

**FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO**

**MICROKERNEL PARA A PLACA ARM
EVALUATOR-7T**

São Paulo
2009

**FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO**

**MICROKERNEL PARA A PLACA ARM
EVALUATOR-7T**

Monografia apresentada à Escola
Politécnica da Universidade de São
Paulo para a Conclusão do Curso de
Engenharia da Computação.

São Paulo
2009

**FELIPE GIUNTE YOSHIDA
MARIANA RAMOS FRANCO
VINICIUS TOSTA RIBEIRO**

**MICROKERNEL PARA A PLACA ARM
EVALUATOR-7T**

Monografia apresentada à Escola
Politécnica da Universidade de São
Paulo para a Conclusão do Curso de
Engenharia da Computação.

Orientador:
Prof. Dr. Jorge Kinoshita

São Paulo
2009

FICHA CATALOGRÁFICA

Yoshida, Felipe Giunte

Microkernel para a placa ARM Evaluator-7T / F.G. Yoshida, M.R. Franco, V.T. Ribeiro. – São Paulo, 2009. 110 p.

Trabalho de Formatura — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Sistemas operacionais (Desenvolvimento). 2. Microprocessadores. I. Franco, Mariana Ramos II. Ribeiro, Vinicius Tosta III. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais. IV. t.

DEDICATÓRIA

Aos meus pais, Noboru e Sonia, que me apoiaram e possibilitaram que chegasse até aqui.

Aos meus avós, por todo o suporte e carinho, essenciais para percorrer esta longa jornada.

E aos meus mestres, que me guiaram ao longo do caminho.

-Felipe Giunte Yoshida

Aos meus pais, Edson e Waldinéia, por sempre me apoiarem e me guiarem nas decisões importantes que me levaram até aqui.

E aos meus irmãos, Vinicius e Fernando, pelo carinho e amizade que nos une.

-Mariana Ramos Franco

-Vinicius Tosta Ribeiro

AGRADECIMENTOS

Ao Professor Jorge Kinoshita, pelo incentivo, orientação e disposição em todos os momentos durante o projeto.

A Escola Politécnica da Universidade de São Paulo, que nos deu a oportunidade de aprendizagem e crescimento.

Ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS), pelo Curso Cooperativo de Engenharia da Computação.

RESUMO

O uso de dispositivos móveis como celulares, *smartphones*, tocadores de MP3 e *video-games* portáteis é cada vez mais comum. A atual líder no segmento de processadores de baixa potência, essencial nestes tipos de aparelhos, é a empresa inglesa ARM. À fim de se modernizar o equipamento usado em aulas, ela disponibilizou à Escola Politécnica algumas placas ARM Evaluator 7-T, cujo processador, o ARM7TDMI, é usado em eletrônicos muito populares atualmente, como o Apple iPod e o Nintendo DS.

Assim, utilizando-se desse hardware mais moderno e pensando em aproximar o ensino das disciplinas de Sistemas Operacionais e do Laboratório de Microprocessadores, este projeto visa o desenvolvimento de um *microkernel* para a placa ARM Evaluator-7T, tema que engloba conhecimento de ambas as disciplinas.

O *microkernel* desenvolvido, chamado de KinOS, é provido de algumas funções básicas, e é apenas o primeiro passo para um projeto muito maior, de desenvolvimento de um sistema operacional totalmente feito por alunos da Escola Politécnica.

Dentre as funções básicas deste *microkernel*, podemos citar o chaveamento de *threads*, algumas chamadas de sistema (*fork*, *exec* e *exit*), funções para manipulação de periféricos e comunicação através de um terminal.

ABSTRACT

Mobile devices such as smartphones, MP3 players and portable video-games are becoming ubiquitous. Low power processors are essential in this market, where the English company ARM is the leader. In order to upgrade the equipment being used in the classes, ARM provided a set of ARM Evaluator 7-T boards to the Escola Politécnica. It has the ARM7TDMI processor, which is used in popular devices such as the Apple iPod and the Nintendo DS.

Using this updated hardware and willing to unite the Operating Systems and Microprocessors Laboratory courses, this project aims the development of a microkernel for the ARM Evaluator 7-T board, which would encompass both courses.

The microkernel, named KinOS, has some basic functions. It is the first step towards a bigger project, the development of a operating system totally created by the Escola Politécnica students.

The functions encompassed by this microkernel include the thread switching, some system calls (fork, exec and exit), functions for peripheral manipulation, and shell communication.

LISTA DE FIGURAS

2.1	<i>Pipeline</i> de 3 estágios (ARM LIMITED, 2001b)	22
2.2	<i>Pipeline</i> do ARM7TDMI (RYZHYK, 2006)	23
2.3	Organização dos registradores no modo ARM (ARM LIMITED, 2001b)	26
2.4	Formato dos registradores de estado CPSR e SPSR (ARM LIMITED, 2001b)	28
2.5	Esquema de uma interrupção no ARM7TDMI (ZAITSEFF, 2003)	32
2.6	Passagem de argumentos (SLOSS; SYMES; WRIGHT, 2004)	36
2.7	Arquitetura da placa Evaluator-7T. (ARM LIMITED, 2000)	37
2.8	Editor de linha de comando do BSL via HyperTerminal	39
3.1	Estrutura de arquivos.	45
3.2	Estrutura de dados do PCB. Fonte: (SLOSS, 2001)	47
3.3	Vetor de <i>threads</i>	48
3.4	Estrutura da memória. Fonte: (SLOSS, 2001)	49
3.5	Fluxograma de inicialização.	51
3.6	Encadeamento de interrupções. Fonte: (SLOSS, 2001)	55
3.7	Chaveamento de processos.	56
3.8	Comunicação da Evaluator-7T em cada porta serial.	66

LISTA DE TABELAS

2.1	Modos de operação (ARM LIMITED, 2005)	25
2.2	Valores para o bit de modo (ARM LIMITED, 2005)	30
2.3	Vetor de interrupção (ARM LIMITED, 2005)	31
2.4	Ordem de prioridade das interrupções (ARM LIMITED, 2001b)	31
2.5	Mapa da memória flash	38

LISTA DE ABREVIATURAS

ADS ARM Developer Suite

ALU Arithmetic Logic Unit

ARM Advanced RISC Machine

CISC Complex Instruction Set Computer

CPSR Current Program Status Register

FIQ Fast Interrupt

IDE Integrated Development Environment

IRQ Interrupt Request

LR Link Register

PC Program Counter

PSR Program Status Register

RISC Reduced Instruction Set Computer

SP Stack Pointer

SPRS Saved Program Register

SWI Software Interruption

USP Universidade de São Paulo

LISTA DE SÍMBOLOS

RX - registrador número X

RX_Y - registrador número X do modo de operação Y

SUMÁRIO

1	Introdução	17
1.1	Objetivo	17
1.2	Motivação	17
1.3	Justificativa	18
1.4	Metodologia de Trabalho	18
1.5	Organização do Documento	19
2	Conceitos e Tecnologias Envolvidas	21
2.1	O Processador ARM7TDMI	21
2.1.1	Arquitetura RISC	21
2.1.2	Pipeline	22
2.1.3	Estados de Operação	24
2.1.4	Modos de Operação	24
2.1.5	Registradores	25
2.1.6	Registradores de Estado	27
2.1.7	Interrupções	30
2.1.8	Programando em C pra o ARM7TDMI	35
2.2	A Placa Experimental Evaluator-7T	36
2.2.1	Bootstrap Loader	38
2.2.2	Angel Debug Monitor	41
2.3	O ambiente de desenvolvimento	42
2.3.1	CodeWarrior	42

2.3.2	AXD Debugger	43
3	O Sistema Operacional KinOS	44
3.1	Organização do código	44
3.1.1	Raiz	44
3.1.2	Pasta “apps”	45
3.1.3	Pasta “interrupt”	46
3.1.4	Pasta “peripherals”	46
3.1.5	Pasta “syscalls”	46
3.1.6	Pasta “mutex”	46
3.2	Estruturas de dados	46
3.2.1	Process Control Block	47
3.2.2	Vetor de <i>threads</i>	48
3.3	Configuração de <i>hardware</i> e <i>software</i>	48
3.3.1	Memória	48
3.3.2	Modos do processador	49
3.3.3	Modos de teste	50
3.3.4	Angel	50
3.4	Inicialização	50
3.4.1	Ponto de entrada e tipo de código	51
3.4.2	Pilhas	51
3.4.3	Vetor de threads e número da thread	52
3.4.4	Periféricos	53
3.4.5	Instalação do tratamento de interrupção	53
3.4.6	Interrupção de timer	55
3.4.7	Habilitando interrupções	55
3.5	Chaveamento de processos	55

3.5.1	Identificação da interrupção	56
3.5.2	Limpeza da interrupção de timer	57
3.5.3	Identificação da próxima thread	57
3.5.4	Localização dos PCBs	58
3.5.5	A troca de processos	59
3.5.6	Retorno à execução da nova rotina	60
3.6	Chamadas de sistema	61
3.6.1	Propriedades gerais	61
3.6.2	fork	62
3.6.3	exec	63
3.6.4	exit	65
3.7	Shell	65
3.7.1	Comunicação via terminal	65
3.8	Mutex	65
3.9	Processos	66
3.10	Inspiração	66
4	Considerações Finais	67
4.1	Conclusão	67
4.2	Contribuições	67
4.3	Trabalhos Futuros	67
	Referências Bibliográficas	68
A	Arquivos Fonte	70
A.1	cinit.h	70
A.2	cinit.c	70
A.3	constants.h	72

A.4	startup.s	73
A.5	apps/tasks.h	75
A.6	apps/tasks.c	75
A.7	apps/terminal.h	77
A.8	apps/terminal.c	77
A.9	interrupt/handler_irq.s	82
A.10	interrupt/handler_swi.s	87
A.11	interrupt/irq.h	89
A.12	interrupt/irq.c	89
A.13	interrupt/swi.h	90
A.14	interrupt/swi.c	91
A.15	mutex/mutex.h	91
A.16	mutex/mutex.c	92
A.17	peripherals/button.h	92
A.18	peripherals/button.c	92
A.19	peripherals/dips.h	93
A.20	peripherals/dips.c	93
A.21	peripherals/led.h	94
A.22	peripherals/segment.h	94
A.23	peripherals/segment.c	94
A.24	peripherals/serial.h	95
A.25	peripherals/serial.c	98
A.26	peripherals/timer.h	103
A.27	peripherals/timer.c	104
A.28	syscalls/exec.s	105
A.29	syscalls/exit.s	107

A.30 syscalls/fork.s	107
--------------------------------	-----

1 INTRODUÇÃO

1.1 Objetivo

O objetivo deste projeto de formatura é desenvolver um *microkernel* para a placa experimental ARM Evalutator-7T, constituída de um processador ARM7TDMI e de alguns periféricos simples.

O *microkernel* implementa os mecanismos básicos de um sistema operacional, como o chaveamento de *threads*, as chamadas de sistema e utiliza algumas rotinas para a comunicação com os periféricos da placa.

Além disso, foram criados alguns programas para testar e exemplificar o funcionamento do *microkernel*. Entre esses programas, um simples terminal foi desenvolvido para a interação dos usuários com o sistema.

1.2 Motivação

As disciplinas de Laboratório de Microprocessadores e de Sistemas Operacionais do curso de Engenharia da Computação na Escola Politécnica da USP, atualmente, estão muito distantes entre si, no entanto o conteúdo das mesmas é muito próximo.

Pensando em como aproximar essas duas disciplinas, surgiu a idéia de desenvolver uma ferramenta didática que unisse um *hardware* e sistema operacional de estudo simples, e que pudesse ser utilizada nas experiências do Laboratório de Microprocessadores.

Para criação desta ferramenta, foi escolhida a placa experimental ARM Evaluator-7T, que possui uma arquitetura ARM e um poder de processamento bastante superior aos sistemas didáticos utilizados atualmente (baseados nos processadores Intel 8051 e no Motorola 68000). Assim sendo, pretende-se atualizar o material didático da disciplina de microprocessadores, trazendo um sistema mais moderno e mais próximo da realidade atual, além de poder se relacionar com o conteúdo da disciplina de Sistemas Operacionais.

Outra motivação do projeto foi aprofundar nossos conhecimentos sobre sistemas operacionais e sobre a arquitetura dos processadores ARM, visto que este processador é, hoje em dia, largamente utilizado em sistemas embarcados e aparelhos celulares.

1.3 Justificativa

O objetivo inicial do projeto era portar um sistema operacional Unix já existente para a placa didática Evaluator-7T.

Inicialmente pensamos em utilizar os sistemas Android e Minix 3, mas ao estudar o *kernel* dos dois sistemas, vimos que os recursos de memória necessários para executá-los era muito maior que os 512kB disponíveis na placa. Além disso, no caso do Minix 3, teríamos que reescrever o *assembly* do *kernel* que atualmente só tem versão para i386, para *assembly* ARM, o que seria impossível com o tempo disponível para o projeto.

Assim surgiu a idéia de desenvolver um *microkernel* próprio, com as funcionalidades básicas de um sistema operacional, e que fosse de fácil entendimento; pois como mencionado anteriormente, espera-se que o material desenvolvido seja destinado a melhorar e aproximar o ensino de Sistemas Operacionais com as experiências do Laboratório de Microprocessadores.

1.4 Metodologia de Trabalho

Para a realização desse projeto de formatura procurou-se seguir uma metodologia de trabalho cujas etapas são descritas a seguir:

- Estudo da Arquitetura ARM e da Placa Didática Evaluator-7T:

Antes de especificar as funcionalidades que seriam desenvolvidas, um estudo aprofundado da arquitetura ARM foi realizado para compreender o funcionamento do processador para o qual o *microkernel* foi desenvolvido, o ARM7TDMI.

Além disso, foram executados alguns programas exemplo na placa didática Evaluator-7T para adquirir conhecimentos sobre o seu funcionamento e limitações.

- Montagem do Ambiente de Trabalho:

Paralelamente ao estudo descrito no item anterior, foi montado um ambiente de trabalho utilizando a IDE CodeWarrior para o desenvolvimento do código-fonte e o AXD Debugger para depurar o funcionamento do *microkernel* com ou sem a utilização da placa didática.

Um repositório de controle de versão também foi montado para estocar o material produzido durante do projeto (documentação e código-fonte) e para sincronizar o trabalho dos integrantes do grupo. Seu endereço é <http://code.google.com/p/arm7linux/>

- Especificação Funcional do Microkernel:

O *microkernel* desenvolvido foi especificado nessa etapa, onde foram levantadas as funcionalidades básicas de um sistema operacional que deveriam ser implementadas, como o chaveamento de *threads* e as chamadas de sistema.

- Desenvolvimento do Microkernel:

Nessa fase, foi desenvolvido o *microkernel* utilizando como base a especificação definida no item anterior.

- Análise do Microkernel e Conclusões:

Ao final do desenvolvimento, com base nas dificuldades e soluções encontradas, foi feita uma análise e conclusão sobre o *microkernel* desenvolvido e sua possível utilização no Laboratório de Microprocessadores para exemplificar os conceitos vistos na disciplina de Sistemas Operacionais.

1.5 Organização do Documento

Este documento foi estruturado da seguinte maneira:

- Capítulo 1 (Introdução):

Apresenta objetivo, motivações, justificativas e a metodologia do trabalho.

- Capítulo 2 (Conceitos e Tecnologias Envolvidas):

Contextualiza o leitor em aspectos técnicos específicos utilizados no desenvolvimento do trabalho.

- Capítulo 3 (O Sistema Operacional KinOS):

Descreve como o *microkernel* foi desenvolvido, quais as suas funcionalidades e como funciona a sua integração com os periféricos da placa didática, com o terminal e com os outros programas implementados.

- Capítulo 4 (Considerações Finais):

Analisa os resultados obtidos em relação ao objetivo do projeto, as conclusões, as contribuições deste trabalho e indica possíveis trabalhos futuros com base neste.

2 CONCEITOS E TECNOLOGIAS ENVOLVIDAS

2.1 O Processador ARM7TDMI

O ARM7TDMI faz parte da família de processadores ARM7 32 bits conhecida por oferecer bom desempenho aliado a um baixo consumo de energia. Essas características fazem com que o ARM7TDMI seja bastante utilizado em media players, videogames e, principalmente, em sistemas embarcados e num grande número de aparelhos celulares (SLOSS; SYMES; WRIGHT, 2004).

2.1.1 Arquitetura RISC

Os processadores ARM, incluindo o ARM7TDMI, foram projetados com a arquitetura RISC.

RISC (*Reduced Instruction Set Computer*) é uma arquitetura de computadores baseada em um conjunto simples e pequeno de instruções capazes de serem executadas em um único ou poucos ciclos de relógio.

A idéia por trás da arquitetura RISC é de reduzir a complexidade das instruções executadas pelo *hardware* e deixar as tarefas mais complexas para o *software*. Como resultado, o RISC demanda mais do compilador do que os tradicionais computadores CISC (*Complex Instruction Set Computer*) que, por sua vez, dependem mais do processador já que suas instruções são mais complicadas (SLOSS; SYMES; WRIGHT, 2004).

As principais características da arquitetura RISC são:

1. Conjunto reduzido e simples de instruções capazes de serem executadas em único ciclo de máquina.
2. Uso de *pipeline*, ou seja, o processamento das instruções é quebrado em pequenas unidades que podem ser executadas em paralelo.

3. Presença de um conjunto de registradores.
4. Arquitetura *Load-Store*: o processador opera somente sobre os dados contidos nos registradores e instruções de *load/store* transferem dados entre a memória e os registradores.
5. Modos simples de endereçamento de memória.

2.1.2 Pipeline

A arquitetura de *pipeline* aumenta a velocidade do fluxo de instruções para o processador, pois permite que várias operações ocorram simultaneamente, fazendo o processador e a memória operarem continuamente (ARM LIMITED, 2001b).

O ARM7 possui uma arquitetura de *pipeline* de três estágios. Durante operação normal, o processador estará sempre ocupado em executar três instruções em diferentes estágios. Enquanto executa a primeira, decodifica a segunda e busca a terceira.

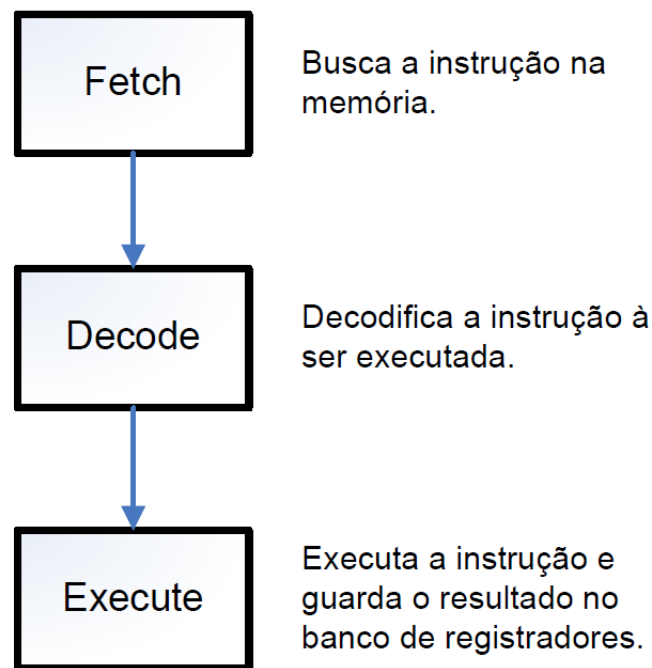


Figura 2.1: Pipeline de 3 estágios (ARM LIMITED, 2001b)

O primeiro estágio de *pipeline* lê a instrução da memória e incrementa o valor do registrador de endereços, que guarda o valor da próxima instrução a ser buscada. O próximo estágio decodifica a instrução e prepara os sinais de controle necessários para executá-la. O terceiro lê os operandos do banco de registradores, executa as operações através da ALU (*Arithmetic*

Logic Unit), lê ou escreve na memória, se necessário, e guarda o resultado das instruções no banco de registradores.

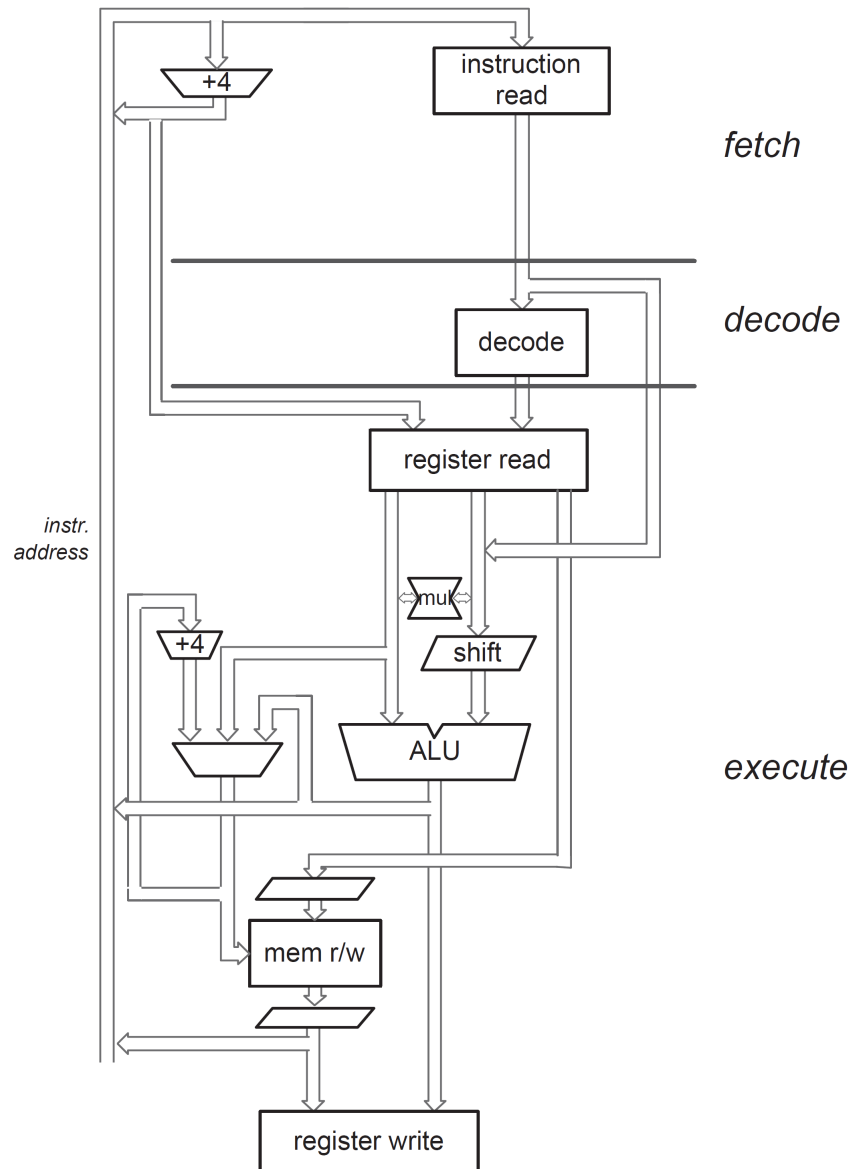


Figura 2.2: Pipeline do ARM7TDMI (RYZHYK, 2006)

Algumas características importantes do *pipeline* do ARM7TDMI:

- O *Program Counter* (PC) ao invés de apontar para a instrução que está sendo executada, aponta para a instrução que está sendo buscada na memória.
- O processador só processa a instrução quando essa passa completamente pelo estágio de execução (*execute*). Ou seja, somente quando a quarta instrução é buscada (*fetched*).

- A execução de uma instrução de *branch* através da modificação do PC provoca a descarga, eliminação, de todas as outras instruções do *pipeline*.
- Uma instrução no estágio *execute* será completada mesmo se acontecer uma interrupção. As outras instruções no *pipeline* serão abandonadas e o processador começará a preencher o *pipeline* a partir da entrada apropriada no vetor de interrupção.

2.1.3 Estados de Operação

O processador ARM7TDMI possui dois estados de operação (ARM LIMITED, 2001b):

- ARM: modo normal, onde o processador executa instruções de 32 bits (cada instrução corresponde a uma palavra);
- Thumb: modo especial, onde o processador executa instruções de 16 bits que correspondem à meia palavra.

Instruções Thumbs são um conjunto de instruções de 16 bits equivalentes as instruções 32 bits ARM. A vantagem em tal esquema, é que a densidade de código aumenta, já que o espaço necessário para um mesmo número de instruções é menor. Em compensação, nem todas as instruções ARM tem um equivalente Thumb.

Neste projeto, o processador é usado no modo ARM que facilita o desenvolvimento por possuir um número maior de instruções.

2.1.4 Modos de Operação

Os processadores ARM possuem 7 modos de operação, como apresentado na tabela 2.1.

Mudanças no modo de operação podem ser realizadas através de programas, ou podem ser causadas por interrupções externas ou exceções (interrupções de software).

A maioria dos programas roda no modo Usuário. Quando o processador esta no modo Usuário, o programa que esta sendo executado não pode acessar alguns recursos protegidos do sistema ou mudar de modo sem ser através de uma interrupção (ARM LIMITED, 2005).

Os outros modos são conhecidos como modos privilegiados. Eles têm total acesso aos recursos do sistema e podem mudar livremente de modo de operação. Cinco desses modos são conhecidos como modos de interrupção: FIQ, IRQ, Supervisor, *Abort* e Indefinido.

Modo	Identificador	Descrição
Usuário	usr	Execução normal de programas.
FIQ (<i>Fast Interrupt</i>)	fiq	Tratamento de interrupções rápidas.
IRQ (<i>Interrupt</i>)	irq	Tratamento de interrupções comuns.
Supervisor	svc	Modo protegido para o sistema operacional.
<i>Abort</i>	abt	Usado para implementar memória virtual ou manipular violações na memória.
Sistema	sys	Executa rotinas privilegiadas do sistema operacional.
Indefinido	und	Modo usado quando uma instrução desconhecida é executada.

Tabela 2.1: Modos de operação (ARM LIMITED, 2005)

Entra-se nesses modos quando uma interrupção ocorre. Cada um deles possui registradores adicionais que permitem salvar o modo Usuário quando uma interrupção ocorre.

O modo remanescente é o modo Sistema, que não é acessível por interrupção e usa os mesmos registradores disponíveis para o modo Usuário. No entanto, este é um modo privilegiado e, assim, não possui as restrições do modo Usuário. Este modo destina-se as operações que necessitam de acesso aos recursos do sistema, mas querem evitar o uso adicional dos registradores associados aos modos de interrupção.

2.1.5 Registradores

O processador ARM7TDMI tem um total de 37 registradores:

- 31 registradores de 32 bits de uso geral
- 6 registradores de estado

Esses registradores não são todos acessíveis ao mesmo tempo. O modo de operação do processador determina quais registradores são disponíveis ao programador (ARM LIMITED, 2001b).

2.1.5.1 Modo Usuário e Sistema

O conjunto de registradores para o modo Usuário (o mesmo usado no modo Sistema) contém 16 registradores diretamente acessíveis, R0 à R15. Um registrador adicional, o CPSR (*Current Program Status Register*), contém os bits de *flag* e de modo.

Os registradores R13 à R15 possuem as seguintes funções especiais (SLOSS; SYMES; WRIGHT, 2004):

- R13: usado como ponteiro de pilha, *Stack Pointer* (SP)
- R14: é chamado de *Link Register* (LR) e é onde se coloca o endereço de retorno sempre que uma sub-rotina é chamada.
- R15: corresponde ao *Program Counter* (PC) e contém o endereço da próxima instrução à ser executada pelo processador.

2.1.5.2 Modos privilegiados

Além dos registradores acessíveis ao programador, o ARM coloca à disposição mais alguns registradores nos modos privilegiados. Esses registradores são mapeados aos registradores acessíveis ao programador no modo Usuário e permitem que estes sejam salvos a cada interrupção.

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM-state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und


 = banked register

Figura 2.3: Organização dos registradores no modo ARM (ARM LIMITED, 2001b)

Como se pode verificar na figura 2.3, cada modo tem o seu próprio R13 e R14. Isso permite que cada modo mantenha seu próprio ponteiro de pilha (SP) e endereço de retorno (LR) (ZAITSEFF, 2003).

Além desses dois registradores, o modo FIQ possui mais cinco registradores especiais: R8_fiq-R12_fiq. Isso significa que quando o processador muda para o modo FIQ, o programa não precisa salvar os registradores de R8 à R12.

Esses registradores especiais mapeiam de um pra um os registradores do modo Usuário. Se ocorrer uma mudança de modo do processador, um registrador particular do novo modo irá substituir o registrador existente.

Por exemplo, quando o processador está no modo IRQ, as instruções executadas continuarão a acessar os registradores R13 e R14. No entanto, esses serão os registradores especiais R13_irq e R14_irq. Os registradores do modo usuário (R13_usr e R14_usr) não serão afetados pelas instruções referenciando esses registradores. O programa continua tendo acesso normal aos outros registradores de R0 à R12 (SLOSS; SYMES; WRIGHT, 2004).

2.1.6 Registradores de Estado

O *Current Program Status Register* (CPSR) é acessível em todos os modos do processador. Ele contém as *flags* de condição, os bits para desabilitar as interrupções, o modo atual do processador, e outras informações de estado e controle. Cada modo de interrupção possui também um *Saved Program Register* (SPSR), que é usado para preservar o valor do CPSR quando a interrupção associada acontece (ARM LIMITED, 2005).

Assim, os registradores de estado (ARM LIMITED, 2001b):

- Guardam informação sobre a operação mais recente executada pela ALU.
- Controlam o ativar e desativar de interrupções.
- Determinam o modo de operação do processador.

Como mostrado na figura 2.4 o CPSR é dividido em 3 campos: *flag*, reservado (não utilizado) e controle.

O campo de controle guarda os bits de modo, estado e de interrupção, enquanto o campo *flag* armazena os bits de condição.

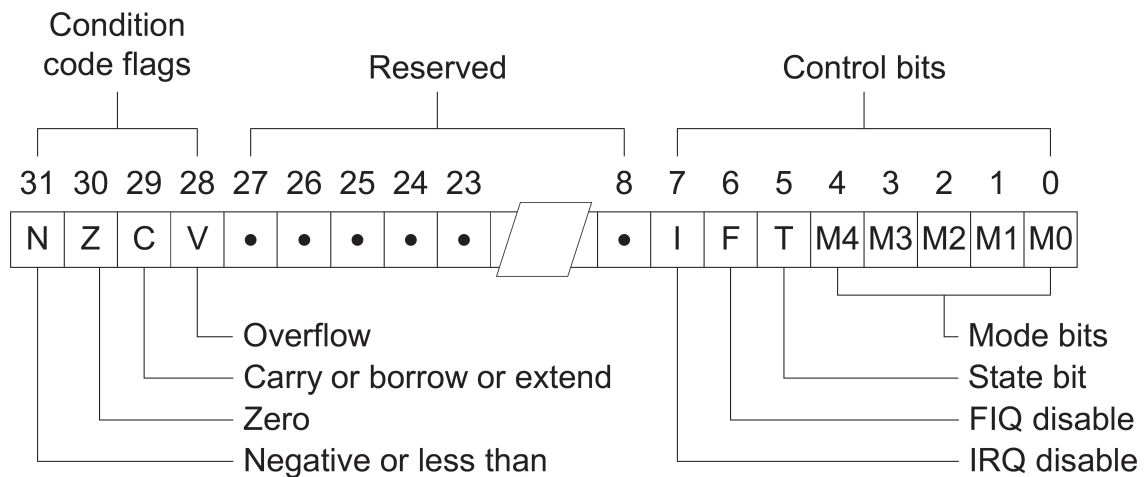


Figura 2.4: Formato dos registradores de estado CPSR e SPSR (ARM LIMITED, 2001b)

2.1.6.1 Flags de Condição

Os bits N, Z, C e V são *flags* de condição, e é possível alterá-los através do resultado de operações lógicas ou aritméticas (ARM LIMITED, 2005).

Os *flags* de condição são normalmente modificados por:

- Uma instrução de comparação (CMN, CMP, TEQ, TST).
- Alguma outra instrução aritmética, lógica ou *move*, onde o registrador de destino não é o R15 (PC).

Nesses dois casos, as novas *flags* de condição (depois de a instrução ter sido executada) normalmente significam:

- N: Indica se o resultado da instrução é um número positivo (N=0) ou negativo (N=1).
- Z: Contém 1 se o resultado da instrução é zero (isso normalmente indica um resultado de igualdade para uma comparação), e 0 se o contrário.
- C: Pode possuir significados diferentes:
 - Para uma adição, C contém 1 se a adição produz "vai-um" (*carry*), e 0 caso contrário.
 - Para uma subtração, C contém 0 se a subtração produz "vem-um" (*borrow*), e 1 caso contrário.

- Para as instruções que incorporam deslocamento, C contém o último bit deslocado para fora pelo deslocador.
- Para outras instruções, C normalmente não é usado.
- V: Possui dois significados:
 - Para adição ou subtração, V contém 1 caso tenha ocorrido um *overflow* considerando os operandos e o resultado em complemento de dois.
 - Para outras instruções, V normalmente não é usado.

2.1.6.2 Bits de Controle

Os oito primeiros bits de um PSR (*Program Status Register*) são conhecidos como bits de controle (ARM LIMITED, 2005). Eles são:

- Bits de desativação de interrupção
- Bit T
- Bits de modo

Os bits de controle mudam quando uma interrupção acontece. Quando o processador está operando em um modo privilegiado, programas podem manipular esses bits.

Bits de desativação de interrupção

Os bits I e F são bits de desativação de interrupção:

- Quando o bit I é ativado, as interrupções IRQ são desativadas.
- Quando o bit F é ativado, as interrupções FIQ são desativadas.

Bit T

O bit T reflete o modo de operação:

- Quando o bit T é ativado, o processador é executado em estado Thumb.
- Quando o bit T é desativado, o processador é executado em estado ARM.

Bits de modo

Os bits M[4:0] determinam o modo de operação. Nem todas as combinações dos bits de modo definem um modo válido, portando deve-se tomar cuidado para usar somente as combinações mostradas na tabela 2.2.

Bit de modo	Modo de operação	Registradores acessíveis
10000	Usuário(usr)	PC,R14-R0,CPSR
10001	FIQ(fiq)	PC,R14_fiq-R8_fiq,R7-R0,CPSR,SPSR_fiq
10010	IRQ(irq)	PC,R14_irq, R13_irq,R12-R0,CPSR,SPSR_irq
10011	Supervisor(svc)	PC,R14_svc, R13_irq,R12-R0,CPSR,SPSR_svc
10111	<i>Abort</i> (abt)	PC,R14_abt, R13_irq,R12-R0,CPSR,SPSR_abt
11011	Indefinido(und)	PC,R14_und, R13_irq,R12-R0,CPSR,SPSR_und
11111	Sistema(sys)	PC,R14-R0,CPRS

Tabela 2.2: Valores para o bit de modo (ARM LIMITED, 2005)

2.1.7 Interrupções

Interrupções surgem sempre que o fluxo normal de um programa deve ser interrompido temporariamente, por exemplo, para servir uma interrupção vinda de um periférico ou a tentativa de executar uma instrução desconhecida. Antes de tentar lidar com uma interrupção, o ARM7TDMI preserva o estado atual de forma que o programa original possa ser retomado quando a rotina de interrupção tiver acabado (ARM LIMITED, 2001b).

A arquitetura ARM suporta 7 tipos de interrupções. A tabela 2.3 lista os tipos de interrupção e o modo do processador usado para lidar com cada tipo. Quando uma interrupção acontece, a execução é forçada para um endereço fixo de memória correspondente ao tipo de interrupção. Esses endereços fixos são chamados de vetores de interrupção (ARM LIMITED, 2005).

Deve-se notar olhando para a tabela 2.3, que existe espaço suficiente para apenas uma instrução entre cada vetor de interrupção (4 bytes). Estes são inicializados com instruções de desvio (*branch*).

2.1.7.1 Prioridade das Interrupções

Quando várias interrupções acontecem ao mesmo tempo, uma prioridade fixa do sistema determina a ordem na qual elas serão manipuladas. Essa prioridade é listada na tabela 2.4:

Tipo de interrupção	Modo de operação	Endereço
<i>Reset</i>	Supervisor	0x00000000
Instrução indefinida	Indefinido	0x00000004
Interrupção de Software (swi)	Supervisor	0x00000008
<i>Prefetch abort</i>	<i>Abort</i>	0x0000000C
<i>Data abort</i>	<i>Abort</i>	0x00000010
Interrupção normal (IRQ)	IRQ	0x00000018
Interrupção rápida (FIQ)	FIQ	0x0000001C

Tabela 2.3: Vetor de interrupção (ARM LIMITED, 2005)

Prioridade	Interrupção
alta	Reset <i>Data abort</i> FIQ IRQ <i>Prefetch abort</i>
baixa	Instrução indefinida e interrupção de software (SWI)

Tabela 2.4: Ordem de prioridade das interrupções (ARM LIMITED, 2001b)

2.1.7.2 Entrada de interrupção

Executar uma interrupção necessita que o processador preserve o estado atual. Em geral, o conteúdo de todos os registradores (especialmente PC e CPSR) devem ser o mesmo depois de uma interrupção.

O processador ARM usa os registradores adicionais de cada modo para ajudar a salvar o estado do processador. Quando uma interrupção acontece, o R14 e o SPSR são usados para guardar o estado atual da seguinte maneira (ARM LIMITED, 2001b):

1. Preserva o endereço da próxima instrução (PC+4 ou PC+8, depende da interrupção) no apropriado LR (R14). Isso permite ao programa continuar do lugar de onde parou no retorno da interrupção.
2. Copia o CPSR para o apropriado SPSR.
3. Força os bits de modo do CPSR para um valor que corresponde ao tipo de interrupção.
4. Força o PC buscar a próxima instrução no vetor de interrupção.

O processador ARM7TDMI também pode ativar a *flag* de interrupção para desabilitar próximas interrupções.

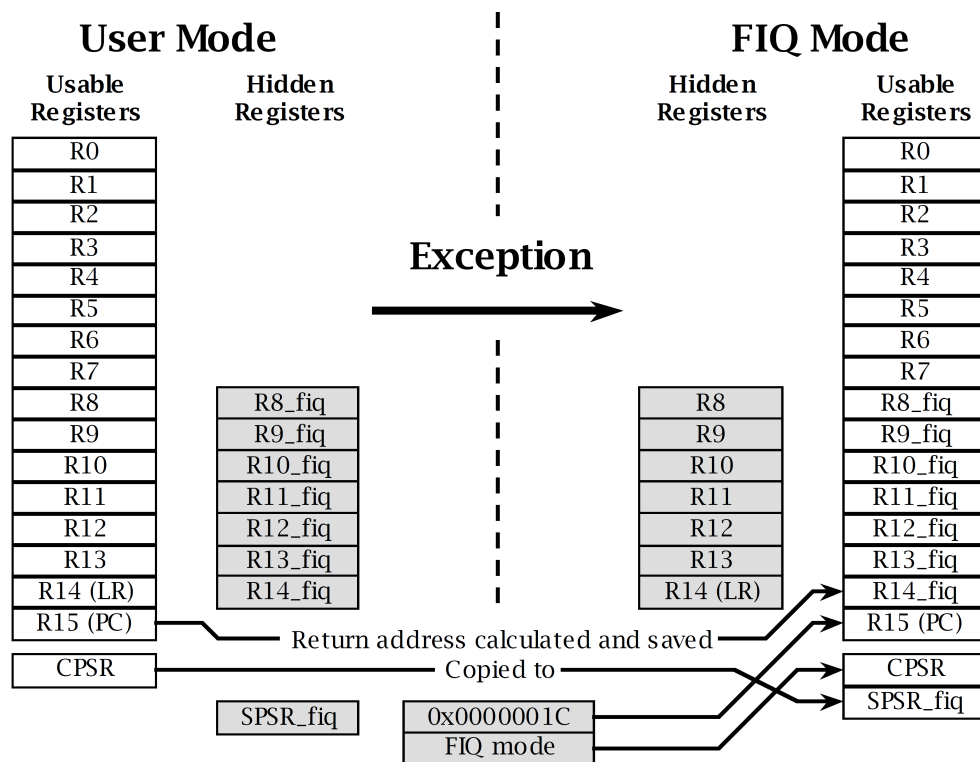


Figura 2.5: Esquema de uma interrupção no ARM7TDMI (ZAITSEFF, 2003)

2.1.7.3 Saída de interrupção

Quando uma interrupção é completada deve-se (ARM LIMITED, 2001b):

1. Mover o LR (R14), menos um *offset*, para o PC. O *offset* varia de acordo com o tipo de interrupção mostrada na figura anterior.
2. Copiar o SPSR de volta para o CPSR.
3. Desativa as *flags* de interrupção que foram ativadas na entrada.

2.1.7.4 Interrupções de software

Uma interrupção de software é uma interrupção inicializada inteiramente por um programa para entrar no modo Supervisor e assim poder utilizar alguma rotina particular, como operações de entrada e saída do sistema (ZAITSEFF, 2003).

Quando uma interrupção de software é executada, as seguintes ações são realizadas (ARM LIMITED, 2005):

1. Copia o endereço da próxima instrução no registrador LR_svc (R14_svc).

```
R14_svc = endereço da próxima instrução
```

2. Copia o CPSR no SPSR_svc.

```
SPSR_svc = CPSR
```

3. Ativa os bits de modo do CPSR com o valor correspondente ao modo Supervisor.

```
CPSR[4:0] = 0b10011 /* modo Supervisor */
```

4. Reforça o estado ARM colocando o bit T do CPSR à zero.

```
CPSR[5] = 0 /* estado ARM */
```

5. Desabilita as interrupções normais ativando o bit I do CPSR. Interrupções FIQ não são desabilitadas e podem continuar ocorrendo.

```
CPSR[7] = 1 /* desabilita interrupções normais */
```

6. Carrega o endereço do vetor de interrupções, 0x00000008, no PC.

```
PC = 0x00000008
```

Para retornar da operação de interrupção, é usada a seguinte instrução para restaurar o PC (a partir do R14_svc) e o CPSR (a partir do SPSR_svc):

```
MOVS PC, LR
```

2.1.7.5 Interrupções de hardware

Interrupções de hardware são mecanismos que permitem que um sinal externo (pedido de interrupção) interrompa a execução normal do programa corrente e desvie a execução para um bloco de código chamado de rotina de interrupção (KINOSHITA, 2007).

Interrupções são úteis, pois permitem que o processador manuseie periféricos de uma maneira mais eficiente. Sem elas, o processador teria que verificar periodicamente a entrada/saída de um dispositivo para ver se esse necessita de tratamento. Com elas, por outro lado, a entrada/saída do dispositivo pode indicar diretamente a ocorrência de um dado evento externo, que será tratado com maior facilidade e rapidez, de modo que o microprocessador não necessite consumir tempo de processamento para pesquisar a ocorrência de eventos externos.

O processador ARM fornece dois sinais que são usados pelos periféricos para pedir uma interrupção: o sinal de interrupção nIRQ e o sinal de interrupção rápida nFIQ. Ambos são ativados em nível baixo, ou seja, colocando o sinal em nível baixo gera-se a interrupção correspondente, se a interrupção não tiver sido desabilitada no CPSR (ZAITSEFF, 2003).

Quando uma interrupção de *hardware* IRQ (ou FIQ) é detectada, as seguintes ações são realizadas (ARM LIMITED, 2005):

1. Copia o endereço da próxima instrução a ser executada + 4 no registrador LR_irq (R14_irq). Isso significa que o LR_irq irá apontar para a segunda instrução a partir do ponto de pedido da interrupção.

$R14_irq = \text{endereço da próxima instrução} + 4$

2. Copia o CPSR no SPSR_irq.

$SPSR_irq = CPSR$

3. Coloca os bits de modo do CPSR para o valor correspondente ao modo IRQ.

$CPSR[4:0] = 0b10010 \text{ /* modo IRQ */}$
--

4. Reforça o estado ARM colocando o bit T do CPSR a zero.

$CPSR[5] = 0 \text{ /* estado ARM */}$
--

5. Desabilita as interrupções normais ativando o bit I do CPSR. Interrupções FIQ não são desabilitadas e podem continuar ocorrendo.

$CPSR[7] = 1 \text{ /* desabilita interrupções normais */}$

6. Carrega o endereço do vetor de interrupções, 0x00000008, no PC.

$PC = 0x00000018$

Assim que a rotina de interrupção é terminada, o processador retorna ao que estava fazendo antes através das seguintes ações:

1. Move o conteúdo do registrador LR_irq menos 4 para o PC.
2. Copia SPSR_irq de volta para CPSR.

A seguinte instrução executa os passos mostrados acima:

SUBS PC, R14,#4

Note que a instrução é SUBS, e não SUB: a instrução SUBS copia automaticamente SPSR no CPSR, mas apenas quando o registrador de destino é o PC (R15) e a instrução é executada em um modo privilegiado.

O processamento das *Fast Interrupt* (FIQ) é praticamente igual ao de uma interrupção normal (IRQ). As diferenças são que um conjunto diferente de registradores é usado (i.e. R14_fiq no lugar de R14_irq), que tanto as interrupções IRQ quanto as FIQ são desativadas (ou seja, os bits I e F do CPSR são ativados), e que o endereço do vetor de interrupção é 0x0000001C (ZAITSEFF, 2003).

2.1.8 Programando em C pra o ARM7TDMI

Neste item são apresentados alguns pontos importantes a serem considerados quando se esta programando em C para o processador ARM7.

2.1.8.1 Alocação de Registradores

O compilador tenta alocar um registrador do processador para cada variável local que encontra em uma função C. Ele tenta usar o mesmo registrador para diferentes variáveis locais se a utilização das variáveis não se sobrepõem. Quando há mais variáveis locais que registradores disponíveis, o compilador armazena as variáveis em excesso na pilha do processador (SLOSS; SYMES; WRIGHT, 2004).

2.1.8.2 Chamadas de Função

A *ARM Procedure Call Standard* (APCS) define como passar argumentos de função e obter valores de retorno.

Os primeiros quatro argumentos inteiros são passadas nos quatro primeiros registradores ARM: R0, R1, R2 e R3. Argumentos inteiros posteriores são colocados na pilha, como na figura 2.6. Se o valor de retorno for inteiro, este é obtido através do registrador R0 (SLOSS; SYMES; WRIGHT, 2004).

Esta descrição abrange apenas os argumentos de tipo inteiro ou ponteiro. Argumentos que ocupam o espaço de duas palavras, como *long long* e *double*, são passados em um par de

registradores consecutivos e retornam em R0, R1.

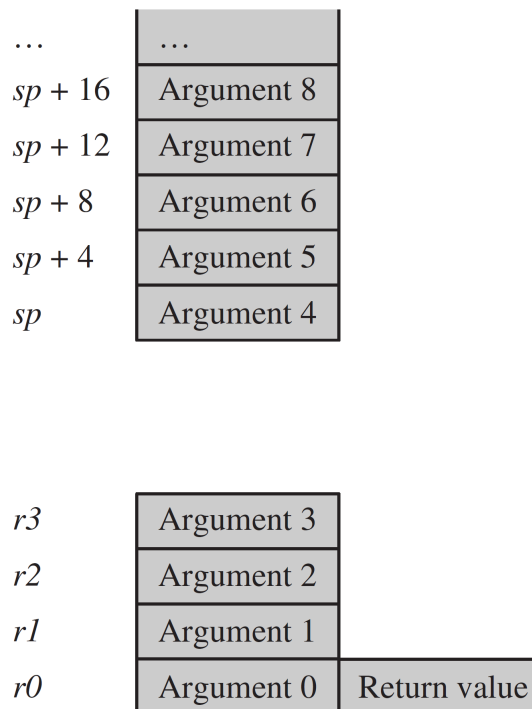


Figura 2.6: Passagem de argumentos (SLOSS; SYMES; WRIGHT, 2004)

2.2 A Placa Experimental Evaluator-7T

O principal elemento de hardware deste projeto é a placa experimental ARM Evaluator-7T, baseada no processador ARM7TDMI, um processador RISC de 32 bits capaz de executar o conjunto de instruções denominado Thumb.

Os principais elementos presentes na arquitetura da placa Evaluator-7T são os seguintes:

- Microcontrolador Samsung KS32C50100
- 512kB EPROM flash
- 512kB RAM estática (SRAM)
- Dois conectores RS232 de 9 pinos tipo D
- Botões de reset e de interrupção

- Quatro LEDs programáveis pelo usuário e um display de 7 segmentos
- Entrada de usuário por um interruptor DIP com 4 elementos
- Conector Multi-ICE
- Clock de 10MHz (o processador usa-o para gerar um clock de 50MHz)
- Regulador de tensão de 3.3V

A figura 2.7 mostra a organização desses elementos na placa experimental.

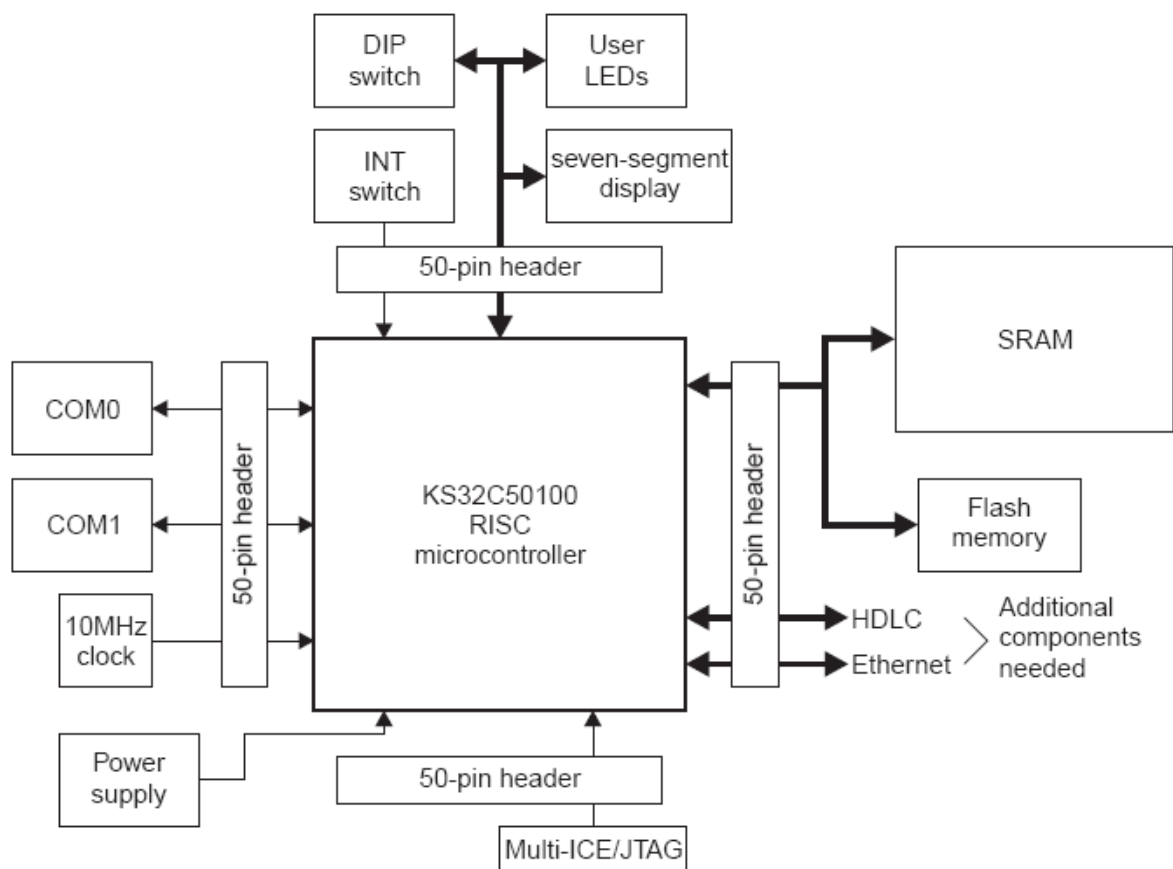


Figura 2.7: Arquitetura da placa Evaluator-7T. (ARM LIMITED, 2000)

Com relação à memória flash da placa, ela vem de fábrica com o bootstrap loader da placa e programa monitor de debug. O restante dela pode ser usado para os programas de usuário. A tabela 2.5 mostra a faixa de endereços de cada região da memória.

Já em relação às duas portas seriais presentes na placa, cada uma tem usos específicos. A primeira, chamada DEBUG, é usada pelo monitor de debug ou pelo programa bootstrap presente na placa. Ela está conectada ao UART1 do microcontrolador. A segunda, chamada

Tabela 2.5: Mapa da memória flash

Faixa de endereço	Descrição
0x01800000 a 0x01806FFF	Bootstrap loader
0x01807000 a 0x01807FFF	Teste de produção
0x01808000 a 0x0180FFFF	Reservado
0x01810000 a 0x0181FFFF	Angel
0x01820000 a 0x0187FFFF	Disponível para outros programas e dados

USER, é de uso genérico e está disponível para uso em programas. Ela está conectada ao UART0 do microcontrolador.

2.2.1 Bootstrap Loader

Como mencionado anteriormente, a memória flash da placa contém uma região reservada para os programas Bootstrap Loader (BSL) e o programa monitor de debug chamado Angel.

O BSL é o primeiro programa a ser executado pelo microcontrolador quando esta é ligada ou reiniciada. Suas principais funções são:

- Fazer a conexão com o computador através da porta serial e uma aplicação de terminal, como o HyperTerminal do Windows
- Prover a infraestrutura necessária à configuração da placa
- Prover ajuda ao usuário
- Gerenciar imagens de memória como um conjunto de módulos executáveis
- Carregar aplicações na SRAM e executá-las

2.2.1.1 Comunicação com o PC

Neste projeto, foi usado um PC com o sistema operacional Windows XP para fazer a comunicação com o BSL da placa Evaluator-7T. Essa comunicação é feita através de um cabo serial conectado à porta COM1 (Debug) da placa. Estando a placa conectada à porta serial e energizada com uma fonte de alimentação própria, pode-se estabelecer a comunicação com o BSL por meio do programa HyperTerminal. As configurações de comunicação utilizadas foram:

- Velocidade de transferência de 9600 bauds

- 8 bits de dados
- Sem paridade
- 1 bit de parada
- Sem controle de fluxo

Após a configuração adequada da placa, é preciso reiniciá-la, pressionando o botão SW1 (SYS RESET). Então, a placa envia a seguinte mensagem ao terminal:

```
ARM Evaluator7T Boot Strap Loader Release 1.01  
Press ENTER within 2 seconds to stop autoboot
```

Pressionando a tecla *Enter* em até dois segundos da exibição da mensagem acima, nenhum outro módulo da memória é executado, além do BSL. Desse momento em diante, o BSL exibe seu editor de linha de comando, a partir do qual é possível gerenciar, embarcar e executar programas na placa. A figura 2.8 mostra o HyperTerminal com o BSL carregado e aguardando um comando.

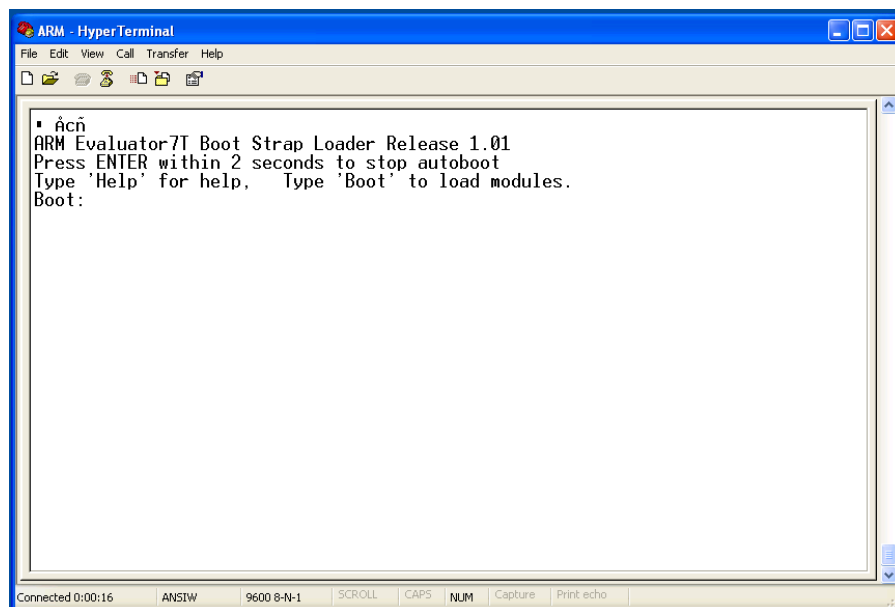


Figura 2.8: Editor de linha de comando do BSL via HyperTerminal

2.2.1.2 Carregando e executando programas via BSL

Após a compilação de um projeto, o ambiente de desenvolvimento cria uma imagem de memória em formato binário (extensão *.bin*). Essa imagem, no entanto, não pode ser carregada

diretamente na placa através do BSL. Ela deve ser convertida para o formato UUE (Unix-to-Unix Encoding), o qual é uma representação em arquivo texto do arquivo binário original. Neste projeto, foi utilizado para essa conversão o programa *uuencode* fornecido no CD-ROM que acompanha a placa Evaluator-7T.

Uma vez convertido o arquivo para o formato adequado, ele está pronto para ser enviado à Evaluator-7T. Para isso, pode-se usar dois diferentes comandos do BSL: *Download* ou *FlashLoad*.

O comando *Download* carrega uma imagem na memória RAM da placa. A sintaxe desse comando é:

```
download [<endereço>]
```

O parâmetro *<endereço>*, que é um número em base hexadecimal, indica em qual endereço da RAM a imagem será carregada. Se esse endereço não for especificado, a imagem é carregada na posição 0x8000.

Assim que o comando é executado, o BSL espera a transferência de um arquivo texto com a imagem de memória desejada. No HyperTerminal, isso é feito pelo comando “Enviar arquivo texto” e apontando para o arquivo desejado, no formato UUE. Terminada a transferência, o BSL informa quantos bytes foram recebidos e a posição de memória a partir da qual eles foram gravados.

Já o comando *FlashLoad* carrega uma imagem na placa e a salva diretamente na memória flash da mesma. Sua sintaxe é a seguinte:

```
flashload <endereço>
```

Neste comando, o parâmetro *<endereço>* é obrigatório, também é um número em base hexadecimal e especifica o endereço da memória flash no qual a imagem será gravada. O envio do arquivo é feito da mesma maneira que o comando *Download*. Como não há restrições quanto ao valor que o usuário pode inserir nesse comando, cabe a ele mesmo tomar cuidado para não escrever dentro da faixa de endereços de 0x01800000 a 0x0180FFFF, uma vez que é nessa área da flash que estão os módulos BSL e de teste de produção.

O comando *FlashLoad* não é o único que manipula a memória flash da placa no BSL. Existem também os comandos *FlashWrite* e *FlashErase*. O primeiro escreve na memória flash uma determinada área da RAM, enquanto que o segundo sobrescreve uma faixa de endereços da flash com 0xFF. As sintaxes desses comandos são:

```
flashwrite <endereço> <fonte> <comprimento>
```

```
flasherase <endereço> <comprimento>
```

Mais uma vez, é preciso exercer cautela durante a utilização desses comandos para não comprometer a área de memória onde se encontram os módulos BSL e de teste de produção.

Carregada a imagem na memória RAM ou na memória flash, ela está pronta para execução. Para executá-la, deve-se, primeiramente, verificar se o Program Counter (PC) do BSL está apontando para a posição de memória onde foi gravada a imagem. Isso é feito através do comando *PC*, cuja sintaxe está abaixo.

```
pc [<endereço>]
```

Esse comando permite verificar a posição a partir da qual o BSL iniciará a execução, se o parâmetro *<endereço>* não for especificado. Quando esse comando é feito com um argumento, o valor do PC é alterado para o valor do argumento inserido. Por exemplo, *pc 10000* coloca o PC na posição de memória 0x10000. Quando os comandos *Download* e *FlashLoad* são executados, o PC é atualizado automaticamente para o valor inserido no parâmetro *<endereço>* desses comandos.

O próximo passo para a execução da imagem pode ser feito com dois comandos diferentes: *Go* ou *GoS*. Ambos iniciam a execução de um programa a partir da posição de memória definida no PC. Enquanto o primeiro executa o programa em Modo Usuário, o segundo o faz em Modo Supervisor (SVC). Opcionalmente, pode-se inserir argumentos de entrada do programa quando esses comandos são chamados. A sintaxe deles é:

```
go [<argumentos do programa>]
```

```
gos [<argumentos do programa>]
```

Assim, o programa começa a executar na placa. Caso seja necessário retornar ao BSL, deve-se reiniciar a placa, pressionando-se o botão SYS RESET. Qualquer imagem que tenha sido carregada apenas na RAM será perdida.

2.2.2 Angel Debug Monitor

O monitor de debug Angel é fornecido conjuntamente com diversas placas da ARM e suas parceiras. Suas principais funcionalidades são:

- Função de depuração de código, incluindo inspeção de memória, download e execução

de imagens de memória, uso de breakpoints e execução passo-a-passo

- Inicialização da CPU e da placa e tratamento básico de exceções
- Uma biblioteca ANSI C completa, com uso de semihosting para prover serviços do computador host que não estão disponíveis na placa

Há duas maneiras pelas quais o Angel se comunica com o ambiente de desenvolvimento de software.

A primeira é através da biblioteca de interfaces chamada "Remote_A". Por ela, os depuradores se comunicam com um alvo do Angel quando fazem depuração ou execução de código.

A segunda é por meio de interrupções de software (SWI). O código do programa faz uma SWI para solicitar serviços dos Angel diretamente ou através da biblioteca C do toolkit.

2.3 O ambiente de desenvolvimento

O hardware descrito na seção 2.2 não pode realizar muitas tarefas se não houver o software adequado embarcado nele. Assim, o desenvolvimento de programas é parte fundamental do projeto. Para realizar tal tarefa, é necessária a existência de um ambiente de desenvolvimento que permita escrever, compilar, embarcar e depurar programas para a Evaluator-7T.

Neste projeto, foi utilizado o ambiente ARM Developer Suite (ADS) versão 1.2. Ele contém a IDE CodeWarrior e o debugger AXD. Ambos estão descritos em detalhes nas sub-seções abaixo.

2.3.1 CodeWarrior

O CodeWarrior é um ambiente integrado de desenvolvimento, ou seja, é um software que provê diversas funcionalidades para facilitar o desenvolvimento de programas. Dentre as funcionalidades que ele fornece, pode-se citar:

- Editor de código-fonte em C/C++ e ARM Assembly
- Compilador C/C++ para Assembly ARM e Thumb
- Automatização da compilação e geração de imagens de memória

TODO...

2.3.2 AXD Debugger

Escrever sobre o debugger aqui.

3 O SISTEMA OPERACIONAL KINOS

O principal objetivo do projeto é auxiliar o ensino de sistemas operacionais e da arquitetura ARM nas disciplinas de Sistemas Operacionais e Laboratório de Microprocessadores. Para tal, foi desenvolvido um *microkernel*, apelidado de KinOS, cujas funções básicas são o chaveamento de *threads* através de interrupção de *timer*, as chamadas de sistema, as rotinas de manipulação de *hardware*, funções de *mutex* e um *shell*.

3.1 Organização do código

A estrutura de arquivos do projeto pode ser vista na figura 3.1. Pode-se dividi-lo em cinco partes:

- **Raiz** Arquivos de inicialização da placa
- **Pasta “apps”** Programas que serão executados pelo *microkernel*
- **Pasta “interrupt”** Rotinas de tratamento de interrupção
- **Pasta “peripherals”** Rotinas de manipulação de *hardware*
- **Pasta “syscalls”** Chamadas de sistema
- **Pasta “mutex”** Rotinas do *mutex*

A pasta KinOS_Data não é considerada parte do projeto pois é utilizada pelo CodeWarrior para o armazenamento do código compilado.

3.1.1 Raiz

Os arquivos encontrados na raiz do projeto são responsáveis pela inicialização da placa e pela declaração de constantes globais. O arquivo `startup.s` contém a chamada inicial do

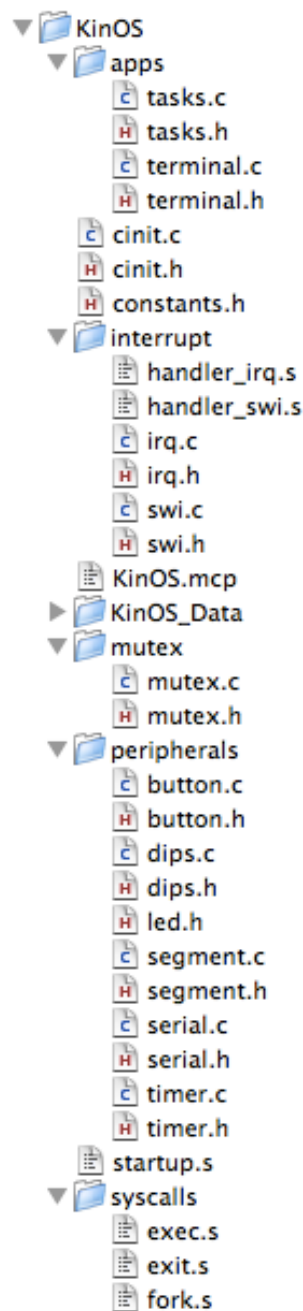


Figura 3.1: Estrutura de arquivos.

microkernel, onde toda parte de inicialização em *assembly* é feita. Já o arquivo `cinit.c` também contém a parte de inicialização, porém, o código está escrito em C. Finalmente, o arquivo `constants.h` é responsável por armazenar as constantes que são utilizadas em todo o projeto.

3.1.2 Pasta “apps”

No arquivo `tasks.c`, várias funções são declaradas, onde cada declaração é considerada uma *thread* pelo *microkernel*. Mais à frente, na seção 3.9, os programas exemplo serão descritos

com mais detalhe. Já no arquivo `terminal.c`, há o *shell* do sistema.

3.1.3 Pasta “interrupt”

Todas as rotinas que tratam e instalam interrupções – tanto de *hardware* quanto de *software* – estão localizadas nesta pasta. O arquivo `handler_irq.s` contém a rotina em *assembly* que trata das interrupções de *hardware*, as encaminha para a rotina específica de acordo com a sua fonte e faz o chaveamento de *threads*. O arquivo `irq.c` contém uma única rotina, que realiza a instalação da rotina de tratamento de interrupção tanto de *hardware* quanto de *software*. A rotina de tratamento de interrupção de *software* é feita no arquivo `handler_swi.s`, que identifica o tipo de interrupção e encaminha para alguma das chamadas de sistema, encontradas em `swi.c`.

3.1.4 Pasta “peripherals”

As rotinas de inicialização e controle dos periféricos se encontram todas nesta pasta. As do botão estão no arquivo `button`, da chave DIP no arquivo `dips`, do display de sete segmentos em `segment`, dos LEDs em `led` e do *timer* em `timer`.

3.1.5 Pasta “syscalls”

As chamadas de sistema estão escritas em *assembly* e se encontram em três arquivos, uma para cada chamada. São elas as chamadas `fork`, `exec` e `exit`.

3.1.6 Pasta “mutex”

No arquivo `mutex` há apenas as funções que permitem a exclusão mútua de código por espera ativa, feita através de um *mutex*.

3.2 Estruturas de dados

A fim de se facilitar a programação e o entendimento do projeto, foram criadas duas estruturas de dados que são acessadas em *assembly*. A primeira, o Process Control Block é responsável pelo armazenamento do estado de uma *thread*. Já a Lista de Threads realiza o controle de quais *threads* estão ativas.

3.2.1 Process Control Block

O Process Control Block (ou simplesmente PCB) é uma estrutura de dados que guarda todas as informações de uma *thread* que aguarda para ser executada enquanto outras estão ativas. Há um PCB para cada uma das nove *threads* e cada um ocupa 68 bytes. Ou seja, o espaço total ocupado pelos PCBs é de $9 \cdot 68 = 612$ bytes. Estes 68 bytes estão estruturados como explicitado na figura 3.2. Cada posição da tabela ocupa uma palavra (4 bytes). A primeira posição é em (base do PCB - 4), a segunda em (base do PCB - 8) e assim por diante. Como pode-se observar pela figura, as posições 1 a 15 ((base do PCB - 4) a (base do PCB - 60)) armazenam o conteúdo dos registradores r0 a r14 do modo *user* em ordem inversa. A posição 16 (base do PCB - 64) armazena o *link register* do modo IRQ, ou seja, o endereço de retorno da interrupção. Finalmente, a posição 17 armazena o registrador de estado do modo *user*. Estes registradores armazenados permitem estabelecer um retrato preciso do estado da *thread* quando houve o chaveamento e permite também que este estado seja restabelecido quando for o turno desta *thread* voltar a ser executada. A estrutura tem seu espaço reservado no arquivo `handler_irq.s`, e é nomeado com a variável `process_control_block`, que indica a base da estrutura. Cada um dos PCBs está logo a seguir do anterior. Por exemplo, a base do primeiro PCB está em (`process_control_block` - 68), do segundo em (`process_control_block` - $2 \cdot 68$) e assim por diante.

Offset	Task Register
-4	r14_usr
-8	r13_usr
-12	r12_usr
-16	r11_usr
-20	r10_usr
-24	r9_usr
-28	r8_usr
-32	r7_usr
-36	r6_usr
-40	r5_usr
-44	r4_usr
-48	r3_usr
-52	r2_usr
-56	r1_usr
-60	r0_usr
-64	r14_irq
-68	SPSR

Figura 3.2: Estrutura de dados do PCB. Fonte: (SLOSS, 2001)

3.2.2 Vetor de threads

O vetor de *threads* é uma lista que armazena quais das *threads* estão ativas e quais não estão, a fim de se identificar quais devem ser colocadas em execução. Cada identificador ocupa 4 bytes, e pode ter os valores 0 (inativo) ou 1 (ativo). Como há 9 *threads*, o tamanho deste vetor é de $4 \cdot 9 = 36$ bytes. Seu espaço é reservado no arquivo `handler_irq.s`, com o nome de `thread_array`. No exemplo na figura 3.3 pode-se ver que as *threads* 1, 2 e 4 estão ativas, enquanto que as outras não estão.

T1	T2	T3	T4	T5	T6	T7	T8	T9
1	1	0	1	0	0	0	0	0

Figura 3.3: Vetor de *threads*.

3.3 Configuração de hardware e software

Nesta seção são apresentados os modos como o *hardware* e o *software* descritos anteriormente são utilizados. Será indicado como foi feito o particionamento da memória, a utilização dos modos do processador e os modos de teste do código.

3.3.1 Memória

A memória volátil da placa foi estruturada como indicado na figura 3.4. Para todo espaço das pilhas, programas, código, vetor de interrupções e área de dados, o espaço disponível é de 128KB (de 0x0 a 0x20000). Como pôde ser visto na seção 2.1.7, a memória entre 0x0 e 0x20 contém o vetor de interrupções e deve ser reservado. A pilha do modo SVC começa no endereço 0x7F80, cresce para baixo e não deve invadir a área reservada para o vetor de interrupção. Já a pilha do modo IRQ, começa no endereço 0x8000, também cresce para baixo e não deve invadir o espaço reservado para a pilha do modo SVC. O código do *kernel* e dos programas começa no endereço 0x8000, mas ao contrário da pilha do modo SVC, cresce para cima. Logo após o código, temos uma área reservada para os dados globais. Finalmente, as pilhas do modo *user* começam no endereço 0x20000 e crescem para baixo. Cada uma tem um offset relativo à anterior de 4048 bytes.

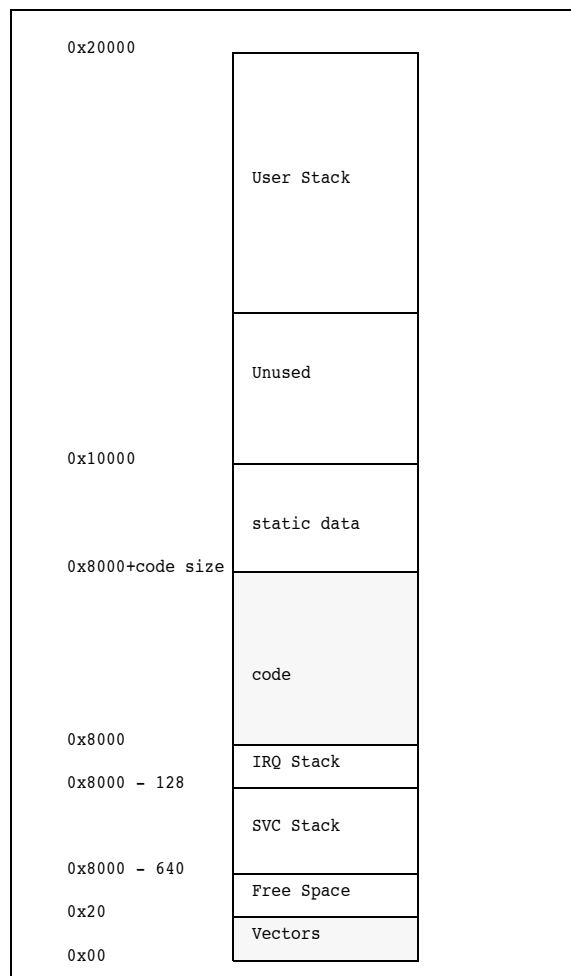


Figura 3.4: Estrutura da memória. Fonte: (SLOSS, 2001)

3.3.2 Modos do processador

Dentre os sete modos do processador, apenas quatro deles são utilizados: o modo de usuário (*user*), o modo de serviço (*SVC*), o modo de sistema (*SYS*) e o modo de interrupção (*IRQ*). O primeiro é o modo não privilegiado no qual as *threads* são executadas. O segundo, é o modo de inicialização do *kernel* e de execução das chamadas de sistema, que é privilegiado. Já o terceiro, é idêntico ao modo de usuário, mas com privilégios. Ele é utilizado na inicialização do sistema para definir a pilha do modo de usuário. Finalmente, o quarto é um modo que também é privilegiado, mas que é usado quando há interrupções de *hardware* e portanto, é usado quando há o chaveamento de *threads* (interrupção de *timer*) ou qualquer outra interrupção que não a de *software*. É importante ressaltar que os modos privilegiados quando chamados por interrupção desabilitam outras interrupções. Isso bloqueia interrupções aninhadas, essencial para o funcionamento do código.

3.3.3 Modos de teste

Depurar o código com a placa não é possível em todas as situações. Quando o código que está sendo executado está dentro de uma região onde as interrupções estão desabilitadas, como no código de tratamento de interrupção, não se pode fazê-lo. Para contornar tal problema, foi utilizado o emulador disponível na IDE CodeWarrior, o ARMulator. Como ele foi desenvolvido para vários modelos de placa, utiliza endereços de periféricos diferentes da placa Evaluator 7-T e não têm o módulo Angel de *debug*. Para manter a compatibilidade entre o emulador e a placa nas partes onde o código se diferencia, como na inicialização do *timer*, foram colocados ambos códigos. A seleção de qual dos dois será executado depende de uma variável global *emulator*, que é declarada no arquivo *constants.h*. Caso seja 1, o código executado é o do emulador, caso seja 0, o código da placa com Angel e caso seja 2, o código para a placa sem o Angel. Uma outra vantagem do código no emulador é que ele permite com que ele possa ser testado sem a presença da placa.

3.3.4 Angel

O Angel é um programa contido na ROM da placa que realiza a comunicação entre a mesma e o computador que efetuou o upload do código. Além de permitir com que o código seja carregado na placa, o Angel realiza o processo de *debug* do código durante a execução. Para isso, deve haver uma comunicação constante entre a placa e o computador, que é feita através de interrupções. Uma vez que a placa é iniciada, o endereço do vetor de interrupções responsável pelas interrupções de *hardware* e se *software* apontam para um endereço pré-estabelecido do Angel.

Caso se queira adicionar alguma outra rotina de tratamento de interrupções, como é o caso deste projeto, deve-se encadear o Angel (como será descrito na seção 3.4.5) quando a rotina instalada não consegue tratar a interrupção. Isto é necessário para que a comunicação com a placa não seja perdida.

3.4 Inicialização

O início do programa se dá no arquivo *assembly statup.s*. Nele, são feitas todas as operações que não podem ser feitas no código em C, como a inicialização das pilhas ou a criação da tabela de threads. Após esta etapa, há a inicialização em C, feita no arquivo *cinit.c*, que inicializa periféricos, instala rotinas de tratamento e inicia a primeira thread em

modo usuário. A rotina completa de inicialização pode ser vista no esquema da figura 3.5.

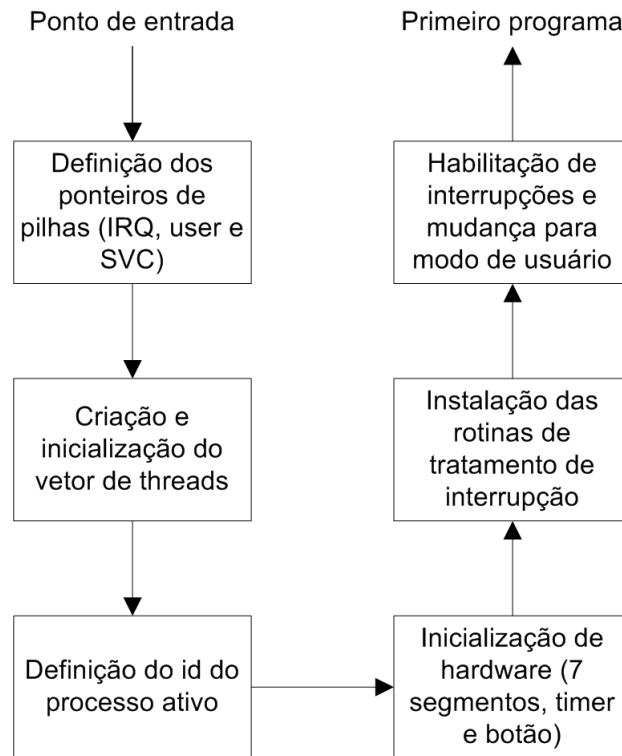


Figura 3.5: Fluxograma de inicialização.

3.4.1 Ponto de entrada e tipo de código

O ponto de entrada do código é indicado pela instrução `ENTRY`. Por padrão, o compilador assume que o código de entrada é ARM. Como descrito anteriormente, há dois tipos de assembly, o ARM e o THUMB. No microkernel, é utilizado apenas código ARM, já que ele fornece mais instruções e favorece a legibilidade. Um ponto negativo deste tipo de código é seu maior espaço ocupado na memória, mas isso não vem a ser um grande problema, pois temos espaço suficiente.

3.4.2 Pilhas

Antes de poder utilizar as pilhas é preciso que elas sejam inicializadas em cada um dos modos que virão a ser utilizados. Neste microkernel, são utilizados os modos de serviço, usuário/sistema e de interrupção. O modo como isto é feito é descrito abaixo:

```

MOV    r0 , #0xC0|0x12      ; r0 = 0xC0 or 0x12 (0xC0 = IRQ disabled , 0x12 =
    IRQ mode)
MSR    CPSR_c, r0           ; status_register = r0
  
```

```
MOV    sp , #0x8000    ; stack pointer = 0x8000
```

A primeira instrução copia para r0 o que será substituído no registrador de estado. Neste exemplo, está se desabilitando as interrupções e mudando o modo do processador para o modo de interrupção. Em seguida, os dados do registrador r0 são colocados no registrador de estado. Uma vez que o estado foi alterado, pode-se mudar o ponteiro de pilha, que neste caso aponta para o endereço 0x8000. Uma operação semelhante pode ser feita tanto no modo de serviço quanto no modo de usuário, usando os endereços de pilha indicados anteriormente. Porém, se o estado for alterado para o modo de usuário fica impossível de se alterar o estado novamente. Para se resolver este problema, ao invés de se mudar para o estado de usuário, muda-se para o estado de sistema. Este é o mesmo modo que o de usuário (usa a mesma pilha e registradores), mas permite que o modo seja alterado novamente.

3.4.3 Vetor de threads e número da thread

O outro ponto importante da inicialização do código em assembly é a criação do vetor de threads. Para tal, temos de definir que todos os processos exceto o primeiro são inicializados desabilitados. Isto é feito com o código apresentado a seguir:

```
; Initializes the thread array with zeros (0 = thread disabled ,
; 1 = thread enabled)
LDR    r0 , =thread_array    ; r0 = thread_array start address
MOV    r1 , #1                ; r1 = 1
STR    r1 , [r0]              ; address(r0) = r1
MOV    r1 , #0                ; r1 = 0 (disabled)
MOV    r2 , #0                ; r2 = 0
init_thread_array_loop
ADD    r2 , r2 , #4            ; r2 = r2 + 4
CMP    r2 , #36                ; r2 = 36?
BEQ    set_active_thread      ; if yes , go to set_active_thread
ADD    r3 , r0 , r2            ; r3 = r0 + r2
STR    r1 , [r3]              ; address(r3) = r1
B      init_thread_array_loop  ; return to init_thread_array_2
```

Nele, r0 armazena a base do vetor, que coincide com o espaço relativo à primeira thread. r1 contém o dado que será colocado na posição de memória. Na posição este valor é 1, e nos demais 0. r2 contém o offset que será somado à base para o cálculo do endereço absoluto, armazenado em r3. O algoritmo funciona inicialmente colocando 1 na base. Após isso, entra em um loop que aumenta o offset de 4 em 4 e coloca 0 em todos os outros espaços.

Ainda na inicialização em assembly, deve-se definir o número da thread que está sendo executada. Este dado é armazenado na variável `current_thread_id`. Pode-se ver abaixo como é definido o id do primeiro processo para 1:

```
LDR    r0, =current_thread_id ; r0 = current thread id address
MOV    r1, #1                  ; r1 = 1
STR    r1, [r0]                ; current thread id = 1
```

Finalmente, a inicialização em C pode ser iniciada. A chamada é feita definindo como endereço de retorno a função `C_entry` e colocando este mesmo endereço no process counter.

```
LDR    lr, =C_Entry            ; link register = C entry
MOV    pc, lr                  ; process counter = C entry
```

3.4.4 Periféricos

Para alguns periférico da placa, como o display de sete segmentos, o timer e os botões, há uma rotina de inicialização que os habilita e define suas configurações. Suas chamadas são `segment_init()`, `timer_init()` e `button_init()` respectivamente. Estas funções se encontram nos arquivos de cada um dos periféricos e são executadas logo no início da etapa C do processo de inicialização da placa.

3.4.5 Instalação do tratamento de interrupção

Como descrito anteriormente, caso uma interrupção de hardware ocorra, a instrução no endereço 0x18 é executada e caso seja uma interrupção de software, a instrução no endereço 0x08. Toda vez que se reinicia a placa, são colocados nestes endereços uma instrução que realiza um desvio para a rotina Angel, descrita anteriormente.

Porém, se algum dos periféricos vai ser utilizado, a interrupção gerada por esse periférico não deve desviada para o Angel, e sim para uma rotina adequada. Para poder identificar qual a origem da interrupção e desviar para a rotina correta, devemos instalar uma nova rotina no vetor de interrupções, substituindo o desvio para o Angel. A instalação da rotina se dá através do desvio para a tal rotina. Todavia, não se pode apenas descartar o endereço do Angel, já que caso não se identifique a origem da interrupção, ainda deve-se desviar para ele. Este processo pode ser observado na figura 3.6. Nele, *Handler2* é a rotina de tratamento de interrupções, e *Handler1* é o Angel.

A instalação da rotina de tratamento de interrupção é a mesma para interrupções de

hardware e de software se dá abaixo:

```
/* Angel branch instruction */
unsigned Angel_branch_instruction;
/* Angel instruction */
unsigned *Angel_address;
/* Getting Angel branch instruction */
Angel_branch_instruction = *vector_address;
/* Separate the instruction from the address */
Angel_branch_instruction ^= 0xe59ff000;
/* Calculating absolute address */
Angel_address = (unsigned *) ((unsigned)vector_address +
    Angel_branch_instruction + 0x8);
/* Store address in the proper position */
if ((unsigned)vector_address == 0x18) {
    Angel_IRQ_Address = *Angel_address;
}
else {
    Angel_SWI_Address = *Angel_address;
}
/* Inserting handler instruction in the vector table */
*Angel_address = handler_routine_address;
```

Os parâmetros de entrada desta função são `handler_routine_address`, o endereço da rotina de tratamento de interrupção e `vector_address`, um ponteiro para a posição no vetor de interrupções onde será instalada a rotina. Sucintamente, o que esta rotina realiza é obter a instrução que está em `vector_address`, aplica uma máscara à rotina para obter apenas o endereço e o salva em uma das variáveis: `Angel_IRQ_Address` caso se esteja instalando a rotina de interrupção de hardware ou `Angel_SWI_Address` caso seja a de software, além de colocar a nova instrução no vetor de interrupções.

Um fator importante que deve ser ressaltado a importância do Angel quando se está usando a placa. Como já descrito anteriormente, o Angel se utiliza das interrupções de hardware e software para se comunicar com a placa. Portanto, se apenas modificarmos o código e substituirmos a instrução que está contida no vetor de interrupção, essa comunicação não se realiza e tanto a placa quanto o programa de debugger travam. Para solucionarmos este problema, devemos passar para a rotina de tratamento de interrupção os endereços que estavam anteriormente no vetor de interrupção, para o caso da interrupção ser do Angel, a rotina correta ser executada. Já no caso em que o código é apenas simulado no emulador, não é preciso armazenar o endereço do Angel.

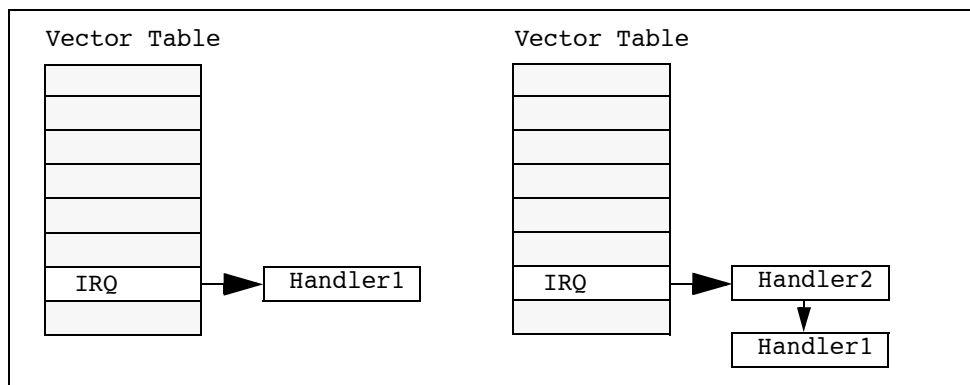


Figura 3.6: Encadeamento de interrupções. Fonte: (SLOSS, 2001)

3.4.6 Interrupção de timer

A interrupção de timer é utilizada neste projeto para realizar o chaveamento entre as threads. Uma vez que haja a interrupção, o estado da thread atual é salva e a próxima thread é colocada em processamento. Para utilizá-la, devemos tanto habilitar quanto iniciar o timer. Essas tarefas são executadas com duas rotinas, sendo que a primeira já foi descrita anteriormente. Já o início do timer é dado pela função `timer_start()`.

3.4.7 Habilitando interrupções

O último passo antes de se começar a executar o código do primeiro programa é habilitar simultaneamente o modo de usuário e as interrupções. Como isso só pode ser feito por código assembly, temos de usar a instrução especial de C `__asm`, conforme o exemplo abaixo

```
__asm {
    MOV    r1, #0x40|0x10
    MSR    CPSR_c, r1
}
```

O registrador `r1` recebe `0x40`, que indica a habilitação das interrupções e `0x10` que altera para o modo de usuário. Logo em seguida, o conteúdo deste registrador é passado para o registrador de estado. Finalmente, o primeiro programa é chamado com a função `shell()`.

3.5 Chaveamento de processos

O chaveamento de processos é realizado inteiramente com o assembly escrito no arquivo `handler_irq.s`. Ele consiste em sete passos, indicados na figura 3.7.

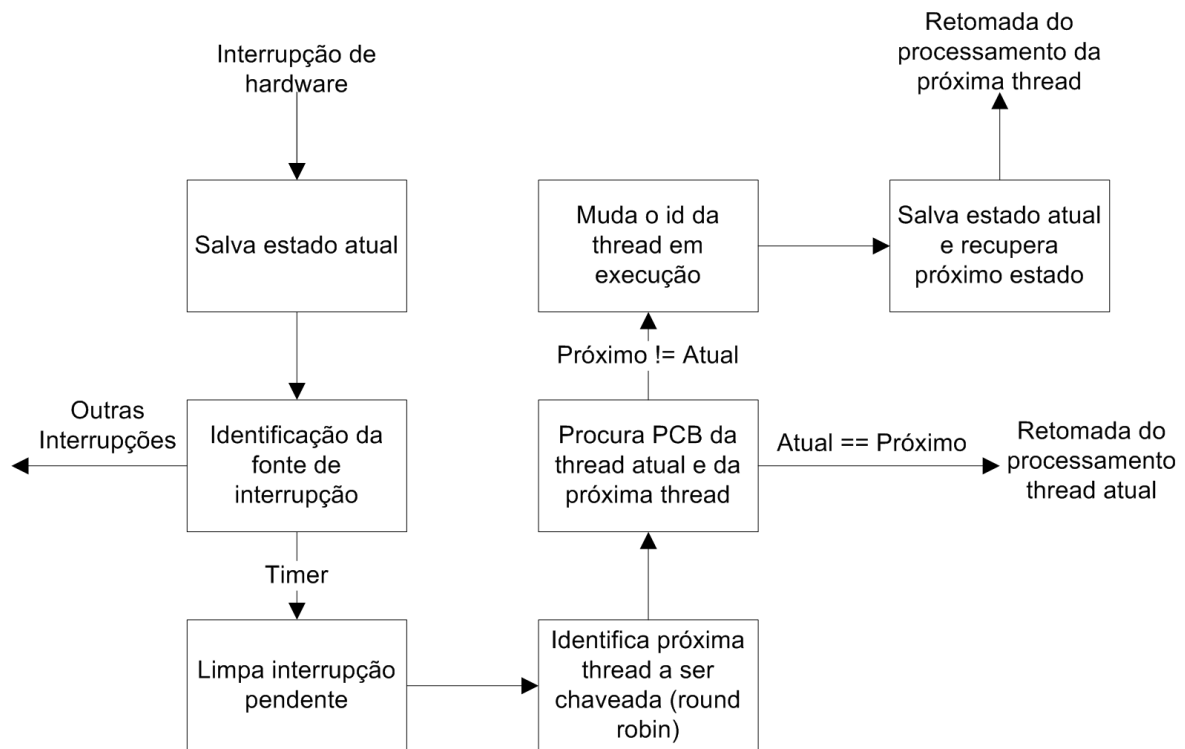


Figura 3.7: Chaveamento de processos.

3.5.1 Identificação da interrupção

```

STMFD sp!, {r0 - r3, lr}    ; Stacking r0 to r3 and the link register
LDR  r0, IRQStatus          ; r0 = irq type address
LDR  r0, [r0]                ; r0 = irq type
TST  r0, #0x0400             ; irq type == 0x0400?
BNE  handler_timer           ; If yes, go to handler_timer
TST  r0, #0x0001             ; irq type = 0x0001?
BNE  handler_button          ; If yes, go to handler_button
LDMFD sp!, {r0 - r3, lr}    ; If it is not any of them, restore r0-r3 and
                             lr
LDR  pc, Angel_IRQ_Address ; and branch to the Angel routine
  
```

Uma vez que há a interrupção de timer, a chamada de interrupção de hardware que se encontra no vetor de interrupção é executada. Durante a instalação da rotina de tratamento de interrupção de hardware, colocou-se nesta posição a rotina `handler_board_angel` caso se estivesse usando a placa com o Angel, a rotina `handler_board_no_angel` caso se estivesse usando a placa sem o Angel ou a rotina `handler_emulator` caso estivesse usando o emulador. A diferença é que enquanto a primeira e a segunda tentam identificar qual a fonte de interrupção, a terceira já assume que a fonte é o timer, já que não há outros periféricos no emulador. Deve-se armazenar toda informação contida nos registradores que são alterados durante o processo

de tratamento de interrupção. Para tal, empilhamos os valores dos registradores r0 a r3, usados durante a rotina de chaveamento, a fim de que nenhum dado se perca durante o processo.

No caso do uso da placa, a fonte da interrupção se encontra no endereço 0x03ff4004, identificado com a variável INTPND. Se o valor contido neste endereço é 0x0400, a fonte foi uma interrupção de timer, caso seja 0x0001, a fonte foi o botão da placa e caso contrário, a fonte foi o Angel. No primeiro caso, há um desvio para a rotina `handler_timer`, no segundo para a rotina `handler_button` e na terceira, para o endereço salvo durante a instalação de rotina de tratamento.

3.5.2 Limpeza da interrupção de timer

Quando é identificada a interrupção de timer, deve-se limpar a interrupção de timer, a fim de que ele possa interromper novamente no futuro. Para tal, executa-se a rotina `timer_irq`, encontrada no arquivo `timer.c`. Como não podemos garantir que a rotina em C manterá intactos os registradores, temos de salvar todos e recupera-los após a chamada. Abaixo podemos observar o código que realiza o salvamento e a recuperação destes registradores.

```

STMFD sp!, {r4 - r12}      ; Stack the rest of the registers (r4-r12)
BL    timer_irq             ; Clear timer interruption
LDMFD sp!, {r4 - r12}      ; Load r4-12 registers again

```

Os registradores r0 a r3 não precisam ser salvos ou recuperados, pois no início da rotina de tratamento eles já foram empilhados para recuperação futura.

3.5.3 Identificação da próxima thread

O método de escolha da próxima thread que será posta em execução é escolhida pelo método *round-robin*, ou seja, a próxima thread é escolhida por ordem numérica. O código para tal tarefa é apresentado abaixo:

```

CMP    r0, #9               ; r0 == 9? (it is the last thread?)
BEQ    last_thread          ; If yes, branch last_thread
ADD    r1, r0, #1           ; If not, r1 = r0 + 1
B      next_thread          ; and branch to next_thread
last_thread
MOV    r1, #1               ; r1 = 1
next_thread
SUB    r2, r1, #1           ; r2 = r1 - 1
MOV    r3, #4               ; r3 = 4

```

```

MUL    r2, r3, r2          ; r2 = r2 * r3
LDR     r3, =thread_array   ; r3 = thread_array bottom address
ADD     r2, r2, r3          ; r2 = r3 + r2
LDR     r2, [r2]            ; r2 = thread array content
CMP     r2, #1              ; thread array content = 1?
BEQ     set_addresses       ; If yes, branch to set_addresses
                                ; Send to the next step the next active
                                ; thread in r1
MOV     r0, r1              ; If not, r0 = r1
B       get_next_taskid_loop ; and loop to get_next_taskid_loop

```

Nele, r0 inicia com o número da thread atual. Caso ele seja igual a 9, a última thread da lista, deve-se iniciar novamente a procura desde a thread 1. Caso contrário, inicia-se com o próximo número. O resultado é armazenado em r1, onde se encontra o número da próxima thread. O valor em r1 é incrementado sucessivamente até encontrar um ponto no vetor de threads que tenha o valor 0, indicando que a thread não está ativa. O cálculo da posição de memória é dado a partir da seguinte função: $(r1 - 1) * 4 + \text{bottom}$ = posição relativa à thread r1, onde bottom é o endereço do início do vetor e 4 é o tamanho de cada espaço dentro do vetor.

3.5.4 Localização dos PCBs

A rotina de troca de processos tem como entrada duas variáveis: o PCB da thread atual e o PCB do próxima thread. Para obter tais dados, é necessário o número da thread atual e da thread que será colocada em execução. Como visto nos itens anteriores, estes dados já foram obtidos. Pode-se então aplicar o seguinte algoritmo:

```

LDR     r2, =current_thread_id ; r2 = current thread id address
LDR     r2, [r2]                ; r2 = current thread id
CMP     r2, r1                  ; Is r2 = current thread id ==
                                ; next thread id
BEQ     no_thread_switch        ; If yes, branch to no_thread_switch
; Setting current_task_addr
MOV     r0, #68                 ; Else start thread switch. r0 = 68
MUL     r0, r2, r0              ; r0 = current thread id * 68
LDR     r2, =process_control_block ; r2 = PCB bottom
ADD     r0, r0, r2              ; r0 = PCB bottom + id * 68
LDR     r2, =current_task_addr  ; r2 = current task addr addr
STR     r0, [r2]                ; current_task_addr = r0
; Setting next_task_addr
MOV     r0, #68                 ; r0 = 68

```

```

MUL    r0, r1, r0          ; r0 = next thread id * 68
LDR     r2, =process_control_block ; r2 = PCB_bottom
ADD     r0, r2, r0          ; r0 = PCB bottom + next id * 68
LDR     r2, =next_task_addr    ; r2 = next_task_addr addr
STR     r0, [r2]            ; next_task_addr = r0

```

O primeiro ponto checado é se a thread atual é igual à thread que vai ser substituída. Caso isso se confirme, o chaveamento se encerra e nada ocorre. Caso contrário, o cálculo dos endereços dos PCBs é iniciado. A fórmula utilizada é: $PCB_{id} = (id - 1) * 68 + base$, onde id é o número da thread e $base$ é o endereço do início dos PCBs. Ao fim do cálculo, estes dados são armazenados nas variáveis `current_task_addr` e `next_task_addr`, que serão utilizadas na próxima etapa do processo.

3.5.5 A troca de processos

A troca de processos se dá em poucos passos usando-se instruções especiais que permitem que haja um grande número de dados empilhados/desempilhados com apenas uma instrução. Inicialmente zera-se a pilha do modo de interrupção e restabelece-se os registradores `r0` a `r3`, que estavam empilhados desde o início da rotina de tratamento. Nota-se que o ponteiro não é totalmente zerado, ele é colocado em uma posição 20 bytes acima do esperado. Isto se dá porque há empilhadas 5 palavras (`r0` a `r3` e o link register) que logo em seguida virão a ser desempilhadas.

Depois disso, muda-se o endereço do ponteiro de pilha para o PCB do processo atual. Um truque vem no próximo passo: empilha-se todos os registradores com o ponteiro de pilha apontando para a posição ($base - 60$) do PCB. Deste modo, em uma única instrução todos os registradores são colocados em suas respectivas posições. Como a estrutura do PCB foi feita tendo este processo em mente, a posição dos dados dos registradores cai exatamente como foi descrito na figura 3.2. Após o armazenamento do estado atual, muda-se novamente o endereço do ponteiro de pilha para o PCB da próxima instrução. Do mesmo modo que o armazenamento, desempilha-se os o valor dos registradores, que são exatamente como estava empilhado este processo quando foi armazenado.

```

; Reset and save IRQ stack
LDR     r0, =irq_stack_pointer    ; r0 = irq_stack_pointer addr
MOV     r1, sp                    ; r1 = irq stack pointer
ADD     r1, r1, #5*4              ; r1 = irq stack pointer + 5 (# of data in
                                ; the stack, r0-r3, lr) * 4 (size of a word)
STR     r1, [r0]                  ; irq_stack_pointer = irq stack pointer

```

```

; without the data that will be removed next
LDMFD    sp!,{r0-r3,lr}      ; Restore the remaining registers
; Load and position r13 to point into current PCB
LDR    r13, =current_task_addr ; r13 = current task PCB bottom address
address
LDR    r13, [r13]             ; r13 = current task PCB bottom address
SUB    r13, r13,#60           ; r13 = current task PCB bottom address - 60
; to point to the right place for the stacking
; (next step)
; Store the current user registers in current PCB
STMIA    r13, {r0-r14}^      ; Stacks the r0-r14 registers in the PCB
MRS    r0, SPSR              ; r0 = status register
STMDB    r13, {r0,r14}       ; Stacks r0 and r14
; Load and position r13 to point into next PCB
LDR    r13, =next_task_addr   ; r13 = next task PCB bottom address
address
LDR    r13, [r13]             ; r13 = next task PCB bottom address
SUB    r13, r13,#60           ; r13 = next task PCB bottom address - 60
; to point to the right place for the stacking
; (next step)
; Load the next task and setup PSR
LDMNEDB    r13, {r0,r14}      ; Restore r0 and r14 (IRQ mode)
MSRNE    spsr_cxsf, r0        ; Restore status register
LDMNEIA    r13, {r0-r14}^     ; Restore r0-r14 for the user mode
NOP      ; NOP! (required for the above instruction)
; Load the IRQ stack into r13_irq
LDR    r13, =irq_stack_pointer ; r13 = stack pointer address address
LDR    r13, [r13]             ; Restore previous stack pointer
B    return                   ; Go to the end

```

3.5.6 Retorno à execução da nova rotina

Como os registradores, o ponteiro de pilha, o endereço de retorno e o registrador de estados já estão com os dados do próximo processo, deve-se apenas fazer com que a instrução imediatamente posterior à aquela executada antes da interrupção seja executada. Porém, o pipeline do processador fez com que o endereço da instrução duas vezes à frente tivesse sido armazenado. Para compensar isso, deve-se subtrair o tamanho de uma instrução (4 bytes) do endereço que vai ser colocado no process counter. Todo este processo é feito com apenas uma instrução: `SUBS pc, r14, #4`, que simultaneamente decrementa do endereço de retorno 4 e coloca o resultado no process counter.

3.6 Chamadas de sistema

Uma system call é uma interrupção de software causada pelo kernel para a execução de código que necessita de privilégios para ser executado. Como uma interrupção de hardware, uma vez que é causada, ela executa a instrução apontada no vetor de interrupções, que foi instalada anteriormente na inicialização do sistema. A rotina de tratamento está localizada no arquivo `handler_swi.s` e é executada em modo SVC. As únicas instruções que chamam tais system calls são as rotinas `fork`, `exec` e `exit`.

3.6.1 Propriedades gerais

Uma vez que uma system call é chamada, umas das funções encontradas em `swi.c` é invocada. O motivo para este passo intermediário é que todas as chamadas de sistema do kernel devem ter a mesma identificação junto à rotina de tratamento. Neste caso, todas são passadas com o primeiro parâmetro como 0. Além disso, todas devem passar o mesmo número de parâmetros, pois todas estão invocando a mesma função, chamada de `syscall` que também é realizado nesta etapa.

Uma vez que a chamada `syscall` é feita, ocorre uma interrupção de software. O procedimento que se passa neste caso é muito parecido com o de uma interrupção de hardware.

```

STMFD    sp!,{r0-r12,lr}      ; Stack registers r0-12 and link register
LDR      r0,[lr,#-4]           ; Calculate address of SWI instruction (r0 = lr-4)
BIC      r0,r0,#0xff000000     ; Mask off top 8 bits of instruction to give SWI
                                ; number
LDR      r1,Angel_SWI_Number   ; r1 = Angel SWI Number
CMP      r0,r1                 ; Compare SWI number to angel interrupt number
BEQ      goto_angel            ; If it is angel interrupt, branch to goto_angel
MOV      r1,#0                 ; r1 = 0
CMP      r0,r1                 ; Compare SWI number to r1
BEQ      os_swi                ; If it is OS SWI, branch to os_swi

```

Novamente há uma rotina de identificação da fonte de interrupção, que pode vir a ser uma do sistema operacional, ou do Angel. O primeiro passo desta rotina é o empilhamento de todos os registradores, para poder futuramente restaurar o estado atual. Em seguida, ocorre a identificação em si, onde uma máscara de bits é aplicada para se obter o identificador da interrupção. Caso ela seja `Angel_SWI_Number`, o estado do processador é restaurado e há um desvio para a instrução previamente armazenada durante a instalação. Caso ela seja 0, o valor estabelecido para o sistema, há um desvio para outro código que identifica quais das

chamadas de sistema foi ativada.

Esta nova identificação pode ser observada abaixo. O primeiro passo é restaurar e armazenar novamente os valores dos registradores, já que na arquitetura ARM os valores passados pelos parâmetros de uma função são passados pelos primeiros registradores. Neste caso, r1 contém o tipo da chamada. Dependendo de qual for o valor, há desvios para `pre_routine_fork`, `pre_routine_exec` e `pre_routine_exit`

```
LDMFD sp!,{r0-r12,lr}    ; Restore r0-r12 registers and link registers
STMFD  sp!,{r0-r12,lr}    ; and stores them again (in order to clean the
                           registers)
MOV    r1, #0              ; r1 = 0
CMP    r0, r1              ; Compare the first parameter to 0
BEQ    pre_routine_fork    ; If it is equal, branch to the fork
MOV    r1, #1              ; r1 = 1
CMP    r0, r1              ; Compare the first parameter to 1
BEQ    pre_routine_exec    ; If it is equal, branch to the exec
MOV    r1, #2              ; r1 = 2
CMP    r0, r1              ; Compare the first parameter to 2
BEQ    pre_routine_exit    ; If it is equal, branch to the exit
LDMFD  sp!,{r0-r12,pc}^    ; If it is an unidentified syscall, go back to the
                           program,
                           ; restoring the registers and putting the return address in
                           ; the process counter
```

3.6.2 fork

TODO ...

Em um sistema operacional, a system call `fork` é responsável pela criação de novos processos. Para tal, ela duplica o processo que a criou, onde o único meio de se identificar qual o processo pai é pelo número de retorno. Caso o número de retorno seja 0, significa que este é o processo filho, e caso seja qualquer outro número, é o processo pai que retornou o identificador do processo filho.

Nosso `fork` teve de ser ligeiramente alterado por causa de uma simplificação que fizemos em nosso kernel. Como dito anteriormente, temos uma área de dados única para todos os processos. Com isso, fica impossível de se duplicar a área de dados de um processo, o que não fazemos.

O processo de duplicação de um processo se inicia com o empilhamento dos registradores

de dados (r0 a r12) e do endereço de retorno (link register) por duas vezes. O motivo é que o primeiro empilhamento serve para a restauração do estado ao fim do processo de duplicação e a segunda para o processo que vai a ser duplicado. Então, tentamos encontrar qual o primeiro espaço disponível dentro da tabela de processos. Uma vez encontrado o espaço, temos de encontrar o espaço do PCB reservado para este processo, onde iremos popular com os dados do estado em execução. Porém, além disso, temos também de duplicar a pilha, que é um processo um pouco mais complexo. Para tal, primeiro temos de descobrir o tamanho da pilha atual. Então, começamos a copiar os dados de uma pilha para a outra. Finalmente, colocamos no ponteiro de pilha do PCB do novo processo o topo da pilha recém copiada. Uma vez resolvido o problema da cópia de pilha, apenas duplicamos os dados do registrador de estados, do link register, do process counter e de todos os registradores de dados. Finalmente, quando o processo está totalmente copiado, devemos habilitar o processo na tabela de processos e restaurar todos os registradores empilhados de volta ao seus lugares, onde o link register entrará no lugar do process counter.

3.6.3 exec

A chamada de sistema *exec* é responsável por substituir a imagem núcleo de um processo pela imagem do programa passado como argumento (TANENBAUM; WOODHULL, 2006).

Nos sistemas operacionais tradicionais, como o Linux ou o Minix, o *exec* é utilizado para iniciar um novo programa no mesmo ambiente do programa que executa a chamada de sistema. Normalmente o *exec* é utilizado na criação de um novo processo da seguinte maneira: um processo já existente se duplica através da chamada de sistema *fork*. O processo filho tem, então, seu código substituído pelo código que deve ser executado através da chamada de sistema *exec*, que permite ao processo filho assumir seu próprio conteúdo, apagando de si o conteúdo do processo pai.

No KinOS, para que um *thread* passe a executar outro programa, é necessário reinicializar o seu PCB, isso é feito pela chamada de sistema *exec*.

Existem 4 principais entradas do PCB que necessitam ser reinicializadas:

- o *program counter* (PC - R13);
- o *link register* (LR - R14);
- o *stack pointer* (SP - R15);
- e o *saved processor status register* (SPSR).

Para reinicializar essas entradas, de forma que a *thread* passe à executar um novo programa, primeiro é necessário calcular o início do PCB da *thread* correspondente.

A rotina *exec*, recebe como parâmetros o id da *thread* que será alterada e o ponteiro para a função/programa que pretende-se executar, como mostrado a seguir:

```
void exec(int process_id, pt2Task process_addr);
```

Assim para calcular o endereço inicial do PCB, obtêm-se o endereço inicial da área reservada para armazenar todos os PCBs, a **process_control_block**, e adiciona-se à esta o valor de 68 multiplicado por **process_id**, visto que cada PCB ocupa um espaço de 68 endereços de memória como mencionado na sessão 3.2.1. O código responsável por calcular o PCB é apresentado a baixo:

```
LDR r3, =process_control_block ; r3 = the start address of the PCB area
MOV r4, #68 ; r4 = 68 (space for each process in the PCB)
MUL r5, r1, r4 ; r5 = (task id) * 68
ADD r3, r3, r5 ; r3 = PCB start address + r5
```

Em seguida, calculado o endereço inicial do PCB, altera-se suas entradas da seguinte maneira:

- LR (PCB[-4]) e PC (PCB[-64]) recebem o endereço da primeira instrução do novo programa (**process_addr**).

```
PCB[-4] = process_addr;
PCB[-64] = process_addr;
```

- SP (PCB[-8]) recebe o endereço de início da pilha da *thread*, fazendo com que esta seja zerada. Para cada pilha de *thread*, 4048 bytes são reservados.

```
PCB[-8] = início da pilha do modo usuário - (4048 * thread id);
```

- SPSR (PCB[-68]) recebe 0x10, pois os programas devem rodar no modo usuário.

```
PCB[-68] = 0x10;
```

Finalmente, após alterar as entradas mostradas a cima, a *thread* começa a executar o novo programa.

3.6.4 exit

A chamada de sistema *exit* é responsável por finalizar um processo, liberando espaço de memória para a execução de um novo processo (TANENBAUM; WOODHULL, 2006).

No KinOS isso é realizado apenas colocando como desativado (igual à 0) o byte na lista de processos que corresponde a *thread* que se deseja finalizar.

Para isso a rotina *exit* recebe como parâmetro o id da *thread* a ser terminada.

```
void exit(int process_id);
```

3.7 Shell

Com o desenvolvimento do microkernel e de suas system calls, torna-se necessário o desenvolvimento de outro ramo do projeto, destinado a permitir a interação do usuário com o Sistema Operacional. Essa interação é feita por um editor de linha de comando, também conhecida por Shell.

Na inicialização do microkernel, o Shell é o primeiro processo criado no sistema. Desse momento em diante, cabe ao usuário solicitar a execução ou o término de outros processos. Além disso, o Shell permite a visualização dos diferentes processos em execução no sistema.

3.7.1 Comunicação via terminal

O Shell, para fazer a interação com o usuário, utiliza a porta serial COM0 (de uso geral) conectada a uma segunda porta serial da máquina host. A porta COM1 (Debug) deve permanecer conectada, pois o Angel mantém comunicações através dela com o AXD (descrito na seção 2.3.2) durante a execução do KinOS, como ilustrado na figura 3.8.

3.8 Mutex

TODO...

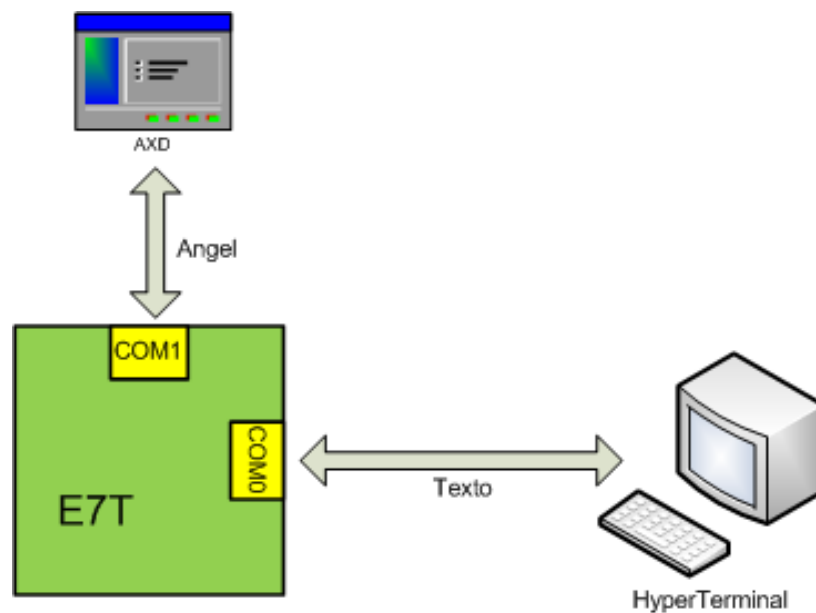


Figura 3.8: Comunicação da Evaluator-7T em cada porta serial.

3.9 Processos

O kernel pode lidar com no máximo nove processos, nomeados de task1 a task9 no arquivo tasks.c. Como eles não têm área de dados própria, não pode-se chamá-los de processos. A implicação de se ter uma área de dados em comum é que todos os processos que rodam um mesmo programa compartilham os valores das variáveis. O mais correto, portanto, seria o chamá-los de threads.

Criamos alguns programas exemplo que se utilizam dos periféricos da placa.

TODO... (Fazer código antes)

3.10 Inspiração

Grande parte do código foi retirada do código presente nos exemplos incluídos no CD de demonstração da placa. O principal deles, é o código *mutex*, desenvolvido por Andrew N. Sloss, de onde foi baseado o chaveamento de processos, a função de semáforo e as rotinas de manipulação de hardware.

TODO ...

4 CONSIDERAÇÕES FINAIS

4.1 Conclusão

Através deste projeto de formatura foi possível desenvolver, como previsto, um *microkernel* simples para a placa ARM Evaluator-7T.

O KinOS é um *microkernel* que implementa, de maneira didática, os mecanismos básicos de um sistema operacional, como: chaveamento de threads, chamadas de sistema e rotinas de comunicação com os periféricos da placa (*leds, display, botão, switches*). Além disso, o KinOS possui um simples terminal para a interação do usuário com o sistema e de onde é possível executar os programas adicionados ao *microkernel*.

4.2 Contribuições

Pretende-se que a monografia e a parte prática desenvolvida durante o projeto, sejam utilizados como material didático para o Laboratório de Microprocessadores, fazendo assim com que este se torne mais atual e mais próximo da disciplina de Sistemas Operacionais.

Quanto à contribuição para os integrantes do grupo, este estudo possibilitou principalmente o aprendizado da arquitetura de processadores ARM, além da consolidação dos aspectos teóricos aprendidos em Sistemas Operacionais, através da implementação do *microkernel*.

4.3 Trabalhos Futuros

REFERÊNCIAS BIBLIOGRÁFICAS

ABDELRAZEK, A. F. M. *Exception and Interrupt Handling in ARM*. [S.l.], Setembro 2006. Disponível em: <http://www.iti.uni-stuttgart.de/radetzki/Seminar06/08_report.pdf>.

ARM LIMITED. *ARM7TDMI Data Sheet (ARM DDI 0029E)*. [S.l.], Agosto 1995. Disponível em: <http://www.eecs.umich.edu/panalyzer/pdfs/ARM_doc.pdf>.

ARM LIMITED. *Application Note 25 - Exception Handling on the ARM (ARM DAI 0025E)*. [S.l.], Setembro 1996. Disponível em: <<http://www.imit.kth.se/courses/2B1445/0304/material/Apps25vE.pdf>>.

ARM LIMITED. *ARM Evaluator-7T Board User Guide*. [S.l.], Agosto 2000. Disponível em: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0134a/DUI0134A_evaluator7t_ug.pdf>.

ARM LIMITED. *ARM Developer Suite AXD and armsd Debuggers Guide*. [S.l.], Novembro 2001. Disponível em: <<http://infocenter.arm.com/help/topic/com.arm.doc.dui0066d/DUI0066.pdf>>.

ARM LIMITED. *ARM7TDMI Technical Reference Manual (ARM DDI 0029G)*. Rev 3. [S.l.], Abril 2001. Disponível em: <<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0029g/index.html>>.

ARM LIMITED. *ARM Architecture Reference Manual (ARM DDI 0100I)*. [S.l.], Julho 2005. Disponível em: <<http://www.arm.com/miscPDFs/14128.pdf>>.

FURBER, S. *ARM System-On-Chip Architecture*. 2. ed. [S.l.]: Addison-Wesley, 2000. ISBN 0-20167-519-6.

KINOSHITA, C. C. e. A. H. J. *Experiência 5: Interrupções*. [S.l.], 2007. Disponível em: <<http://www.pcs.usp.br/jkinoshi/2007/tomas5.doc>>.

MORROW, M. G. *ARM7TDMI Instruction Set Reference*. [S.l.], Setembro 2008. Disponível em: <http://eceserv0.ece.wisc.edu/morrow/ECE353/arm7tdmi_instruction_set_reference.pdf>.

RYZHYK, L. *The arm architecture*. Junho 2006. Disponível em: <<http://www.cse.unsw.edu.au/cs9244/06/seminars/08-leonidr.pdf>>.

SAMSUNG ELECTRONICS. *KS32C50100 RISC MicroController User Manual*. [S.l.], Agosto 2007. Disponível em: <<http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/SystemLSI/Net>>.

SLOSS, A. *Interrupt Handling*. [S.l.], Abril 2001.

SLOSS, A.; SYMES, D.; WRIGHT, C. *ARM System Developer's Guide: designing and optimizing system software*. 1. ed. [S.l.]: Morgan Kaufman, 2004. ISBN 1-55860-874-5.

TANENBAUM, A. S.; WOODHULL, A. S. *Operating Systems: Design and Implementation*. 3rd. ed. [S.l.]: Pearson Prentice Hall, 2006. ISBN 0-13-142938-8.

ZAITSEFF, J. *ELEC2041 Microprocessors - Laboratory Manual*. [S.l.], Junho 2003. Disponível em: <<http://www.zap.org.au/elec2041-cdrom/unsw/elec2041/README.html>>.

Apêndice A – ARQUIVOS FONTE

A.1 cinit.h

```
1 #include "constants.h"
2 #include "apps/tasks.h"
3 #include "apps/terminal.h"
4 #include "interrupt/irq.h"
5 #include "peripherals/button.h"
6 #include "peripherals/segment.h"
7 #include "peripherals/timer.h"
8
9 extern void handler_board_angel(void);
10 extern void handler_board_no_angel(void);
11 extern void handler_swi(void);
12 extern void handler_emulator(void);
13 extern void task1(void);
```

A.2 cinit.c

```
1 /*
2  KinOS – Microkernel for ARM Evaluator 7–T
3  Seniors project – Computer Engineering
4  Escola Politecnica da USP, 2009
5
6  Felipe Giunte Yoshida
7  Mariana Ramos Franco
8  Vinicius Tosta Ribeiro
9  */
10
11 /*
12  The program was based on the mutex program by ARM – Strategic Support Group
13  ,
```

```

13 contained on the ARM Evaluator 7-T example CD, under the folder /Evaluator7
    -T/
14 source/examples/mutex/
15 */
16
17 /* Initialization code in C */
18
19 #include "cinit.h"
20
21 /* Entry point for C part */
22 int C_Entry (void) {
23     /* Initialize 7-segment display */
24     segment_init();
25     /* Initialize timer */
26     timer_init();
27     /* Initialize button */
28     button_init();
29     /* Install hardware interruption handler */
30     if (emulator == 1) {
31         install_handler ((unsigned)handler_emulator, (unsigned *)IRQVector);
32     }
33     else if (emulator == 0) {
34         install_handler ((unsigned)handler_board_angel, (unsigned *)IRQVector);
35     }
36     else {
37         install_handler ((unsigned)handler_board_no_angel, (unsigned *)
38             IRQVector);
39     }
40     /* Install software interruption handler */
41     install_handler ((unsigned)handler_swi, (unsigned *)SWIVector);
42     /* Start timer */
43     timer_start();
44     /* Enabling IRQ interruption and changing to user mode */
45     __asm {
46         MOV    r1, #0x40|0x10
47         MSR    CPSR_c, r1
48     }
49     /* Start with process 1 */
50     task1();
51     //shell();
52     /* The return below should not be reachable */
53     return 0;
54 }

```


A.3 constants.h

```

1  /****** GENERAL VARIABLES *****/
2  /* Defines if the program is running on: */
3  /* 0 – Evaluator 7–T board with Angel */
4  /* 1 – CodeWarrior ARMULATOR */
5  /* 2 – Evaluator 7–T board no Angel */
6  #define emulator 0
7  /* The number of the operating system software interrupt */
8  #define OS_SWI 0
9  /* Interrupt table SWI instruction position */
10 #define SWIVector (unsigned *) 0x08
11 /* Interrupt table IRQ instruction position */
12 #define IRQVector (unsigned *) 0x18
13 /* Time set for the timer */
14 // #define COUNTDOWN 0x00effff0
15 #define COUNTDOWN 0x000ffff0
16
17 /****** EMULATOR VARIABLES *****/
18 /* Timer interrupt ID */
19 #define IRQTimer 0x0010
20 /* IRQ interrupt controller addresses */
21 #define IRQEnableSet (volatile unsigned *) 0x0A000008
22 #define IRQEnableClear (volatile unsigned *) 0x0A00000C
23 /* Timer registers */
24 #define EmulatorIRQTimerLoad (volatile unsigned *) 0x0A800000
25 #define EmulatorIRQTimerControl (volatile unsigned *) 0x0A800008
26 #define IRQTimerClear (volatile unsigned *) 0x0A80000C
27
28 /****** BOARD VARIABLES *****/
29 /* Inoput/output data address */
30 #define IOData (volatile unsigned *) 0x03ff5008
31 /* IRQ interrupt controller addresses */
32 #define IRQStatus (volatile unsigned *) 0x03ff4004
33 /* Timer registers */
34 #define TimerEnableSet (volatile unsigned *) 0x03ff6000
35 #define EvaluatorIRQTimerLoad (volatile unsigned *) 0x03ff6004
36 #define EvaluatorIRQTimerControl (volatile unsigned *) 0x03ff4008
37 /* Button addresses */
38 #define IRQButtonControl (volatile unsigned *) 0x03ff5004

```

```

39  /* Segment addresses */
40  #define IOPMod                (volatile unsigned *)0x03ff5000
41  /* The bits taken up by the display in IOData register */
42  #define Segment_mask  0x1FC00
43  /* Define segments in terms of IO lines */
44  #define SEG_A    (1<<10)
45  #define SEG_B    (1<<11)
46  #define SEG_C    (1<<12)
47  #define SEG_D    (1<<13)
48  #define SEG_E    (1<<14)
49  #define SEG_F    (1<<16)
50  #define SEG_G    (1<<15)
51  #define DISP_0    (SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F)
52  #define DISP_1    (SEG_B|SEG_C)
53  #define DISP_2    (SEG_A|SEG_B|SEG_D|SEG_E|SEG_G)
54  #define DISP_3    (SEG_A|SEG_B|SEG_C|SEG_D|SEG_G)
55  #define DISP_4    (SEG_B|SEG_C|SEG_F|SEG_G)
56  #define DISP_5    (SEG_A|SEG_C|SEG_D|SEG_F|SEG_G)
57  #define DISP_6    (SEG_A|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
58  #define DISP_7    (SEG_A|SEG_B|SEG_C)
59  #define DISP_8    (SEG_A|SEG_B|SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
60  #define DISP_9    (SEG_A|SEG_B|SEG_C|SEG_D|SEG_F|SEG_G)
61  #define DISP_A    (SEG_A|SEG_B|SEG_C|SEG_E|SEG_F|SEG_G)
62  #define DISP_B    (SEG_C|SEG_D|SEG_E|SEG_F|SEG_G)
63  #define DISP_C    (SEG_A|SEG_D|SEG_E|SEG_F)
64  #define DISP_D    (SEG_B|SEG_C|SEG_D|SEG_E|SEG_G)
65  #define DISP_E    (SEG_A|SEG_D|SEG_E|SEG_F|SEG_G)
66  #define DISP_F    (SEG_A|SEG_E|SEG_F|SEG_G)

```

A.4 startup.s

```

1  ; Startup assembly code
2  ; Obs.: This code was built supposing that the generated assembly is ARM
3  ; assembly, not THUMB!!!
4
5  IMPORT  current_thread_id
6  IMPORT  thread_array
7  IMPORT  C_Entry
8
9  ; Identifying that from below on it is assembly code (readable only)
10 AREA asm_init, CODE

```

```

11
12 ; Entry point of the program
13 ENTRY
14
15 ; Beginning assembly initialization
16 start
17 ; Changing to IRQ mode and disabling interruptions , then setting up
18 ; IRQ stack pointer to 0x8000
19 MOV    r0 , #0xC0|0x12    ; r0 = 0xC0 or 0x12 (0xC0 = IRQ disabled ,
20                      ; 0x12 = IRQ mode)
21 MSR    CPSR_c, r0        ; status_register = r0
22 MOV    sp , #0x8000      ; stack pointer = 0x8000
23
24 ; Changing to system mode and disabling interruptions , then setting up
25 ; user stack pointer to 0x20000
26 MOV    r0 , #0xC0|0x1F    ; r0 = 0xC0 or 0x1F (0xC0 = IRQ disabled ,
27                      ; 0x1F = system mode)
28 MSR    CPSR_c, r0        ; status_register = r0
29 MOV    sp , #0x20000      ; stack pointer = 0x20000
30
31 ; Changing to SVC mode and disabling interruptions , then setting up
32 ; SVC stack pointer to 0x8000 - 128
33 MOV    r0 , #0xC0|0x13    ; r0 = 0xC0 or 0x13 (0xC0 = IRQ disabled ,
34                      ; 0x13 = SVC mode)
35 MSR    CPSR_c, r0        ; status_register = r0
36 MOV    r0 , #0x8000      ; r0 = 0x8000
37 SUB    r0 , r0 , #128     ; r0 = r0 - 128
38 MOV    sp , r0           ; stack pointer = r0
39
40 ; Initializes the thread array with zeros (0 = thread disabled ,
41 ; 1 = thread enabled)
42 LDR    r0 , =thread_array ; r0 = thread_array start address
43 MOV    r1 , #1           ; r1 = 1
44 STR    r1 , [r0]         ; address(r0) = r1
45 MOV    r1 , #0           ; r1 = 0 (disabled)
46 MOV    r2 , #0           ; r2 = 0
47 init_thread_array_loop
48 ADD    r2 , r2 , #4       ; r2 = r2 + 4
49 CMP    r2 , #36          ; r2 = 36?
50 BEQ    set_active_thread ; if yes , go to set_active_thread
51 ADD    r3 , r0 , r2       ; r3 = r0 + r2
52 STR    r1 , [r3]         ; address(r3) = r1
53 B      init_thread_array_loop ; return to init_thread_array_2

```

```

54
55     ; Setting the thread id to 1
56 set_active_thread
57     LDR    r0, =current_thread_id ; r0 = current thread id address
58     MOV    r1, #1                ; r1 = 1
59     STR    r1, [r0]              ; current thread id = 1
60
61     ; Pass control to C_Entry
62     LDR    lr, =C_Entry           ; link register = C entry
63     MOV    pc, lr                 ; process counter = C entry
64
65     ; End of assembly code
66     END

```

A.5 apps/tasks.h

```

1  #include "../peripherals/segment.h"
2  #include "../interrupt/swi.h"
3
4
5  void task1 (void);
6  void task2 (int);
7  void set_segment(int);

```

A.6 apps/tasks.c

```

1  #include "tasks.h"
2
3
4  void task1 (void) {
5
6      int a = 0;
7      //char* newTask = "set_segment";
8
9      int j;
10
11     a = fork();
12     if(a != -1 && a != 0){
13         // exec(a, get_task_addr(newTask), 2);
14     }

```

```
15
16     while (1) {
17         segment_set(1);
18         if(j==1000000){
19             exit(a);
20         }
21         j++;
22     }
23 }
24
25 void task2 (int value) {
26     while (1) {
27         segment_set(2);
28     }
29 }
30
31
32 void set_segment(int value){
33     while (1) {
34         segment_set(value);
35     }
36 }
37
38
39
40
41 void task3 (void) {
42     while (1) {
43         segment_set(3);
44     }
45 }
46
47 void task4 (void) {
48     while (1) {
49         segment_set(4);
50     }
51 }
52
53 void task5 (void) {
54     while (1) {
55         segment_set(5);
56     }
57 }
```

```

58
59 void task6 (void) {
60     while (1) {
61         segment_set(6);
62     }
63 }
64
65 void task7 (void) {
66     while (1) {
67         segment_set(7);
68     }
69 }
70
71 void task8 (void) {
72     while (1) {
73         segment_set(8);
74     }
75 }
76
77 void task9 (void) {
78     while (1) {
79         segment_set(9);
80     }
81 }

```

A.7 apps/terminal.h

```

1 int strcmp(char* str1, char* str2);
2
3 void shell (void);

```

A.8 apps/terminal.c

```

1 /******
2  *
3  *  ARM Strategic Support Group
4  *
5  *****/
6
7 /******

```

```

8  *
9  *  Module      : comm.c
10 *  Description  : communicates with the serial driver.
11 *  Tool Chain  : ARM Developer Suite 1.0
12 *  Platform    : Evaluator7T
13 *  History     :
14 *
15 *              2000-03-22 Andrew N. Sloss
16 *              - implemented
17 *
18 *
19 *****/
20
21 /*****
22  *  IMPORT
23  *****/
24
25 #include "serial.h"
26 #include "tasks.h"
27
28 #include <string.h>
29
30 /*****
31  *  MACROS
32  *****/
33
34 #define angel_SWI 0x123456
35 #define CMD_LENGTH 32
36
37
38 struct { char* name; void (*task_ptr)(int); } tasks_name[] = {
39     {"task2", &task2},
40     {"set_segment", &set_segment}
41 };
42
43
44 /*****
45  *  MISC
46  *****/
47
48 __swi (angel_SWI) void _Exit(unsigned op, unsigned except);
49 #define Exit() _Exit(0x18,0x20026)
50

```

```

51 __swi (angel_SWI) void _WriteC(unsigned op, const char *c);
52 #define WriteC(c) _WriteC (0x3,c)
53
54
55 /******
56  * ROUTINES
57  *****/
58
59
60 /*
61 int strcmp (char* str1, char* str2){
62     int i;
63     for (i = 0; str1[i] == str2[i]; i++){
64         if (str1[i] == '\0'){
65             return 0;
66         }
67     }
68     return str1[i] - str2[i];
69 }
70 */
71
72
73 pt2Task get_task_addr(char* name){
74     int i;
75     for(i=0; i<sizeof(tasks_name); i++){
76         if(strcmp(tasks_name[i].name, name)==0){
77             return tasks_name[i].task_ptr;
78         }
79     }
80     return 0;
81 }
82
83
84
85 /* -- comm_print -----
86  *
87  * Description : write a string via the Angel SWI call WriteC
88  *
89  * Parameters : const char *string - string to be written
90  * Return      : none...
91  * Notes       : none...
92  *
93  */

```



```

94
95 void comm_print (const char *string)
96 {
97     int pos = 0;
98     while (string[pos] != 0) WriteC(&string[pos++]);
99 }
100
101 /* -- comm_init -----
102 *
103 * Description : initialize the COM0 port and set to 9600 baud.
104 *
105 * Parameters : none...
106 * Return     : none...
107 * Notes      : none...
108 *
109 */
110
111 void comm_init (void)
112 {
113     serial_initcom0user (BAUD_9600);
114 }
115
116 /* -- comm_banner -----
117 *
118 * Description : print out standard banner out of the COM0 port
119 *
120 * Parameters : none...
121 * Return     : none...
122 * Notes      : none...
123 *
124 */
125
126 void comm_banner (void)
127 {
128     serial_print (COM0_USER, "\n**_Welcome_to_KinOS!!");
129     serial_print (COM0_USER, "_ _Version_0.1_**\n\r");
130 }
131
132 /* -- comm_getkey -----
133 *
134 * Description : wait until a key is press from the host PC.
135 *
136 * Parameters : none...

```

```

137  * Return      : none...
138  * Notes      : none...
139  *
140  */
141
142  void comm_getkey (void)
143  {
144      serial_getkey();
145  }
146
147
148  char comm_getcmd (void)
149  {
150      return serial_getcmd();
151  }
152
153
154  /* — C_Entry —————
155  *
156  * Description  : Entry point into C
157  *
158  * Parameters  : none...
159  * Return      : none...
160  * Notes      : none...
161  *
162  */
163
164  void shell (void)
165  {
166      char cmd[CMD.LENGTH];
167      char c;
168      int i;
169
170      //int a = 0;
171      //char* newTask = "set_segment";
172
173      comm_print ( "\n—_switch_to_a_serial_terminal_program_(baud=9600)\n" );
174
175      comm_init();
176      comm_banner();
177
178      // wait for a key press...
179

```

```

180     serial_print (COM0_USER, "—_Press_any_key_\n\r");
181
182
183     c=0;
184     i=0;
185     while (c != '\r' && i<CMD_LENGTH) {
186         c = comm_getcmd();
187         if (c == 8) { // backspace
188             if (i>0) i--;
189         } else {
190             cmd[i++] = c;
191         }
192     }
193
194
195
196     //a = fork();
197     //if(a != -1 && a != 0){
198     //    exec(a ,get_task_addr(newTask), 1);
199     //}
200
201     //while(1){
202         segment_set(1);
203     //}
204
205     comm_getkey();
206     serial_print (COM0_USER, "\n\r—_Key_pressed_\n\r");
207     serial_print (COM0_USER, "\n\r**_Program_Terminating_**\n\r");
208
209     comm_print (" \n\n_**_program_terminating_normally_**");
210
211     Exit();
212 }
213
214
215 /******
216  * END OF comm.c
217  *****/

```

A.9 interrupt/handler_irq.s

```

1 ; Hardware interrupt handling code
2
3 IMPORT  button_irq
4 IMPORT  timer_irq
5
6 EXPORT  Angel_IRQ_Address
7 EXPORT  current_thread_id
8 EXPORT  handler_board_angel
9 EXPORT  handler_board_no_angel
10 EXPORT  handler_emulator
11 EXPORT  process_control_block
12 EXPORT  thread_array
13
14 ; Beginning handler code
15 AREA  handler_irq, CODE
16
17 ; Routine designed to the emulator, all the hardware IRQ is caused by the
    timer
18 handler_emulator
19     STMFD sp!, {r0 - r3, lr} ; Stacking r0 to r3 and the link register
20     B    handler_timer ; Branch to handler_timer
21
22 ; Routine designed to the board, have the Angel handler routine, button and
    timer
23 handler_board_angel
24     ; Save current context for APCS
25     STMFD sp!, {r0 - r3, lr} ; Stacking r0 to r3 and the link register
26     LDR  r0, IRQStatus ; r0 = irq type address
27     LDR  r0, [r0] ; r0 = irq type
28     TST  r0, #0x0400 ; irq type == 0x0400?
29     BNE  handler_timer ; If yes, go to handler_timer
30     TST  r0, #0x0001 ; irq type = 0x0001?
31     BNE  handler_button ; If yes, go to handler_button
32     LDMFD sp!, {r0 - r3, lr} ; If it is not any of them, restore r0-r3 and
        lr
33     LDR  pc, Angel_IRQ_Address ; and branch to the Angel routine
34
35 ; Routine designed to the board, not have the Angel handler routine, button
    and timer
36 handler_board_no_angel
37     ; Save current context for APCS
38     STMFD sp!, {r0 - r3, lr} ; Stacking r0 to r3 and the link register
39     LDR  r0, IRQStatus ; r0 = irq type address

```

```

40     LDR    r0, [r0]          ; r0 = irq type
41     TST    r0, #0x0400      ; irq type == 0x0400?
42     BNE    handler_timer    ; If yes, go to handler_timer
43     TST    r0, #0x0001      ; irq type = 0x0001?
44     BNE    handler_button    ; If yes, go to handler_button
45     LDMFD  sp!, {r0 - r3, lr} ; If it is not any of them, restore r0-r3
                                and lr
46     B      return          ; and return
47
48 ; handler routine for the button interruption
49 handler_button
50     BL     button_irq        ; C routine for the button
51     B      no_thread_switch  ; End the handler
52
53 ; Timer interruption handler routine
54 handler_timer
55     STMFD  sp!, {r4 - r12}    ; Stack the rest of the registers (r4-r12)
56     BL     timer_irq          ; Clear timer interruption
57     LDMFD  sp!, {r4 - r12}    ; Load r4-12 registers again
58     LDR    r0, =current_thread_id ; r0 = current_thread_id address
59     LDR    r0, [r0]          ; r0 = current_thread_id
60                                ; Send to the next step r0 as the current
61                                ; thread ID
62
63 ; Finds out the next active thread id (send result in r1)
64 get_next_taskid_loop
65     CMP    r0, #9            ; r0 == 9? (it is the last thread?)
66     BEQ    last_thread        ; If yes, branch last_thread
67     ADD    r1, r0, #1         ; If not, r1 = r0 + 1
68     B      next_thread        ; and branch to next_thread
69 last_thread
70     MOV    r1, #1            ; r1 = 1
71 next_thread
72     SUB    r2, r1, #1         ; r2 = r1 - 1
73     MOV    r3, #4            ; r3 = 4
74     MUL    r2, r3, r2         ; r2 = r2 * r3
75     LDR    r3, =thread_array  ; r3 = thread_array bottom address
76     ADD    r2, r2, r3         ; r2 = r3 + r2
77     LDR    r2, [r2]          ; r2 = thread array content
78     CMP    r2, #1            ; thread array content = 1?
79     BEQ    set_addresses      ; If yes, branch to set_addresses
80                                ; Send to the next step the next active
81                                ; thread in r1

```

```

82  MOV    r0, r1          ; If not, r0 = r1
83  B      get_next_taskid_loop ; and loop to get_next_taskid_loop
84
85  ; Sets current and next thread PCB addresses
86  set_addresses
87  LDR    r2, =current_thread_id ; r2 = current thread id address
88  LDR    r2, [r2]             ; r2 = current thread id
89  CMP    r2, r1              ; Is r2 = current thread id ==
90                               ; next thread id
91  BEQ    no_thread_switch    ; If yes, branch to no_thread_switch
92  ; Setting current_task_addr
93  MOV    r0, #68             ; Else start thread switch. r0 = 68
94  MUL    r0, r2, r0          ; r0 = current thread id * 68
95  LDR    r2, =process_control_block ; r2 = PCB bottom
96  ADD    r0, r0, r2          ; r0 = PCB bottom + id * 68
97  LDR    r2, =current_task_addr ; r2 = current task addr addr
98  STR    r0, [r2]            ; current_task_addr = r0
99  ; Setting next_task_addr
100  MOV    r0, #68             ; r0 = 68
101  MUL    r0, r1, r0          ; r0 = next thread id * 68
102  LDR    r2, =process_control_block ; r2 = PCB_bottom
103  ADD    r0, r2, r0          ; r0 = PCB bottom + next id * 68
104  LDR    r2, =next_task_addr  ; r2 = next_task_addr addr
105  STR    r0, [r2]            ; next_task_addr = r0
106
107  ; Setting new current_thread_id
108  LDR    r0, =current_thread_id ; r0 = current_thread_id
109  STR    r1, [r0]            ; current_thread_id = next thread id
110
111  ; Carry out process switch
112  ; Reset and save IRQ stack
113  LDR    r0, =irq_stack_pointer ; r0 = irq_stack_pointer addr
114  MOV    r1, sp              ; r1 = irq stack pointer
115  ADD    r1, r1, #5*4        ; r1 = irq stack pointer + 5 (# of data in
116                               ; the stack, r0-r3, lr) * 4 (size of a word)
117  STR    r1, [r0]            ; irq_stack_pointer = irq stack pointer
118                               ; without the data that will be removed next
119  LDMFD  sp!, {r0-r3, lr}    ; Restore the remaining registers
120  ; Load and position r13 to point into current PCB
121  LDR    r13, =current_task_addr ; r13 = current task PCB bottom address
122                               address
123  LDR    r13, [r13]          ; r13 = current task PCB bottom address
124  SUB    r13, r13, #60       ; r13 = current task PCB bottom address - 60

```

```

124             ; to point to the right place for the stacking
125             ; (next step)
126 ; Store the current user registers in current PCB
127 STMIA r13, {r0-r14}^ ; Stacks the r0-r14 registers in the PCB
128 MRS r0, SPSR ; r0 = status register
129 STMDB r13, {r0,r14} ; Stacks r0 and r14
130 ; Load and position r13 to point into next PCB
131 LDR r13, =next_task_addr ; r13 = next task PCB bottom address
132 address
133 LDR r13, [r13] ; r13 = next task PCB bottom address
134 SUB r13, r13, #60 ; r13 = next task PCB bottom address - 60
135             ; to point to the right place for the stacking
136             ; (next step)
137 ; Load the next task and setup PSR
138 LDMNEDB r13, {r0,r14} ; Restore r0 and r14 (IRQ mode)
139 MSRNE spsr_cxsf, r0 ; Restore status register
140 LDMNEIA r13, {r0-r14}^ ; Restore r0-r14 for the user mode
141 NOP ; NOP! (required for the above instruction)
142 ; Load the IRQ stack into r13_irq
143 LDR r13, =irq_stack_pointer ; r13 = stack pointer address address
144 LDR r13, [r13] ; Restore previous stack pointer
145 B return ; Go to the end
146
147 no_thread_switch
148 LDMFD sp!, {r0-r3, lr} ; Restore the remaining registers
149
150 return
151 SUBS pc, r14, #4 ; Process counter = IRQ mode link register - 4
152 ; (-4 is required for the pipeline)
153
154 ; Data area
155 AREA irq_vars, DATA
156
157 IRQStatus ; IRQ interrupt type address
158 DCD 0x03ff4004
159
160 Angel_IRQ_Address ; Reserved space for the Angel IRQ Interrupt address
161 DCD 0x00000000
162
163 current_thread_id ; Context task ID
164 DCD 0x0
165
166 current_task_addr ; Address of the PCB for the current Task
167 DCD 0x0
168
169 next_task_addr ; Address of the PCB for the next Task
170 DCD 0x0

```

```

irq_stack_pointer    ; Copy of the IRQ stack
    DCD 0x0
process_control_block ; PCB for all the tasks (each size = 68) Offsets =
    bottom +
    % 612          ; 68 * process#
thread_array         ; Thread status array, where each thread has one word to
    indicate
    % 36          ; if it is active (1) or inactive (0). Offset = bottom + 4 *
        thread#

; End of assembly code
END

```

A.10 interrupt/handler_swi.s

```

1 ; Software interrupt handling code
2
3 IMPORT  routine_fork
4 IMPORT  routine_exec
5 IMPORT  routine_exit
6
7 EXPORT  Angel_SWI_Address
8 EXPORT  handler_swi
9
10 ; Beginning handler code
11 AREA  handler, CODE
12
13 ; Software interrupt routine handler
14 handler_swi
15     STMFD    sp!,{r0-r12,lr}      ; Stack registers r0-12 and link register
16     LDR      r0,[lr,#-4]           ; Calculate address of SWI instruction (r0 = lr
                                   -4)
17     BIC      r0,r0,#0xff000000    ; Mask off top 8 bits of instruction to give
                                   SWI
18                                   ; number
19     LDR      r1, Angel_SWI_Number ; r1 = Angel SWI Number
20     CMP      r0, r1               ; Compare SWI number to angel interrupt number
21     BEQ      goto_angel           ; If it is angel interrupt, branch to goto_angel
22     MOV      r1, #0               ; r1 = 0
23     CMP      r0, r1               ; Compare SWI number to r1
24     BEQ      os_swi               ; If it is OS SWI, branch to os_swi

```



```

25
26 ; Go to Angel routine
27 goto_angel
28   LDMFD sp!,{r0-r12,lr}      ; Restore registers r0-r12 and link register
29   LDR   pc, Angel_SWI_Address ; Branch to the Angel
30
31 ; Operating system SWI handler, identify the routine
32 os_swi
33   LDMFD sp!,{r0-r12,lr}      ; Restore r0-r12 registers and link registers
34   STMFD  sp!,{r0-r12,lr}      ; and stores them again (in order to clean the
      registers)
35   MOV    r1, #0              ; r1 = 0
36   CMP    r0, r1              ; Compare the first parameter to 0
37   BEQ    pre_routine_fork    ; If it is equal, branch to the fork
38   MOV    r1, #1              ; r1 = 1
39   CMP    r0, r1              ; Compare the first parameter to 1
40   BEQ    pre_routine_exec    ; If it is equal, branch to the exec
41   MOV    r1, #2              ; r1 = 2
42   CMP    r0, r1              ; Compare the first parameter to 2
43   BEQ    pre_routine_exit    ; If it is equal, branch to the exit
44   LDMFD  sp!,{r0-r12,pc}^     ; If it is an unidentified syscall, go back to
      the program,
45           ; restoring the registers and putting the return address in
46           ; the process counter
47
48 ; Fork caller
49 pre_routine_fork
50   LDMFD  sp!,{r0-r12,lr}      ; Restore r0-r12 registers and link registers
51   STMFD  sp!,{r0-r12,lr}      ; and stores them again (in order to clean the
      registers)
52   B      routine_fork         ; Branch to the fork C routine
53
54 ; Exec caller
55 pre_routine_exec
56   LDMFD  sp!,{r0-r12,lr}      ; Restore r0-r12 registers and link registers
57   STMFD  sp!,{r0-r12,lr}      ; and stores them again (in order to clean the
      registers)
58   B      routine_exec         ; Branch to the exec C routine
59
60 ; Exit caller
61 pre_routine_exit
62   LDMFD  sp!,{r0-r12,lr}      ; Restore r0-r12 registers and link registers

```

```

63   STMFD    sp!,{r0-r12,lr} ; and stores them again (in order to clean the
        registers)
64   B routine_exit    ; Branch to the exit C routine
65
66   ; Data area
67   AREA    swi_vars , DATA
68
69   Angel_SWI_Number    ; Identification number for the Angel SWI
70       DCD 0x00123456
71   Angel_SWI_Address    ; Reserved space for the Angel SWI Interrupt address
72       DCD 0x00000000
73
74   ; End of assembly code
75   END

```

A.11 interrupt/irq.h

```

1  #include "timer.h"
2
3  void install_handler (unsigned handler_routine_address , unsigned *
        vector_address);

```

A.12 interrupt/irq.c

```

1  /* C functions for hardware interruptions */
2
3  #include "irq.h"
4
5  /* Reserved spaces where the Angel IRQ/SWI addressess will be stored */
6  extern int   Angel_IRQ_Address;
7  extern int   Angel_SWI_Address;
8
9  /* Installs a handler branch on the interrupt vector */
10 void install_handler (unsigned handler_routine_address , unsigned *
        vector_address) {
11
12     /* Case it is running in the emulator or without angel */
13     if (emulator == 1 || emulator == 2) {
14         /* The instruction that will be put in the IRQ vector */
15         unsigned branch_to_handler_instruction;

```

```

16      /* Handler relative address */
17      unsigned offset;
18      /* -0x8 due to the pipeline, >> 2 due to the word alignment */
19      offset = ((handler_routine_address - (unsigned)vector_address - 0x8) >>
20                2);
21      /* Add to the address, the branch instruction */
22      branch_to_handler_instruction = 0xea000000 | offset;
23      /* Put the instruction in the vector */
24      *vector_address = branch_to_handler_instruction;
25      /* Case it is running with the angel */
26      else {
27          /* Angel branch instruction */
28          unsigned Angel_branch_instruction;
29          /* Angel instruction */
30          unsigned *Angel_address;
31          /* Getting Angel branch instruction */
32          Angel_branch_instruction = *vector_address;
33          /* Separate the instruction from the address */
34          Angel_branch_instruction ^= 0xe59ff000;
35          /* Calculating absolute address */
36          Angel_address = (unsigned *) ((unsigned)vector_address +
37                                         Angel_branch_instruction + 0x8);
38          /* Store address in the proper position */
39          if ((unsigned)vector_address == 0x18) {
40              Angel_IRQ_Address = *Angel_address;
41          }
42          else {
43              Angel_SWI_Address = *Angel_address;
44          }
45          /* Inserting handler instruction in the vector table */
46          *Angel_address = handler_routine_address;
47      }

```

A.13 interrupt/swi.h

```

1  #include "constants.h"
2
3  typedef void (*pt2Task)(int);
4

```

```

5  __swi(OS_SWI) int syscall(int, int, pt2Task, int);
6
7  int fork (void);
8  void exec (int, pt2Task, int);
9  void exit (int);

```

A.14 interrupt/swi.c

```

1  #include "swi.h"
2
3  extern void routine_fork(void);
4  extern void routine_exec(void);
5  extern void routine_exit(void);
6
7  int fork(){
8      int pid = 0;
9      pid = syscall(0, 0, 0, 0);
10     return pid;
11 }
12
13 void exec(int process_id, pt2Task process_addr, int arg1){
14     syscall(1, process_id, process_addr, arg1);
15 }
16
17 void exit(int process_id){
18     syscall(2, process_id, 0, 0);
19 }

```

A.15 mutex/mutex.h

```

1
2  #define WAIT      while (semaphore==1) {} mutex_gatelock();
3  #define SIGNAL    mutex_gateunlock();
4
5  extern unsigned volatile int semaphore; // do not access directly
6
7  void mutex_gatelock (void);
8  void mutex_gateunlock (void);

```

A.16 mutex/mutex.c

```

1 unsigned volatile int semaphore = 2; // this is a start value
2
3 void mutex_gatelock (void) {
4     __asm {
5         spin:
6         mov     r1, &semaphore
7         mov     r2, #1
8         swp     r3, r2, [r1]
9         cmp     r3, #1
10        beq     spin
11    }
12 }
13
14 void mutex_gateunlock (void) {
15     __asm {
16         mov     r1, &semaphore
17         mov     r2, #0
18         swp     r0, r2, [r1]
19     }
20 }

```

A.17 peripherals/button.h

```

1 #include "constants.h"
2
3 void button_init (void);
4 void button_irq (void);

```

A.18 peripherals/button.c

```

1 /* This file contains routines to initialize and handle button
2    interruptions */
3
4 #include "button.h"
5
6 /* Initializes the button */
7 void button_init (void) {
8     /* Force global disable off */

```

```

8      *(volatile int*)EvaluatorIRQTimerControl &= ~((1 << 21) | (1<<10) |
          (1<<0));
9      /* Enable int0 */
10     *(unsigned *)IRQButtonControl |= 1 << 4;
11     /* Set as active high */
12     *(unsigned *)IRQButtonControl |= 1 << 3;
13     /* Allow for rising edge */
14     *(unsigned *)IRQButtonControl|= 1;
15 }
16
17 /* Handles a button interruption */
18 void button_irq (void) {
19     *(unsigned *) IRQStatus |= 1;
20
21     /* Do something */
22
23 }

```

A.19 peripherals/dips.h

```

1 #include "constants.h"
2
3 unsigned dips_read (void);

```

A.20 peripherals/dips.c

```

1 /* This file contains routines to initialize and handle DIPS interruptions
   */
2
3 #include "dips.h"
4
5 /* Return the value of the dip switches */
6 unsigned dips_read (void)
7 {
8     /* 0xf = switch mask */
9     return 0XF & *IOData;
10 }

```

A.21 peripherals/led.h

```

1  /* LED changing functions */
2
3  #define LEDBANK      *((unsigned *)0x03ff5008)
4
5  #define LED_4_ON      (LEDBANK=LEDBANK|0x00000010)
6  #define LED_3_ON      (LEDBANK=LEDBANK|0x00000020)
7  #define LED_2_ON      (LEDBANK=LEDBANK|0x00000040)
8  #define LED_1_ON      (LEDBANK=LEDBANK|0x00000080)
9  #define LED_4_OFF     (LEDBANK=LEDBANK&~0x00000010)
10 #define LED_3_OFF     (LEDBANK=LEDBANK&~0x00000020)
11 #define LED_2_OFF     (LEDBANK=LEDBANK&~0x00000040)
12 #define LED_1_OFF     (LEDBANK=LEDBANK&~0x00000080)

```

A.22 peripherals/segment.h

```

1  #include "constants.h"
2
3  void segment_init (void);
4  void segment_set (int seg);

```

A.23 peripherals/segment.c

```

1  /* This file contains routines to initialize and handle the 7 segment
   display */
2
3  #include "segment.h"
4
5  /* Calculates the proper display addresses value according to the number */
6  static unsigned int numeric_display [16] = {
7      DISP_0,
8      DISP_1,
9      DISP_2,
10     DISP_3,
11     DISP_4,
12     DISP_5,
13     DISP_6,
14     DISP_7,
15     DISP_8,

```

```

16     DISP_9 ,
17     DISP_A ,
18     DISP_B ,
19     DISP_C ,
20     DISP_D ,
21     DISP_E ,
22     DISP_F
23 };
24
25 /* Set number on the display */
26 void segment_set (int seg) {
27     if ( seg >= 0 & seg <= 0xf ) {
28         *IOData    &= ~Segment_mask;
29         *IOData    |= numeric_display[seg];
30     }
31 }
32
33 /* Initialize 7-segment display */
34 void segment_init (void) {
35     *IOPMod |= Segment_mask;
36     *IOData |= Segment_mask;
37 }

```

A.24 peripherals/serial.h

```

1
2  /******
3   *
4   *  ARM Strategic Support Group
5   *
6   *  *****/
7
8  /******
9   *
10  *  Module      : serial.h
11  *  Description : simple code to drive the serial port on the
12  *                Evaluator7T.
13  *  Tool Chain  : ARM Developer Suite 1.0
14  *  Platform    : Evaluator7T
15  *  History     :
16  *

```



```

17  *      2000-3-29 Andrew N. Sloss
18  *      - started serial module
19  *
20  *****/
21
22 /******
23  *  IMPORT
24  *****/
25
26 // none...
27
28 /******
29  *  MACROS
30  *****/
31
32 #define BAUD_9600      (162 << 4)
33
34 #define COM1_DEBUG     (1)
35 #define COM0_USER      (0)
36
37 /******
38  *  DATATYPES
39  *****/
40
41 // none...
42
43 /******
44  *  STATICS
45  *****/
46
47 // none...
48
49 /******
50  *  ROUTINES
51  *****/
52
53 /* — serial_initcom0user —————
54  *
55  *  Description   : initializes the USER/COM0 serial port.
56  *
57  *  Parameters   : unsigned baudrate — baudrate i.e. 9600
58  *  Return       : none...
59  *  Notes       : none...

```

```

60  *
61  */
62
63  void serial_initcom0user (unsigned baudrate);
64
65  /* — serial_initcom1debug —————
66  *
67  * Description : initializes the DEBUG/COM1 serial port.
68  *
69  * Parameters : unsigned baudrate — baudrate i.e. 9600
70  * Return : none...
71  * Notes : none...
72  *
73  */
74
75  void serial_initcom1debug (unsigned baudrate);
76
77  /* — serial_print —————
78  *
79  * Description : print out a string through the com port
80  *
81  * Parameters : unsigned port — USER/DEBUG
82  *               : char *s — string to be printed out.
83  * Return : none...
84  * Notes : none...
85  *
86  */
87
88  void serial_print (unsigned port, char *s);
89
90  /* — serial_getkey —————
91  *
92  * Description : standard implementation of getkey.
93  *
94  * Parameters : none...
95  * Return : none...
96  * Notes :
97  *
98  *           waits until a key is pressed then echo's back.
99  *
100  */
101
102  void serial_getkey (void);

```

```

103
104
105 char serial_getcmd (void);
106
107 /* *****
108  * END OF serial.h
109  * *****

```

A.25 peripherals/serial.c

```

1  /* *****
2  *
3  *  ARM Strategic Support Group
4  *
5  * *****
6
7  /* *****
8  *
9  *  Module      : serial.c
10 *  Description : simple code to drive the serial port on the
11 *               Evaluator7T.
12 *  Tool Chain  : ARM Developer Suite 1.0
13 *  Platform    : Evaluator7T
14 *  History     :
15 *
16 *    2000-3-29 Andrew N. Sloss
17 *    - started serial module
18 *
19 * *****
20
21 /* *****
22 *  IMPORT
23 * *****
24
25 // none...
26
27 /* *****
28 *  MACROS
29 * *****
30
31 #define SYSCFG      (0x03ff0000)

```

```

32 #define UART0_BASE (SYSCFG + 0xD000)
33 #define UART1_BASE (SYSCFG + 0xE000)
34
35 /*
36  * Serial settings .....
37  */
38
39 #define ULCON 0x00
40 #define UCON 0x04
41 #define USTAT 0x08
42 #define UTXBUF 0x0C
43 #define URXBUF 0x10
44 #define UBRDIV 0x14
45
46 /*
47  * Line control register bits .....
48  */
49
50 #define ULCR8bits (3)
51 #define ULCRS1StopBit (0)
52 #define ULCRNoParity (0)
53
54 /*
55  * UART Control Register bits .....
56  */
57
58 #define UCRRxM (1)
59 #define UCRRxSI (1 << 2)
60 #define UCRTxM (1 << 3)
61 #define UCRLPB (1 << 7)
62
63 /*
64  * UART Status Register bits
65  */
66
67 #define USROverrun (1 << 0)
68 #define USRParity (1 << 1)
69 #define USRFraming (1 << 2)
70 #define USRBreak (1 << 3)
71 #define USRDTR (1 << 4)
72 #define USRRxData (1 << 5)
73 #define USRTxHoldEmpty (1 << 6)
74 #define USRTxEmpty (1 << 7)

```

```

75
76  /* default baud rate value */
77
78  #define BAUD_9600    (162 << 4)
79
80  // UART registers are on word aligned , D8
81
82  /* UART primitives */
83
84  #define GET_STATUS(p) (*(volatile unsigned *)((p) + USTAT))
85  #define RX_DATA(s)      ((s) & USRRxData)
86  #define GET_CHAR(p)     (*(volatile unsigned *)((p) + URXBUF))
87  #define TX_READY(s)     ((s) & USRTxHoldEmpty)
88  #define PUT_CHAR(p,c)   (*(unsigned *)((p) + UTXBUF) = (unsigned )(c))
89
90  #define COM1_DEBUG  (1)
91  #define COM0_USER  (0)
92
93  /* — serial_init —————
94  *
95  * Description   : wait until a key is press from the host PC.
96  *
97  * Parameters   : unsigned int port — com port either USER/DEBUG
98  *               : unsigned int baud — baud rate i.e. 9600
99  * Return      : none...
100 * Notes       : none...
101 *
102 */
103
104 void serial_init (unsigned int port, unsigned int baud)
105 {
106     /* Disable interrupts */
107     (*(volatile unsigned *) (port + UCON) = 0;
108
109     /* Set port for 8 bit , one stop , no parity */
110     (*(volatile unsigned *) (port + ULCON) = (ULCR8bits);
111
112     /* Enable interrupt operation on UART */
113     (*(volatile unsigned *) (port + UCON) = UCRRxM | UCRTxM;
114
115     /* Set baud rate */
116     (*(volatile unsigned *) (port + UBRDIV) = baud;
117

```

```

118 }
119
120 /* — serial_initcom0user —————
121 *
122 * Description : initializes the USER/COM0 serial port.
123 *
124 * Parameters : unsigned baudrate — baudrate i.e. 9600
125 * Return : none...
126 * Notes : none...
127 *
128 */
129
130 void serial_initcom0user (unsigned baudrate)
131 {
132     serial_init(UART0_BASE, baudrate);
133 }
134
135 /* — serial_initcom1debug —————
136 *
137 * Description : initializes the DEBUG/COM1 serial port.
138 *
139 * Parameters : unsigned baudrate — baudrate i.e. 9600
140 * Return : none...
141 * Notes : none...
142 *
143 */
144
145 void serial_initcom1debug (unsigned baudrate)
146 { serial_init(UART1_BASE, baudrate); }
147
148 /* — serial_print —————
149 *
150 * Description : print out a string through the com port
151 *
152 * Parameters : unsigned port — USER/DEBUG
153 *               : char *s — string to be printed out.
154 * Return : none...
155 * Notes : none...
156 *
157 */
158
159 void serial_print (unsigned port, char *s)
160 {

```

```

161  while ( *s != 0 ) {
162      switch ( port ) {
163          case COM0_USER:
164              while ( TX_READY(GET_STATUS(UART0_BASE))==0);
165              PUT_CHAR(UART0_BASE,*s++);
166              break;
167          case COM1_DEBUG:
168              while ( TX_READY(GET_STATUS(UART1_BASE))==0);
169              PUT_CHAR(UART1_BASE,*s++);
170              break;
171      }
172  }
173 }
174
175 /* — serial_getkey —————
176  *
177  * Description : standard implementation of getkey.
178  *
179  * Parameters : none...
180  * Return : none...
181  * Notes :
182  *
183  * waits until a key is pressed then echo's back.
184  *
185  */
186
187 void serial_getkey (void)
188 {
189     char c;
190
191     while ( (RX_DATA(GET_STATUS(UART0_BASE)))==0 )
192     {
193
194     };
195
196     c = GET_CHAR(UART0_BASE);
197
198     while ( TX_READY(GET_STATUS(UART0_BASE))==0);
199     PUT_CHAR(UART0_BASE,c);
200 }
201
202
203

```

```

204 /* -- serial_getkey -----
205 *
206 * Description : standard implementation of getkey.
207 *
208 * Parameters : none...
209 * Return : none...
210 * Notes :
211 *
212 * waits until a key is pressed then echo's back.
213 *
214 */
215
216 char serial_getcmd (void)
217 {
218     char c;
219
220     while ( (RX_DATA(GET_STATUS(UART0_BASE)))==0 )
221     {
222
223     };
224
225     c = GET_CHAR(UART0_BASE);
226
227     while ( TX_READY(GET_STATUS(UART0_BASE))==0);
228     PUT_CHAR(UART0_BASE, c);
229
230     return c;
231 }
232
233
234 /* *****
235 * END OF serial.c
236 * *****/

```

A.26 peripherals/timer.h

```

1 #include "constants.h"
2
3 void timer_init (void);
4 void timer_irq (void);
5 void timer_start (void);

```


A.27 peripherals/timer.c

```

1  /* This file contains routines to initialize and handle timer interruptions
   */
2
3  #include "timer.h"
4
5  /* Initiate timer settings */
6  void timer_init (void) {
7      /* Case it's from the emulator */
8      if (emulator == 1) {
9          /* Clear/disable all interrupts */
10         *IRQEnableClear = ~0;
11         /* Disable counters by clearing the control bytes */
12         *EmulatorIRQTimerControl = 0;
13         /* Clear counter/timer interrupts */
14         *IRQTimerClear = 0 ;
15     }
16     /* Case it's the board */
17     else {
18         /* Disable interrupt */
19         *TimerEnableSet = 0;
20         /* Clear pending interrupts */
21         *IRQStatus = 0x00000000;
22     }
23 }
24
25 /* Restart timer interrupt */
26 void timer_irq(void) {
27     if (emulator == 1) {
28         /* Clear the interrupt */
29         *IRQTimerClear = 0;
30     }
31     else {
32         /* Clear pending interrupts */
33         *IRQStatus = 1 << 10;
34         /* Load counter values */
35         *EvaluatorIRQTimerLoad = COUNTDOWN;
36         /* Unmask the interrupt source */
37         *(volatile int*)EvaluatorIRQTimerControl &= ~((1<<21) | (1<<10) |
38             (1<<0));
39     }
40 }

```

```

40
41 /* Start timer */
42 void timer_start (void) {
43     if (emulator == 1) {
44         /* Load counter values */
45         *EmulatorIRQTimerLoad = COUNTDOWN;
46         /* Enable the Timer | Periodic Timer producing interrupt | Set Maximum
           Prescale – 8 bits */
47         *EmulatorIRQTimerControl = (0x80 | 0x40 | 0x08 );
48         /* Enable interrupt */
49         *IRQEnableSet = IRQTimer;
50     }
51     else {
52         /* Load counter values */
53         *EvaluatorIRQTimerLoad = COUNTDOWN;
54         /* Enable interrupt */
55         *TimerEnableSet |= 0x1;
56         /* Unmask the interrupt source */
57         *(volatile int*)EvaluatorIRQTimerControl &= ~((1 << 21) | (1 << 10) |
           (1 << 0));
58     }
59 }

```

A.28 syscalls/exec.s

```

1 ; Exec system call
2
3 EXPORT routine_exec
4
5 IMPORT process_control_block
6
7 ; Beginning fork code
8 AREA exec, CODE
9
10 ; From the call of the function: r1 = task id, r2 = task address
11 routine_exec
12 ; Store variables
13 STMFD sp!,{r0-r12,lr} ; Push r0-12 in the stack
14
15 MOV r6, r3 ; r6 = value of the first argument
16

```

```

17 ; Put the task address in task_pcb_address - 4 (Process counter)
18
19 MOV    r0, r2                ; r0 = task address
20 ADD    r0, r0, #4            ; r0 = task address + 4 (+4 due to the pipeline
    )
21 LDR    r3, =process_control_block ; r3 = PCB bottom
22 MOV    r4, #68                ; r4 = 68
23 MUL    r5, r1, r4            ; r5 = (task id) * 68
24 ADD    r3, r3, r5            ; r3 = PCB bottom + (task id) * 68
25 SUB    r3, r3, #4            ; r3 = r3 - 4
26 STR    r0, [r3]              ; MEM[r3] = r0
27
28 ; Set up user stack for the task
29 SUB    r3, r3, #4            ; r3 = r3 - 4 (stack pointer)
30 MOV    r4, #0x20000          ; r4 = SP_USER_BOTTOM
31 loop
32 SUB    r1, r1, #1            ; r1 = task id - 1
33 CMP    r1, #0                ; r1 = 0 ?
34 BEQ    end_loop              ; if equal, end_loop
35 SUB    r4, r4, #4048          ; r4 = r4 - 4048 (next stack)
36 B      loop                  ; Go to next stack
37 end_loop
38 STR    r0, [r3]              ; MEM[r3] = r4 (the process stack pointer)
39
40 ; Set up the r0
41 SUB    r3, r3, #52            ; r3 = r3 - 52
42 STR    r6, [r3]
43
44 ; Set up link register
45 SUB    r3, r3, #4            ; r3 = r3 - 4 (link register)
46 MOV    r0, r2                ; r0 = task address
47 ADD    r0, r0, #4            ; r0 = r0 - 4
48 STR    r0, [r3]              ; MEM[r3] = r0
49
50 ; Set up SPSR
51 SUB    r3, r3, #4            ; r3 = r3 - 4
52 MOV    r0, #0x10             ; r0 = 0x10 (user mode)
53 STR    r0, [r3]              ; MEM[r3] = r0
54
55 ; Return
56 LDMFD  sp!, {r0-r12, pc}^ ; Pop r0-r12 and link register to process counter
57
58 ; End of assembly code

```

59 END

A.29 syscalls/exit.s

```

1 ; Exec system call
2
3 EXPORT  routine_exit
4
5 IMPORT  thread_array
6
7 ; Beginning fork code
8 AREA  exit, CODE
9
10 ; r1 comes as task id
11 routine_exit
12     STMFD  sp!,{r0-r12, lr}    ; save registers
13     MOV    r2, r1              ; r2 = task id
14     LDR    r0, =thread_array    ; r0 = thread_array
15     MOV    r1, #0              ; r1 is the state value = inactive
16     SUB    r2, r2, #1          ; r2 = task id - 1
17     MOV    r3, #4              ; r3 = 4
18     MUL    r4, r2, r3          ; r4 = (task id - 1) * 4
19     ADD    r4, r0, r4          ; r4 = thread_array + (task id - 1) * 4
20     STR    r1, [r4]            ; Mem[r4] = r1 (inactive)
21
22     LDMFD  sp!,{r0-r12, pc}^    ; return
23
24 ; End of assembly code
25     END

```

A.30 syscalls/fork.s

```

1 ; Fork system call
2
3 IMPORT  thread_array
4 IMPORT  process_control_block
5 IMPORT  current_thread_id
6
7 EXPORT  routine_fork
8

```

```

9      ; Beginning fork code
10     AREA fork, CODE
11
12     ; Routine to duplicate a process code
13     routine_fork
14     ; Stacks current state twice
15     STMFD sp!,{r1-r12,lr} ; Stacks the link register and r1-r12
16     STMFD sp!,{r0-r12} ; Stacks r0-r12
17     STMFD sp!,{lr} ; Stacks the link register (In a separate
        instruction
18     ; to stack it in the top)
19
20     ; Finds the first available space in the process table (return id in r0 and
        its address in r1)
21     LDR r1, =thread_array ; r1 = bottom of the thread array address
22     MOV r0, #1 ; r0 = 1
23     routine_fork_loop
24     LDR r2, [r1] ; r2 = thread array position
25     CMP r2, #0 ; r2 = 0?
26     BEQ pcb_bottom ; If the position is available (r2 = 0), go to
        pcb_bottom
27     ADD r0, r0, #1 ; r0 = r0 + 1 (next id)
28     CMP r0, #9 ; Is this the last thread slot being checked?
29     BEQ fork_fail ; if it is, there is no available slot, go to
        fork_fail
30     ADD r1, r1, #4 ; r1 = r1 + 4 (next address)
31     B routine_fork_loop ; Check next slot (go to routine_fork_loop)
32
33     ; Get the PCB bottom of the new process (return it in r2)
34     pcb_bottom
35     LDR r2, =process_control_block ; r2 = pcb_bottom
36     MOV r3, #68 ; r3 = 68
37     MUL r3, r0, r3 ; r3 = 68 * available thread id
38     ADD r2, r2, r3 ; r2 = pcb bottom + (thread id * 68)
39
40     ; Retrieves user mode stack pointer (returns it in r3)
41     SUB r13, r13, #4 ; Opens a space in the stack
42     STMIA r13, {r13}^ ; Store user mode stack pointer in the SVC stack
43     NOP ; No operation (necessary for the above instruction)
44     LDMFD sp!,{r3} ; r3 = user mode stack pointer
45
46     ; Retrieves user mode stack base (returns in r4)
47     MOV r4, #0x20000 ; r4 = 0x20000 (User mode stack pointer base)

```

```

48  MOV    r6, #4048          ; r6 = 4048 (Distance between each thread stack)
49  LDR     r5, =current_thread_id ; r5 = current thread id address
50  LDR     r5, [r5]          ; r5 = current thread id
51  SUB     r5, r5, #1         ; r5 = current thread id - 1
52  MUL     r6, r5, r6         ; r6 = (current thread id - 1) * 4048
53  SUB     r4, r4, r6         ; r4 = 0x20000 - (current thread id - 1) * 4048
54          ; (this is the base of the current thread stack)
55
56 ; Retrieves new thread stack base (returns in r5)
57  MOV     r5, #0x20000      ; r5 = 0x20000 (User mode stack pointer base)
58  MOV     r6, #4048         ; r6 = 4048 (Distance between each thread stack)
59  SUB     r7, r0, #1        ; r7 = new thread id - 1
60  MUL     r6, r7, r6         ; r6 = (new thread id - 1) * 4048
61  SUB     r5, r5, r6         ; r5 = 0x20000 - ((new thread id - 1) * 4048)
62
63 ; Duplicates stack
64 loop_stack_copy
65  LDR     r6, [r4]          ; r6 = original stack data
66  STR     r6, [r5]          ; Stores data in new stack (stack_top = r6)
67  CMP     r4, r3            ; Is this the top of the stack? (r4 == r3?)
68  BEQ     build_new_pcb     ; if it is, branch to build_new_pcb
69  SUB     r5, r5, #4         ; if not, go to next space in the new stack (r5 =
    r5 - 4)
70  SUB     r4, r4, #4         ; and next data in the original stack (r4 = r4 - 4)
71  B       loop_stack_copy    ; restart sequence (go to loop_stack_copy)
72
73 build_new_pcb
74 ; Store SPSR
75  SUB     r2, r2, #68       ; r2 = r2 - 68 (r2 = PCB[-68] address)
76  MOV     r3, #0x10         ; r3 = #0x10 (User mode)
77  STR     r3, [r2]          ; PCB[-68] = #0x10
78
79 ; Store stack pointer
80  ADD     r2, r2, #60       ; r2 = r2 + 60 (r2 = PCB[-8] address)
81  STR     r5, [r2]          ; PCB[-8] = new stack pointer
82
83 ; Stores r14 and LR
84  ADD     r2, r2, #4         ; r2 = r2 + 4 (r2 = PCB[-4] address)
85  LDMFD   sp!, {r3}         ; Restore link register from the stack to r3
86  ADD     r3, r3, #4         ; r3 = r3 + 4 (due to the pipeline)
87  STR     r3, [r2]          ; PCB[-4] = return address
88  SUB     r2, r2, #60       ; r2 = r2 - 60 (r2 = PCB[-64] address)
89  STR     r3, [r2]          ; PCB[-64] = return address

```

```

90
91 ; Copy registers
92 MOV    r3, #0      ; r3 = 0
93 MOV    r4, #12     ; r4 = 12
94 registers_loop
95 ADD    r2, r2, #4   ; r2 = r2 + 4 (Next PCB register space)
96 LDMFD  sp!, {r5}    ; Restore register from the stack to r5
97 STR    r5, [r2]     ; Store register in the PCB
98 CMP    r3, r4       ; r12 was copied? (r3 == r4?)
99 BEQ    enable_thread ; If yes, go to enable_thread
100 ADD    r3, r3, #1   ; r3 = r3 + 1 (Next register)
101 B      registers_loop ; Copy next register
102
103 ; Enable thread in the thread vector
104 enable_thread
105 MOV    r2, #1       ; r2 = 1
106 STR    r2, [r1]     ; New process in thread array = 1
107 LDMFD  sp!, {r1-r12,pc}^ ; Restore all the registers but r0
108                               ; (it contains the new process id)
109
110 ; Case when there is no thread space
111 fork_fail
112 LDMFD  sp!, {lr}     ; Restore link register
113 LDMFD  sp!, {r0-r12} ; Restore r0-r12
114 LDMFD  sp!, {r1-r12} ; Restore r1-r12
115 MOV    r0, #0xFFFFFFFF ; r0 = -1 (r0 is the return value)
116 LDMFD  sp!, {pc}^    ; Restore return address to the process counter
117
118 ; End of assembly code
119 END

```