwołań do rejestrów, jako, że w architekturze ARM mamy jedną przestrzeń adresową i do odwołania do rejestrów w przestrzeni I/O niczym nie różnią sie od zwykłych odwoła do pamięci, jak ma to miejsce w architekturze IA-32.

Procedura obsługi przerwania pobiera znak z odbiornika i wstawia go do kolejki lub pobiera znak z kolejki i wstawia do rejestru nadajnika. Obsługa na poziomie użytkownika opiera się na funkcjach `serial_read()` i `serial_write()`, których działanie zgodne jest z przyjętymi konwencjami dla tego typu funkcji. Działają one analogicznie do funkcji `read` i `write` znanych z biblioteki standardowej. Pierwszym argumentem tych funkcji jest wirtualny numer portu szeregowego, następnie podajemy adres bufora i jego długość. Funkcja `serial_read()` jest blokująca. Z tego powodu posiada dodatkowy argument (`timeout`) pozwalający określić czas, po który funkcja zakończy działanie z błędem, jeśli nie uda się jej odebrać żadnych danych.

c) Testowanie

Dla celów testowych przygotowano specjalne komendy interfejsu loadera.

W komendzie `info` wyświetlany jest kontekst procesora. Może to posłużyć do monitorowania wskaźnika stosu, zawartości rejestrów oraz kontroli trybu pracy i stanu przerwań.

Komenda `dump` pozwala wyświetlić zawartość dowolnego fragmentu pamięci. Jest to niezwykle przydatne przy kontrolowaniu stanu zmiennych i zawartości określonych obszarów pamięci. Opcja ta okazała się niezwykle przydatna przy implementacji zapisywania danych do pamięci Flash. Dzięki podglądowi pamięci udało się ustalić, że skutek zapisu danych do niewyczyszczonego wcześniej sektora jest nieokreślony.

Komendy `get` i `set` pozwalają odczytać lub ustawić dowolną komórkę pamięci. Mechanizm może zostać wykorzystany do wstrzykiwania danych testowych do struktur danych. Dzięki możliwości ręcznego zapisywania komórek pamięci udało się odnaleźć przyczynę błędu w systemie przerwań. Okazało się, że funkcja inicjalizująca timer nie zeruje rejestru identyfikacji przerwań. Powodowało to paraliż systemu przerwań po rozpoczęciu wykonania programu od adresu zero bez przejścia przez stan Reset.

Do celów testowania konsoli wprowadzona został także komenda `send`, która umożliwiała przysyłanie terminalowi sekwencji sterujących generując sekwencję CSI lub ESC. Pozwoliło to na skrupulatne sprawdzenie działania sekwencji sterujących terminalem w standardzie ECMA-48. Okazało się, że są problemy z obsługą emulacji VT100 w programie Hyper Terminal systemu Windows. Problem polega na tym, że po ustawieniu przewijania tak, aby pozostawić górne wiersze, przewijane jest tylko 40 kolumn zamiast 80. Problem ten nie występuje w stosowanym w projekcje emulatorze Minicom. Przy wykorzystaniu komendy `send` możliwe było także sprawdzenie, która z dwóch sekwencji zapamiętywania i odtwarzania kursowa działa poprawnie. Okazało się że sekwencja CSIs / CSIl nie działa. Konieczne jest korzystanie z sekwencji ESC7 / ESC8

Testowanie poprawności działania systemu obsługi przerwań i wyjątków przeprowadzono uruchamiając przerwanie z linii zewnętrznej. Sprawdzano jak system zachowuje się, jeśli źródło przerwania nie zostanie zdjęte. System zachowywał się poprawnie, wchodząc tylko raz w obsługę przerwania.

Testowanie portu szeregowe przeprowadzono przez implementację funkcji echo oraz poprzez wypisywanie kodów znaków przesyłanych do mikrokontrolera. Ujawniło to problem z gubieniem znaków przy obsłudze bez wykorzystania przerwań. Problem został rozwiązany poprzez uruchomienie kolejek FIFO kontrolera.

Przy implementacji ładowania obrazu jądra w formacie ELF wykorzystywane było wpisywanie zawartości odczytanych nagłówków pliku. Pozwalało to kontrolować proces ładowania poprzez sprawdzeniem czy odpowiednie sekcje pliku zostały załadowane pod wskazane adresy. Jednocześnie, uruchomieniem pobranego jądra można było skontrolować poprawność wskaźnika wejściowego. Istotne było, czy kompilator ustawił najmłodszy bit wskaźnika na funkcję `_start()`, aby zaznaczyć, że mamy do czynienia z funkcją implementowaną w trybie Thumb.

3. Zakres zmian w jądrze Phoenix-RTOS

Zgodnie z założeniami projektu Phoenix-RTOS, system ten ma wyraźnie wydzielą warstwę zależną od sprzętu (`hal`, ang. Hardware Abstraction Layer). Warstwa ta zapewnia spójny, niezależny od platformy, interfejs dla wyższych warstw systemu.

Przeniesienie systemu na nową platformę powinno ograniczać się do zmian w warstwie zależnej od sprzętu, poprzez jej dostosowanie.

W warstwie `hal` znajduje się inicjalizacja systemu, obsługa konsoli systemowej, obsługa przerwań i wyjątków, definicje funkcji zależnych od architektury procesora, konfiguracja systemu zarządzania pamięcią, oraz definicje spinlocków i obsługa przerwania zegarowego.

Inicjalizacja

W warstwie `hal` znajduje się plik `_init.S`, który zawiera procedury inicjalizacyjne systemu. W pliku tym zdefiniowane jest wejście (ang. entry point) do systemu. Procedura `_start()`, która jest pierwszą wykonywaną po starcie jądra, przekazuje sterowanie do funkcji `main()`.

Konfiguracja

Warstwa zależna od sprzętu udostępnia interfejs konfigurujący podstawowe podzespoły systemu, czyli system obsługi przerwań i wyjątków, spinlocki oraz konsolę systemową. Inicjalizacja podstawowych podzespołów powinna zostać wykonany w funkcji `_hal_init()`, która jest pierwszą procedurą wołaną z funkcji `main()`.

Zarządzanie pamięcią

Osobnym problemem jest inicjalizacja podsystemu zarządzania pamięcią. Warstwa `hal` udostępnia interfejs jednopoziomowej tablicy stron (`pmap`). Strony w tablicy mogą przyjmować dwa atrybuty: wolna lub zaalokowana dla jądra. Interfejs ten definiuje także rozmiar strony. Dla procesora IA-32 ustalono rozmiar strony na 4 KB.

4. Inicjalizacja sytemu

a) Projekt

Start systemu

Założono, że system rozpoczyna działanie od funkcji `_start()`. Funkcja ta ma za zadanie przygotować system do przekazania sterowania funkcji `main()`.

Jądro powinno mieć możliwość samodzielnego startu z pamięci Flash, a co za tym idzie, obraz powinien zaczynać się od wektorów przerwań. Wersja testowa uruchamiana jest przez loader z pamięci SRAM.

Uruchamianie z pamięci SRAM wymaga przygotowania wektorów przerwań. Muszą one zostać przekopiowane do odpowiedniej lokalizacji. Wektory muszą znajdować się na początku pamięci SRAM. Pierwsze 64 bajty pamięci SRAM może zostać przemapowane na początek przestrzeni adresowej, co pozwala przygotować w tym obszarze wektora przerwań. Kod wykorzystywany przy budowaniu wektora musi być niezależny od położenia w pamięci.

W związku z brakiem konieczności wykorzystywania loadera w przyszłości, jądro, po starcie, musi przeprowadzić pełną inicjalizację. Należy przeprowadzić konfigurację pętli fazowej oraz jednostki akceleracji pamięci. Przed oddaniem sterowania do funkcji `main()` konieczna jest także inicjalizacja stosu.

Podobnie jak w przypadku loadera ustalono, że częstotliwość taktowania rdzenia wynosić będzie 60MHz, natomiast częstotliwość taktowania magistrali peryferyjnej będzie wynosić 15MHz. System ma działać z pełną akceleracją pamięci.

Spinlocki

Spinlocki wykorzystywane są przez jądro systemu Phoenix-RTOS do tworzenia sekcji krytycznych w kodzie. Ustawienie spinlocka zapewnia po pierwsze zablokowanie przerwań, a po drugie, zapewnia synchronizację wątków wykorzystujących z tego samego spinlocka.

Konsola systemowa

Konsola jest jednym z najważniejszych komponentów, gdyż pozwala na komunikację z użytkownikiem. Mikrokontroler nie jest wyposażony w graficzną konsolę. Konieczna była całkowita przebudowa istniejącej obsługi.

Założono, że funkcję konsoli przejmie jeden z portów szeregowych mikrokontrolera. Aby zapewnić elementarną kontrolę nad terminalem szeregowym postanowiono wykorzystać emulator terminala VT100. Wymaga to na kontrolerze konsoli korzystanie ze standardu ECMA-48[48]. Standard wykorzystywany jest do przemieszczania kursora i zmiany koloru wyświetlanego tekstu. Kontrolowane jest także przewijanie tekstu.

Założono, że konsola musi być wizualnie w pełni kompatybilna z oryginalną konsolą systemu Phoenix-RTOS.

48 Opis standardu znajduje się w [ECM]. Inne nazwy standardu to: ISO/IEC 6429 lub ANSI X3.64

b) Implementacja

Funkcje inicjalizujące

Funkcja `_start()`, po wyłączeniu przerwań, przechodzi do wywołania funkcji inicjalizujących podstawowe podzespoły. Funkcje konfigurujące pętlę fazową oraz moduł akceleracji pamięci wywoływane są przed inicjalizacją stosu, a co za tym idzie nie zapewniają zachowania stanu rejestrów zgodnie z konwencją wołania procedur.

`pll_init()` konfiguruje pętlę fazową (PLL) zgodnie z poczynionymi założeniami. Kod wykorzystywany w tym celu zapożyczony jest, w niezmienionej formie, z loadera.

Procedura `mam_init()` odpowiedzialna jest za konfigurację modułu akceleracji pamięci (MAM). Podobnie jak w przypadku pętli fazowej, kod jest zapożyczony z loadera.

`stack_init()` ustawia początkowe wartości wskaźników stosu dla poszczególnych trybów. W tym miejscu także wykorzystywany jest kod stworzony dla loadera.

Dla zapewnienia inicjalizacji zmiennych globalnych zerami, przed przejściem do kodu C konieczne jest wyzerowanie segmentu `.bss`. Wykonywane jest to z pomocą procedury `bss_init()`.

Funkcje inicjalizujące zebrane są w pliku `_hal.S`.

Kolejnym etapem jest przygotowanie wektorów przerwań. Jądro może zostać załadowane pod dowolne adresy. Zadaniem tego fragmentu konfiguracji jest zapewnienie, że wektory przerwań znajdą się na początku przestrzeni adresowej. W tym celu wektory te, wraz z tablicą adresów funkcji obsługi, kopiowane są do pierwszych 64 bajtów pamięci SRAM. Następnie wykorzystywany jest rejestr kontroli mapowania (MEMMAP). Ustawiany jest tryb 2, czyli “User RAM”. Oznacza to, że pierwsze 64 bajty pamięci SRAM wmapowane są na początek przestrzeni adresowej.

Po przygotowaniu stosu, wektorów przerwań oraz po skonfigurowaniu podstawowych podzespołów, sterowanie zostaje przekazane do funkcji `main()`. Funkcja ta znajduje się w warstwie sprzętowo niezależnej, a co za tym idzie korzysta z interfejsu dostarczanego przez warstwę `hal` w celu przeprowadzenia dalszych procedur konfiguracyjnych.

Spinlocki

Blokada realizowana jest przy pomocy instrukcji typu “test and set”. W architekturze ARM taka instrukcją jest SWP, która umożliwia atomową wymianę zawartości rejestru i komórki pamięci. Instrukcja ta niedostępna jest w trybie Thumb, toteż do założenia lub zdjęcia blokady konieczne jest chwilowe przejście do trybu ARM. Sposób wykonania takiego przejścia ilustrują kody źródłowe funkcji `hal_spinlockSet()` oraz `hal_spinlockClear()`:

```
void hal_spinlockSet(spinlock_t *spinlock)
{
 __asm__ volatile
         (" \
                  .align 2;\
                  .thumb;\
                  bx        pc;\
                  nop;\
                  .arm;\
```

```
                mrs     r2, cpsr;\
                orr     r1, r2, #(0x80 | 0x40);\
                msr     cpsr_c, r1;\
        1: \
                mov     r1, #0;\
                swp     r1, r1, [%1];\
                cmp     r1, #0;\
                beq     1b;\
                str     r2, [%0];\
                \
                add     r2, pc, #1;\
                bx      r2;\
                .thumb"
        :
        : "r" (&spinlock->psr), "r" (&spinlock->lock)
        : "r1", "r2" );
        hal_cpuGetCycles((void *)&spinlock->b);
}


void hal_spinlockClear(spinlock_t *spinlock, sop_t sop)
{
        hal_cpuGetCycles((void *)&spinlock->e);

        if (sop == sopGetCycles) {

            /* Calculate maximum lock time */
            if ((cycles_t)(spinlock->e - spinlock->b) >
                                              spinlock->dmax) {
                    spinlock->dmax = spinlock->e – spinlock >b;
            }

            /* Calculate minimum lock time */
            if (spinlock->e - spinlock->b < spinlock->dmin)
                 spinlock->dmin = spinlock->e - spinlock->b;
        }

   __asm__ volatile
        (" \
                .align 2;\
                .thumb;\
                bx      pc;\
                nop;\
                .arm;\
                mov     r0, #1;\
                swp     r0, r0, [%0];\
                ldr     r0, [%1];\
                msr     cpsr_all, r0;\
                \
                add     r0, pc, #1;\
                bx      r0;\
                .thumb"
                :
                : "r" (&spinlock->lock), "r" (&spinlock->psr)
                : "r0");

        return;
}
```

Fragment pliku spinlock.c

Wysokopoziomowa część implementacji spinlocków przeniesiona została w niemal nie zmienionej formie.

Konsola systemowa

Konsola wykorzystuje pierwszy port szeregowy mikrokontrolera. Ze względu na konieczność bezwarunkowego działania konsoli w każdych warunkach zdecydowano, że nie będzie wykorzystywać przerwań. Zwiększa to przydatność konsoli w przy testowaniu i uruchamianiu systemu. Wykorzystywany jest interfejs sterowania portem szeregowym opracowany dla loadera.

Do obsługi przewijania i kolorowania tekstu, konsola wykorzystuje standard ECMA-48. Zgodnie ze standardem, wykorzystywane są specjalne sekwencje sterujące.

Inicjalizacja konsoli wydziela w pierwszym jej wierszy specjalny pasek statusu. Reszta konsoli jest przewijana. Wykorzystywane są trzy kolory tekstu: biały – dla napisów informacyjnych, czerwona – dla zgłaszania błędów oraz zielony – dla informacji debuggingowych.

Interfejs konsoli składa się z czterech funkcji:

Funkcja `_hal_consoleInit()` służy do inicjalizacji konsoli. Konfigurowany jest port szeregowy. Ustawiany jest na prędkość 115200 bodów oraz na format 8-bitowy z jednym bitem stopu i bez kontroli parzystości. Dla większej dokładności, prędkość dostrajana jest przy pomocy ułamkowego dzielnika częstotliwości. Funkcja ta, po skonfigurowaniu portu, wysyła do terminala sekwencję konfiguracyjną. Sekwencja ta czyści terminal, ustawia przewijanie od drugiego wiersza i pozycjonuje kursor na początku drugiej linii. Wysyłana sekwencja wygląda następująco: `CSI "7h" CSI "2J" CSI "2;25r" CSI "2;0H"`.[49]

`hal_consoleClear()` czyści konsolę przywracając stan po inicjalizacji.

Funckaj `hal_consolePrint()` służy do wypisywania ciągu znaków na konsolę. Pierwszym argumentem jest typ napisu (info/debug/error) warunkujący kolor wysyłanego na terminal tekstu.

`hal_consoleStatus()` pozwala umieszczać napisy na jednym z trzech pól lini statusu. Funckaj ta wyskakuje kursorem do określonej pozycji w linii statusu, czyści zawartość danego pola i nadpiuje go nową wartościa, a następnie wraca do pozycji wyjściowej. Cała operacja wykonywana jest w sekcji krytycznej, gdyż jej przerwanie mogłoby poważnie uszkodzić konsolę.

5. Obsługa przerwań i wyjątków

a) Projekt

Kontroler przerwań (VIC, ang. Vectored Interrupt Controller)

Po gruntownych przemyśleniach, zmieniono na potrzeby projektu sposób wykorzystania kontrolera przerwań. W założeniu wektor adresów funkcji obsługi miał posłużyć do bezpośredniego pobierania adresu funkcji obsługi przy pomocy rejestru VICVectAddr. Metoda ta wymaga jednak, aby funkcje obsługi były implementowane w specyficzny sposób. Funkcja musiała być implementowana w trybie `ARM` i nie mogła przyjmować żadnych argumentów, ani zwracać wartości. Okazało sie to dużym problemem, gdyż bardzo utrudniało modularyzację systemu. Problem ten udało się

49 CSI to początek sekwencji sterującej. Składa się z dwóch znaków: ESC i '[' (łańcuch znakowy: `"\x1B["`)

rozwiązać poprzez wykorzystanie możliwości kontrolera w sposób odmienny od wizji jego projektantów.

Postanowiono wykorzystać wektor adresów funkcji obsługi do przechowywania numeru przerwania (zgodnego z numerem wektora). Pozwala to przechowywać funkcje obsługi we własnych strukturach danych i wywoływać je z określonymi parametrami. Wymagało to przygotowania pewnej infrastruktury i struktur danych. Możliwości kontrolera wykorzystywane są do identyfikacji numeru przerwania, a także do priorytetowania i włączania przerwań od poszczególnych urządzeń. Szczegóły implementacji tego mechanizmu znajdują się w rozdziale dotyczącym implementacji.

Założenia dotyczące obsługi przerwań.

Założono, że używane będą jedynie przerwania wektorowane. Wynika z tego, iż możliwe jest skonfigurowanie 16 urządzeń jako źródeł przerwań. Wszystkie przerwania zgłaszane są jako IRQ, za pośrednictwem kontrolera VIC.

Przerwanie zapamiętuje i przekazuje do funkcji obsługi kontekst procesora. Przed powrotem z przerwania kontekst jest przywracany z zapamiętanych wartości. Pozwala to na implementację przełączania kontekstu. Obsługa przerwań dokonywana jest na trzech poziomach funkcji obsługi: przyjęcie przerwania, identyfikacja źródła, obsługa.

Przerwania zgłaszane są poziomem. Fakt ten sprawia, że nie ma konieczności przeglądania wszystkich funkcji obsługi na okoliczność oczekujących przerwań, jak miało to miejsce w oryginalnym systemie.

Założenia dotyczące obsługi wyjątków.

Procesory ARM zgłaszają trzy wyjątki sprzętowe. Obsługa tych wyjątków jest w dużej mierze analogiczna do obsługi przerwań. Funkcja obsługi sytuacji wyjątkowej zapamiętuje kontekst wyjątku i przekazuje go dalej, do funkcji identyfikacji wyjątku, a następnie do zarejestrowanej procedury obsługi.

b) Implementacja

Inicjalizacja kontrolera przerwań.

Z perspektywy rejestrów sterujących kontrolera inicjalizacja jego pracy składa się z trzech etapów.

Po pierwsze, korzystając z rejestru wyboru przerwań (VICInstSelect, ang. Interrupt Select Register) należy zdecydować, czy któreś z przerwań powinno być zgłaszane na linii FIQ. Nie zaleca sie wybierania więcej niż jednego przerwania, aby zminimalizować czas jego obsługi.

Drugim etapem jest włączenie przerwań od interesujących nas urządzeń. Dokonać tego można poprzez modyfikację rejestru włączania przerwań (VICIntEnable, ang. Interrupt Enable Register).

Kolejnym krokiem jest przydzielenie poszczególnych źródeł przerwań do slotów kontrolera. Jak opisane było wcześniej, każde z 32 źródeł przerwań może być przypisane do jednego z 16 slotów. Przypisania dokonuje się poprzez modyfikację rejestrów sterujących poszczególnych slotów (VICVectCntlx, ang. Vector control x register). Rejestr ten posiada dwa pola. 5-bitowy numer źródła przerwania, oraz bit określający, czy dany wpis jest ważny. Jednocześni do odpowiednich rejestrów

adresowych (VICVectAddrx, ang. Vector address x register) wpisywany jest numer przerwania (taki sam jak numer rejestru).

Dodatkowo należy ustawić rejestr domyślnego adresu funkcji obsługi, najlepiej na wartość łatwo identyfikowalną, jak na przykład 0xffh. Jest to potrzebne, gdy planujemy używać przerwań niewektorowanych. Nawet jeżeli nie planujemy używania tego typu przerwań, to prawidłowa ich obsługa jest niezbędna, aby radzić sobie z przerwaniami zgłaszanymi w wyniku sytuacji błędnej. Problem pojawia się w przypadku, gdy zgłoszone zostaje przerwanie, ale przed przystąpieniem do obsługi program użytkownika zmodyfikuje kontroler przerwań w sposób wyłączający dane przerwanie. Wtedy rejestr adresowy zwróci domyślny adres funkcji obsługi, czyli w naszym przypadku wartość 0xffh.

Obok standardowej konfiguracji, możliwe jest także ustawienie specjalnego rejestru kontroli dostępu (VICProtection, ang. Protection Enable register). Ustawienie bitu 0 w tym rejestrze sprawia, że rejestry sterujące kontrolera dostępne są tylko z trybów uprzywilejowanych.

Kontekst procesora

Kontekst procesora, zapamiętywany i odtwarzany przez obsługę przerwania, definiuje struktura:

```
typedef struct {
        u32 savesp;
        u32 psr;
        u32 reg[15];
        u32 ret;
} cpu_context_t;
```

Znajdziemy w niej zapamiętany wskaźnik stosu jądra na którym odłożony jest kontekst. Dalej znajduje się pełny zestaw rejestrów trybu użytkownika (oczywiście z pominięciem PC) wraz z rejestrem statusu procesora. Na końcu struktury znajduje się adres powrotu do procesu (po zakończeniu przerwania.

Kontekst wyjątku jest bardzo podobny:

```
typedef struct {
        u32 psr;
        u32 reg[15];
        u32 ip;
        u32 ret;
} exc_context_t;
```

Obok pól znanych z kontekstu procesora, pojawia się tutaj jeszcze adres instrukcji, która spowodowała wyjątek.

Obsługa przerwań

Przy obsłudze przerwań wykorzystywane są specjalnie przygotowane struktury danych. Jest wśród nich, znana a loadera, struktura sterująca kontrolera `vic_cntl_t`. Centralną strukturą całego modułu jest `interrupts`:

```
struct {
        vic_cntl_t vic_cntl;
        vu32 *irq_no;
        vu32 *irq_cntl;
        vu32 *irq_uexp;
        spinlock_t spinlocks[SIZE_HANDLERS];
        intr_handler_t handlers[SIZE_HANDLERS];
        unsigned int counters[SIZE_HANDLERS];
} interrupts;
```

Równie istotna jest struktura `intr_handler_t` przechowująca zarejestrowaną funkcję obsługi wraz ze wskaźnikiem na dane będące drugim argumentem funkcji:

```
typedef struct _intr_handler_t {
        int (*f)(unsigned int, cpu_context_t *, void *);
        void *data;
} intr_handler_t;
```

Jak wspominano w założeniach, obsługa przerwań opiera się na trzech warstwach:

Pierwszą warstwę obsługi przerwania stanowi wejście procedury obsługi przerwania. Jest ono uniwersalne, niezależne od źródła przerwania. Centralnym elementem tej warstwy jest napisana w języku maszynowym funkcja obsługi:

```
ENTRY(_irq_handler)
4.      sub     lr, lr, #4
5.      str     lr, [sp, #-4]!
6.      stmfd   sp, {r0-r14}^
7.      sub     sp, sp, #0x3c
8.      mrs     r4, spsr
9.      str     r4, [sp, #-4]!
10.     sub     sp, sp, #4
11.     str     sp, [sp]
12.     mvn     r11, #4032   /* prepare VICVectAddr address */
13.     sub     r11, r11, #15 /* 0xffff ff030 */
14.     ldr     r0, [r11]  /* get irq number (isr arg 1) */
15.     mov     r1, sp    /* cpu context pointer (isr arg 2) */
16.
17.     cmp     r0, #0xff
18.     ldrne   r3, _dispatcher
19.     ldreq   r3, _unexpected
20.     mov     lr, pc    /* call isr(irq_no, cpu_context) */
21.     bx      r3
22.     mov     r2, #255  /* VICVectAddr = 0xff */
23.     str     r2, [r11]  /* update priority hardware*/
24.     ldr     sp, [sp]
25.     add     sp, sp, #4
26.     ldr     r4, [sp], #4
27.     msr     spsr_fsxc, r4
28.     ldmfd   sp, {r0-r14}^
29.     add     sp, sp, #0x3c
30.     ldmfd   sp!, {pc}^         /* return from interrupt */

_dispatcher:       .word    interrupts_dispatchIRQ
_unexpected:       .word    interrupts_dispatchUnexpected
```

W funkcji tej możemy wyróżnić trzy fazy. Faza pierwsza to zapamiętanie kontekstu procesora na stosie. Obejmuje ona instrukcje 1-8. Faza druga polega na pobraniu z kontrolera numeru przerwania oraz przygotowaniu argumentów wywołania funkcji identyfikującej źródło przerwania. Faza ta obejmuje wiersze instrukcje 9-18 i

zakończona jest wywołaniem funkcji identyfikującej źródło (ang. IRQ Dispatcher). Faza trzecia to uaktualnienie układu ustalania priorytetów kontrolera oraz przywrócenie kontekstu. Obejmuje ona wiersze 19-27 i skutkuje powrotem z przerwania. Funkcję ta znajdziemy w pliku `_interrupts.S`.

Drugą warstwę obsługi przerwania stanowi dispatcher. Funkcja ta wykorzystuje przekazany numer przerwania celem wybrania odpowiadającej mu procedury obsługi. Procedury obsługi przerwań są rejestrowane przez użytkownika i są niezależne od platformy. Zależne są tylko od urządzenia którym sterują. Funkcja obsługi przerwania przyjmuje dwa argumenty: wskaźnik na zapamiętany kontekst procesora oraz wskaźnik na dane zarejestrowany dla danej funkcji obsługi.

Definicja dispatchera oraz pozostałych funkcji interfejsu obsługi przerwań znajdują się w pliku `interrupts.c`.

Trzecią warstwę stanowi zarejestrowana przez użytkownika i niezależna od platformy, procedura obsługi przerwania.

Obsługa wyjątków

Wyjątki sprzętowe obsługiwane są w podobny sposób jak przerwania. Wykorzystywana jest tutaj prosta struktura danych `exceptions`, przechowująca zarejestrowane funkcje obsługi oraz spinlock do wykonywania operacji atomowych:

```
struct {
        void *handlers[SIZE_EXCHANDLERS];
        spinlock_t lock;
} exceptions;
```

Podobnie jak przy obsłudze przerwań, można wyróżnić trzy warstwy funkcji obsługi.

Warstwę pierwszą stanowi stub assemblerowy. Ujednolica obsługę wyższych warstw, gdyż wszystkie trzy wyjątki wołają wspólny dispatcher z odpowiednimi parametrami, czyli przygotowanym kontekstem i numerem wyjątku.

Stub funkcji obsługi wygląda następująco:

```
#define EXCSTUB(exc)\
                    str       lr, [sp, #-4]!;\
                    sub       lr, lr, #4;\
                    str       lr, [sp, #-4]!;\
\
                    stmfd     sp, {r0-r14}^;\
                    sub       sp, sp, #0x3c;\
                    mrs       r4, spsr;\
                    str       r4, [sp, #-4]!;\
                    tst       r4, #0x20;\
                    subne     lr, lr, #2;\
                    strne     lr, [sp, #0x44];\
\
                    mov       r0, #(exc);\
                    mov       r1, sp;\
\
                    ldr       r3, _handlers;\
                    ldr       r3, [r3, r0, LSL #2];\
                    mov       lr, pc;\
                    bx        r3;\
\
                    ldr       r4, [sp], #4;\
                    msr       spsr_fsxc, r4;\
                    ldmfd     sp, {r0-r14}^;\
\
                    add       sp, sp, #4;\
                    ldmfd     sp, {pc}^
```

Przy jego pomocy zdefiniowane są trzy funkcje obsługi, do których przekazywane jest sterowanie po nadejściu wyjątku. Funkcje te zdefiniowane są w pliku `_exceptions.S`.

Wspólny dispatcher stanowi drugą warstwę obsługi. Jego zadaniem jest zawołanie zarejestrowanej funkcji obsługi dla danego wyjątku. Przy inicjalizacji do wszystkich wyjątków przypisywana jest prosta, domyślna funkcja obsługi.

Funkcja ta wyświetla na konsoli kontekst wyjątku i zawiesza wykonanie, nie powracając z obsługi. Jest to proste rozwiązanie pozwalające często wykryć przyczynę wystąpienia wyjątku, którą z reguły jest błąd w implementacji. Funkcja ta jest bardzo pomocna przy testowaniu i uruchamianiu.

Funkcje interfejsu obsługi wyjątków umieszczone są w pliku `exceptions.c`.

6. Podsystem zarządzania pamięcią

a) Projekt

System przygotowywany był na procesor wyposażony w jednostkę zarządzania pamięcią (MMU, ang, Memory Management Unit). W tym wypadku nie do końca sprawdziła się pełna niezależność warstw wyższych. W systemie bez sprzętowej jednostki zarządzania pamięcią nie mamy wsparcia dla mapowania przestrzeni wirtualnej w fizyczną. System musi operować na adresach fizycznych, a rola interfejsu zarządzania pamięcią sprowadza się do mapowania adresów na takie same.

Projekt zmian w interfejsie `pmap`

Interfejs pmap w założeniu udostępnia abstrakcję jednopoziomowej tablicy stron. W związku z brakiem stronicowania funkcjonalność interfejsu została znacznie

zmniejszona. Większość funkcji wymagała znaczącego uproszczenia, a często strywializowania.

Zakładany jest podział pamięci fizycznej na strony bez wirtualizacji adresów. Rozmiar strony został zmniejszony z 4KB do 512 bajtów ze względu na niewielką ilość pamięci w urządzeniu.

Obszary pamięci fizycznej dzielone są na wolne, zarezerwowane oraz zaalokowane dla jądra.

Początkowo strony mogą być wolne, bądź zaalokowane dla jądra.

W związku z brakiem jednostki zarządzania pamięcią niemożliwe jest wprowadzenie żadnych mechanizmów ochrony pamięci.

Projekt zmian w podsystemie `vm`

Założono, że alokator obszarów drobnych może maksymalnie zaalokować 2KB, czyli 4 strony. Oznacza to 3 elementową tablicę kolejki stron. Wielkość tej tablicy określa jaki maksymalny spójny obszar pamięci można zaalokować za jej pomocą. Z racji faktu, że stałe definiujące powyższe wartości znajdują sie w podsystemie `vm`, niezbędna okazała się modyfikacja tego podsystemu.

b) Implementacja

Interfejs `pmap`

Interfejs ten składa się z siedmiu funkcji:

- `_pmap_init()` – inicjalizuje interfejs; jej zadaniem jest określenie granicznych adresów pamięci RAM oraz sterty jądra. Przy braku stronicowania jest to jedyna wymagana inicjalizacja.
- `_pmap_kspaceInit()` – funkcja ta alokowała tablice stron dla przestrzeni jądra. Przy braku stronicowania funkcja ta nie wykonuje żadnych czynności.
- `pmap_getpage()` – służy do pobierania informacji o stronie do jakiej odnosi się podany adres fizyczny. W wersji dla bez MMU jej zadanie sprowadza się do sprawdzenia, czy adres leży wolnej przestrzeni pamięci RAM, czy w przestrzeni w jakikolwiek sposób zarezerwowanej. W obrębie pamięci SRAM zarezerwowane są strony w których rezyduje jądro, pierwsza strona w pamięci, gdzie rezydują wektory przerwań oraz syspage, a także ostatnia strona pamięci, gdzie umieszczone są początkowe wskaźniki stosu, oraz rezydują procedury IAP.
- `pmap_resolve()` – funkcja ta zwraca adres fizyczny odpowiadający danemu adresowi wirtualnemu. W wersji bez stronicowania funkcja ta jest trywialna. Zwraca ten sam adres który otrzymała.
- `pmap_enter()` – służyła do mapowania strony na podany adres. W wersji bez stronicowania funkcja ta jest pusta.
- `pmap_switch()` – funkcja służyła do przełączania przestrzeni adresowych. W obecnej wersji jest pusta.
- `pmap_create()` – funkcja tworzyła pustą tablicę stron. W wersji bez stronicowania jedynie przepisuje adresy graniczne.

Zmiany w podsystemie `vm`

W podsystemie tym niezbędna była redefinicja stałych. Zmniejszono do trzech ilość poziomów w tablicy kolejek wolnych stron stron (`SIZE_VM_FREE`), do ośmiu rozmiar startowej puli stron (`SIZE_VM_IPOOL`). Zmniejszono także do pięciu maksymalny indeks kolejki alokatora obszarów drobnych (`SIZE_KMALLOC_AREAQ`), aby kolejka mieściła się w obrębie strony.

Konieczne było także strywializowanie funkcji `vm_kmap()`, która mapowała stronę na wolny adres wirtualny. Obecnie zwracany jest adres fizyczny podanej strony.

Zmiany w podsystemie `vm` zostały wprowadzone w sposób nie zakłócający oryginalnej wersji. W tym celu została zdefiniowana specjalna stała `ARCH_ARM7`, która została użyta do warunkowej redefinicji stałych i trywializacji funkcji `vm_kmap()`.

7. Pozostałe moduły i dalszy rozwój projektu.

W tym miejscu kończy się proces przenoszenia systemu na nową platformę. Pozostałe podsystemy należą w całości do warstwy niezależnej od platformy i jako takie nie powinny wymagać modyfikacji. Aby pokazać, że tak jest w istocie, uruchomiony został podsystem `proc` odpowiedzialny za wątki i przełączanie zadań.

a) Przełączanie kontekstu.

Przetwarzanie danych w systemie Phoenix-RTOS opiera się na wątkach. Cały interfejs podsystemu zarządzania zadaniami jest z założenia niezależny od platformy. Wszystkie elementy zależne od platformy, jak kontekst procesora, jego zapamiętywani i odtwarzanie, przerzucone są na odpowiednie funkcje warstwy `hal`. Są to:

- struktura danych `cpu_context_t` opisująca kontekst procesora,
- funkcja `hal_cpuCreateContext()` tworząca pusty kontekst dla nowego wątku,
- `hal_cpuRestoreContext()` przełączająca na kontekst nowego wątku, oraz
- `hal_cpuReschedule()` oznaczająca zrzekanie się procesora i uruchamiająca scheduler.

Przejściowe zmiany w podsystemie `proc`

W związku z faktem, iż nie został uruchomiony podsystem systemu plików `fs` konieczne było wprowadzenie tymczasowych modyfikacji podsystemu `proc`. Niezbędne okazało się wyłączenie funkcji ładujących nowe wątki z pliku.

b) Sterowniki urządzeń.

Sterowniki urządzeń znajdują się w podsystemie `dev`. Są one w zasadzie niezależne od platformy, a jedynie od rządzenia którym sterują. Potencjalnym problemem przy uruchamianiu sterowników będzie fakt, że systemach z procesorem ARM inaczej wykonuje się odwołania do rejestrów sterujących urządzeń I/O. Nie wymagają one, jak w IA-32, osobnego interfejsu, ale wykonywane są dokładnie tak samo jak inne

odwołania do pamięci. Wynika z tego, że interfejs warstwy `hal` nie musi, i nie zawiera funkcji umożliwiających zapis i odczyt z przestrzeni wejścia wyjścia. Wynika z tego fakt, że nie możliwe jest bezpośrednie zastosowania sterowników napisanych dla urządzeń na platformę IA-32, pomimo tego, że na przykład sterownik portu szeregowego niemal niczym nie różni się na tych dwóch platformach. Pomimo, że mamy do czynienia ze zgodnymi sprzętowo układami peryferyjnymi, konieczne jest napisanie zmodyfikowanego sterownika do obsługi portu szeregowego miktokontrolera serii LPC. Warto sie w tym miejscu zastanowić, czy nie przygotować trywialnych wersji funkcji `hal_ina()` i `ha_outa()` służących do zapisu i odczytu danych z przestrzeni I/O. Jednakże według mnie, byłaby to próba nie tyle stworzenia abstrakcyjnej warstwy niezależnej od platformy, ale wymuszone emulowanie mechanizmów znanych z architektury IA-32.

c) Dalszy rozwój systemu.

Sam fakt uruchomienia jądra na platformie ARM jest tylko pierwszych krokiem większego procesu. Autor pracy zamierza nadal rozwijać wersję Phoenix-RTOS na platformie ARM. Kolejnym krokiem powinno być uruchomienie pozostałych modułów, oraz przygotowanie sterowników urządzeń. W pierwszej kolejności należałoby przygotować sterownik portu szeregowego, a w zasadzie zaadaptować istniejący bazując na adaptacji wykonanej w loaderze. Kolejnym istotnym krokiem powinno być uruchomienie kontrolera USB oraz stosu sieciowego wraz z interfejsem.

Obok sterowników konieczne jest przygotowanie jądra do romowania w pamięci Flash, gdyż jest to jedyna sensowna metoda pracy w docelowym systemie. Przechowywanie jądra w pamięci SRAM w docelowym urządzeniu, jest ewidentnym marnotrawstwem pamięci operacyjnej. Obecna wersja jądra zajmuje w pamięci 20KB, a każdy dołączony moduł wydatnie zwiększy to zapotrzebowanie.

W dalszej perspektywie jest przygotowanie wersji na procesory z rdzeniem ARM9 i jednostką zarządzania pamięcią, a następnie na rdzenie XScale.

8. Testowanie

Pierwsze testy dotyczyły konfiguracji wstępnej. Badanie polegało na sprawdzaniu, czy po przejściu procedury inicjalizacyjnej system jest skonfigurowany poprawnie.

Istotnym elementem było przetestowanie mapowania obszarów pamięci. W tym celu wywoływana była sytuacja wyjątkowa i obserwowana była reakcja systemu. Istotne było czy uruchomi się funkcja obsługi wyjątku jądra, czy loadera. Pop poprawnym skonfigurowaniu powinna uruchamiać się funkcja jądra. Na tym etapie pojawiły się problemu z dokładnym przekopiowaniem wektorów. W pierwszej wersji wykryto błąd przy indeksowaniu, który skutkował przesunięciem wektora o 4 bajty. Kolejna wersja nie kopiowała pierwszych 4 bajtów. Dopiero po kilku testach i gruntownym przeanalizowaniu procedury kopiującej udało się ten problem rozwiązać.

Przy testowaniu systemu przerwań istotna była sprawa zgłaszania przerwania. Działanie kontrolera przerwań nie było do końca jasno wyjaśnione, co wymagało w fazie testów sprawdzić jak się on zachowuje. Istotną sprawą był sposób zgłaszania przerwań. Zostało zasygnalizowano niebezpieczeństwo tego, że nieobsłużone przerwanie nie zostanie na powrót zgłoszone. Po przeprowadzeniu specjalnych testów z pustą funkcją obsługi, nie zerującą źródła przerwania a jedynie uaktualniającą układ

ustalania priorytetów, udało się dowieść, że nieobsłużone przerwanie zostanie zgłoszone ponownie.

Testowanie konsoli wykonywane było z poziomu loadera i opisane jest w rozdziale V.2.c. Testowanie linii statusu konsoli zostało wykonane poprzez cykliczne zapisywanie nowych wartości do jednego z pól na przemian z wypisywaniem danych na konsoli i obserwowanie wyniku. Początkowo, zanim skutecznie rozwiązano problem z synchronizacją zapisu, pojawiały się błędy polegające na wstrzykiwaniu znaków z linii statusu na konsoli. Występowało także niekontrolowane propagowanie kolorów między linią statusu a konsolą. Wszystkie problemy miały związek z synchronizacją dostępu do konsoli.

Testowanie mechanizmów jądra systemu Phoenix-RTOS umożliwia specjalny podsystem `test`. Udostępnia on przygotowany przez autora systemu interfejs wraz z procedurami testującymi poszczególne podsystemy jądra.

Wykonane zostały testy alokacji stron, alokacji obszarów drobnych oraz przełączania wątków.

Podczas testowania uruchamiane były trzy współbieżne wątki. Raportowane obciążenie systemu wynosiło 75% natomiast zajętość pamięci 22KB.

Testy synchronizacji za pomocą spinlocków przeprowadzane były w ramach tych współbieżnych wątków. Testowano synchronizacje dostępu do konsoli każdego z tych wątków. Pierwsze testy wykazały, że funkcja `hal_spinlockClear()` nie działa poprawnie. Błędnie realizowany był powrót do trybu Thumb. W wyniku tego błędu program przeskakiwał jedną instrukcję i dalsze wykonanie nie było przez to poprawne. Nie zwrócono uwagi na fakt, że kiedy używamy rejestru PC jako argumentu instrukcji, to wskazuje on dwie instrukcje naprzód, a nie jak mogłoby się wydawać na kolejną.

Wykrycie tego błędu w interpretacji działania instrukcji skłoniło do uważniejszego przyjrzenia się wszystkim fragmentom kodu pisanym w assemblerze. Znaleziono jeszcze kilka analogicznych błędów, które pozostawały okryte, gdyż szczęśliwym trafem nie zmieniały ścieżki wykonania programu.

VI. Podsumowanie

1. Wnioski.

Udało się osiągnąć wszystkie zakładane cele pracy. Okazało się, że system Phoenix-RTOS został dobrze przygotowany, na etapie konceptualnym, do przenoszenia na nowe platformy sprzętowe. Modyfikacje ograniczały się niemal wyłącznie do warstwy `hal`. Jedyna zmiana dokonana w części niezależnej od platformy dokonana została w podsystemie zarządzania pamięcią (`vm`). Projektant systemu nie przewidział wykorzystania systemu na platformach bez sprzętowej jednostki zarządzania pamięcią. Interfejs okazał się jednak na tyle elastyczny, że zmiany były minimalne. Wymagane było jedynie strywializowanie funkcji mapujących adresy wirtualne na fizyczne.

Mikrokontrolery z rdzeniem ARM okazały się bardzo wdzięcznym narzędziem pracy. Są bardzo wygodne w konfiguracji i nie stwarzają poważniejszych problemów. Niski pobór mocy w połączeniu z bogatym wyposażeniem i niewielkimi rozmiarami sprawiają, że układy te mogą być wykorzystywane w wielu aplikacjach. Niska cena sprawia, że z pewnością mikrokontrolery te mogą z powodzeniem zastąpić wysłużone już układy rodziny '51 i inne 8-bitowe mikrokontrolery. Pozwalają one korzystać z zalet 32-bitowej architektury za rozsądną cenę.

Architektura ARM jest dobrze zaprojektowana. Jest to klasyczny, 32-bitowy RISC. Język maszynowy jest specyficzny, ale przejrzysty i pozwalający na wydajne programowanie. Duża ilość rejestrów ogólnego przeznaczenia sprawia, że koszt wywołania funkcji często niczym nie różni się od innych skoków. Ciekawym rozwiązaniem jest także tryb Thumb, w którym mamy 16-bitowe słowo rozkazowe co pozwala znakomicie zwiększyć gęstość kodu.

Kolejnym etapem projektu powinno być uruchomienie dodatkowych podsystemów. Krytyczne jest uruchomienie podsystemu `dev`, odpowiadającego za obsługę urządzeń, oraz podsystemu `fs` – czyli obsługi systemu plików.

Następnie konieczne jest przygotowanie sterowników urządzeń. W pierwszej kolejności należałoby przygotować sterowniki dla urządzeń peryferyjnych mikrokontrolera LPC2148, a w szczególności dla kontrolera asynchronicznych portów szeregowych oraz kontrolera USB. Niezwykle przydatne byłyby także sterowniki dla przetworników: cyforowo-analogowego i analogowo-cyfrowego, oraz prosty sterownik dla wyświetlacza ciekłokrystalicznego. Użyteczny byłby sterownik synchronicznego portu szeregowe, przydatny chociażby przy obsłudze kart MMC (ang. MultiMedia Card). Kary pamięci mogłyby być wykorzystywane jako zewnętrzne nośniki pamięci masowej.

Istotne jest także uruchomienie podsystemu `net`, implementującego stos UDP/IP, oraz przygotowanie obsługi interfejsów sieciowych.

W dalszej perspektywie konieczne jest przygotowanie obsługi jednostki zarządzania pamięcią, obecnej w bardziej zaawansowanych mikrokontrolerach ARM. Mowa tutaj głównie o mikrokontrolerach z rdzeniem ARM9.[50] Autor ma nadzieję rozwinąć ten wątek projektu w ramach pracy magisterskiej.

50 Patrz [A9]

2. Kod źródłowy i instalacja.

Aktualną wersje kodu źródłowego systemu Phoenix-RTOS znaleźć można na na dołączonej płycie CD oraz na stronach projektu:

https://devel.phoenix-rtos.org/phoenix-rtos/

Do kompilacji zaleca się użycie kompilatora GNU/GCC. Binarne wersje odpowiednich pakietów znaleźć można na stronach http://www.gnuarm.org/. Zalecana wersja kompilatora to 3.4.3.

Aby załadować obraz programu ładującego (`arm-plo`) do pamięci Flash mikrokontrolera niezbędny jest specjalizowany program, implementujący procedury protokołu ISP (ang. In-System-Programming).. Najlepiej użyć do tego celu programu dostarczonego przez producenta mikrokontrolera. W przypadku LPC2000 jest to, dostarczany przez firmę Philips, LPC2000 Flash Utility[51]. Program ten jest udostępniony do ściągnięcia na stronach NXP Semiconductors (dawne Philips Semiconductors). Dostępne są także inne programy ładujące przygotowane pod różne platformy, zarówno Windows jak i GNU/Linux.

Jądro systemu w obecnej wersji ładowane jest przez program ładujący, do pamięci SRAM, przez drugi port szeregowy mikrokontrolera. Po stronie serwera używany jest program Phoenix Server będący częścią projektu.

51 Program do ściągnięcia ze strony: [ISP]

Bibliografia

[PM] Paweł Pisarczyk - *Phoenix - Mikrojądro Rozproszonego Systemu o Podwyższonej Niezawodności* - Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, 2001, 120str.

[PH] Phoenix-RTOS - free realtime operating system - *http://www.phoenix-rtos.org*

[WIKI] Architektura ARM - *http://pl.wikipedia.org/wiki/Architektura_ARM*

[ARL] ARM Ltd. - *http://www.arm.com/*

[ARM] *ARM Architecture Reference Manual* - ARM Ltd.,1996-2000, wyd. E, 811str.

[LPC] Philips Semiconductors (obecnie NXP Semiconductors) - *Microcontrollers Selection Guide* (2006)

[AT] Atmel - *Atmel Smart ARM Microcontrollers Product Selection Guide* (2006)

[UM] *UM10139 Volume 1: LPC214x User Manual* - Philips Semiconductors (obecnie NXP Semiconductors),2005, wyd. 01, 348str.

[UM] *UM10139 Volume 1: LPC214x User Manual* - Philips Semiconductors,2005, wyd. 01, 348str.

[A7] ARM7TDMI-S - *http://www.arm.com/products/CPUs/ARM7TDMIS.html*

[NXP] NXP Semiconductors (dawny Philips Semiconductors) - Microcontrollers [Products - LPC2000] - *http://www.standardics.nxp.com/products/lpc2000/*

[ISP] NXP Semiconductors - Software Tool ISP Software LPC2100 (ARM7) - *http://www.nxp.com/products/microcontrollers/support/development_tools/general/includes/arm7tdmi/isp_sw.html*

[AR] D.W. Hawkins - *Real-time processing with the Philips LPC ARM microcontroller; using GCC and the MicroC/OS-II RTOS* - Philips 05: Project Number AR1803 - 2006

[PSO] A. Silberschatz, P.B. Gavin - *Podstawy Systemów Operacyjnych* - WNT, 2002, wyd. 5, 995str., ISBN 83-204-2768-1

[PA] Palm Software and Palm OS - *http://www.palmsource.com/*

[CE] Windows CE Home Page - *http://msdn.microsoft.com/embedded/windowsce/*

[SYM] Symbian OS - the mobile operating system - *http://www.symbian.com/*

[GHS] Green Hills Software Inc. - *http://www.ghs.com/*

[ECO] eCos - *http://ecos.sourceware.org/*

[QNX] QNX Realtime operating system (RTOS) - *http://www.qnx.com/*

[BSD] NetBSD/arm - *http://www.netbsd.org/Ports/arm/*

[DEB] Debian -- ARM Port - *http://www.debian.org/ports/arm/*

[PCS] ARM Ltd. - *Procedure Call Standard for the ARM Architecture* (2006)

[GCC] GNU Press - *Using the GNU Compiler Collection* (2004)

[ECM] *Standard ECMA-48: Control Functions for Coded Character Sets* - ECMA,1991, wyd. 1, 108str.

[A9] ARM9 - *http://www.arm.com/products/CPUs/families/ARM9Family.html*