ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Studio e realizzazione dell'emulatore μARM e del progetto JaeOS per la didattica dei Sistemi Operativi

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Marco Melletti

Sessione II
Anno Accademico 2015-2016

Indice

1 Introduzione **5**
- 1.1 Stato dell'arte . . . 6
- 1.2 μARM/JaeOS . . . 9

2 Hardware Emulato e Firmware **11**
- 2.1 Processore . . . 11
  - 2.1.1 Registri . . . 12
  - 2.1.2 Eccezioni . . . 13
  - 2.1.3 ISA . . . 14
  - 2.1.4 Coprocessore . . . 15
- 2.2 Bus . . . 16
  - 2.2.1 Registri di sistema . . . 16
  - 2.2.2 Device . . . 16
  - 2.2.3 Memoria . . . 18
- 2.3 BIOS e Libreria di Sistema . . . 21
  - 2.3.1 Servizi firmware . . . 21
  - 2.3.2 Funzioni di Libreria . . . 22

3 Interfaccia **23**
- 3.1 Interazione . . . 24
  - 3.1.1 Impostazioni . . . 24
  - 3.1.2 Esecuzione . . . 25
  - 3.1.3 Terminali . . . 27
- 3.2 Debugger . . . 27
  - 3.2.1 Stato Processore . . . 28
  - 3.2.2 Breakpoint . . . 29
  - 3.2.3 Visualizzatore Bus . . . 30
  - 3.2.4 Visualizzatore Strutture Statiche . . . 31
  - 3.2.5 Visualizzatore TLB . . . 31
- 3.3 Accessibilità . . . 33
  - 3.3.1 Annotazioni . . . 33

3.3.2 Modalità Accessibilità Aumentata 34

4 Progetto **35**
4.1 JaeOS Phase 1 36
4.2 JaeOS Phase 2 36
4.2.1 Gestione Processi 37
4.2.2 Active Semaphore List 37
4.2.3 Interrupt Handler 38
4.2.4 Low Level System Calls 38
4.3 JaeOS Phase 3 38
4.3.1 Memoria Virtuale 39
4.3.2 Gestione Dischi 40
4.3.3 Caricamento ed Esecuzione Dinamici 40
4.3.4 Temporizzazione 40
4.3.5 High (User) Level System Calls 41

5 Conclusioni e Futuri Sviluppi **42**

Appendices **48**

A Emulated Hardware Manual **49**
A.1 Introduction 50
A.2 Processor 52
A.2.1 Operating modes and Processor registers 52
A.2.2 System Coprocessor 55
A.2.3 Execution Control 57
A.2.4 Processor States 58
A.2.5 Exception Handling 60
A.3 System Bus 64
A.3.1 Reserved address space 64
A.3.2 Memory address space 70
A.4 Memory Interface 71
A.4.1 Physical addressing mode 71
A.4.2 Virtual addressing mode 72
A.5 External Devices 80
A.5.1 Device Registers 82
A.5.2 The Bus Device and Interval Timer 83
A.5.3 Disk Devices 84
A.5.4 Tape Devices 85
A.5.5 Network (Ethernet) Adapters 87
A.5.6 Printer Devices 90

A.5.7 Terminal Devices . . . 91
A.6 BIOS & System Library . . . 94
A.6.1 BIOS . . . 94
A.6.2 System Library . . . 95
A.7 Emulator Usage . . . 99
A.7.1 Compiling and Running the Machine . . . 99
A.7.2 `uarm-mkdev` tool . . . 111
A.7.3 Debugging . . . 112

B JaeOS Specifications 115
B.1 Phase 1 - Level 2: The Queues Manager . . . 115
B.1.1 The Allocation and Deallocation of ProcBlk's . . . 116
B.1.2 Process Queue Maintenance . . . 116
B.1.3 Process Tree Maintenance . . . 117
B.1.4 The Active Semaphore List . . . 119
B.1.5 Nuts and Bolts . . . 121
B.1.6 Testing . . . 121
B.2 Phase 2 - Level 3: The Nucleus . . . 122
B.2.1 The Scheduler . . . 122
B.2.2 Nucleus Initialization . . . 123
B.2.3 SYS/Bp Exception Handling . . . 125
B.2.4 PgmTrap Exception Handling . . . 129
B.2.5 TLB Exception Handling . . . 130
B.2.6 Interrupt Exception Handling . . . 130
B.2.7 Nuts and Bolts . . . 131
B.3 Phase 3 - Level 4: The VM-I/O Support Level . . . 134
B.3.1 SYS/Bp Exception Handling . . . 136
B.3.2 PgmTrap Exception Handling . . . 141
B.3.3 Delay Facility . . . 141
B.3.4 Virtual P and V Service . . . 142
B.3.5 Implementing Virtual Memory . . . 143
B.3.6 VM-I/O Support Level Initialization . . . 150
B.3.7 U-proc Initialization . . . 151
B.3.8 Nuts and Bolts . . . 153

Elenco delle figure

3.1 Schermata principale: Barra Principale 24
3.2 Configurazioni: pagina *General* 25
3.3 Configurazioni: pagina *Devices* 26
3.4 Configurazioni: pagina *Accessibility* 26
3.5 Visualizzatore terminale 27
3.6 Schermata principale: Stato del Processore 28
3.7 Schermata dei Breakpoint 29
3.8 Visualizzatore del Bus 30
3.9 Visualizzatore Strutture Statiche 32
3.10 Visualizzatore TLB 33

Capitolo 1

Introduzione

Il Costruzionismo è una teoria pedagogica, sviluppata da Seymour Papert con il gruppo "Epistemology and Learning" a cavallo tra gli anni '70 e '80, che può essere semplicisticamente ridotta al concetto di "*Imparare Facendo*" [1]. L'insegnamento della matematica nelle scuole elementari è stato il principale ambito di studio della teoria, tuttavia si tratta di una formulazione molto più completa ed adattabile a qualsiasi ambito dell'istruzione. Il Costruzionismo è legato ad una famiglia di teorie psicologiche che ricade sotto il nome di Costruttivismo, come spiega Papert "*[il costruzionismo] condivide con in costruttivismo la connotazione dell'apprendimento come "costruzione di strutture di conoscenza" a prescindere dalle circostanze dell'apprendimento. Vi aggiunge quindi l'idea che questo succeda in maniera particolarmente proficua in un contesto in cui lo studente sia consciamente partecipe nella costruzione di un'entità pubblica, sia che questa sia un castello di sabbia sulla spiaggia o una teoria dell'universo.*" [1]

Il presente lavoro mira a sviluppare uno strumento didattico di livello universitario, nello specifico per insegnamenti di *Sistemi Operativi* nell'ambito di Corsi di Laurea di primo livello; seguendo la teoria del Costruzionismo, il modo più adeguato per permettere agli studenti di apprendere al meglio una materia complessa e vasta come Sistemi Operativi è quello di coinvolgerli nello sviluppo di un intero sistema operativo. Tuttavia è necessario proporre agli studenti un'esperienza adeguata, in termini di complessità. Da questa osservazione scaturisce la filosofia della "Completeness over Complexity" [2] (completezza più che complessità), che ha guidato lo sviluppo di svariati strumenti per la didattica dei sistemi operativi [3].

Viene qui presentato un nuovo strumento, che discende da una prolifica stirpe [4–6], mirato all'insegnamento dei sistemi operativi, composto da μARM, un emulatore basato su processore ARM7TDMI, e JaeOS, un progetto per la didattica che richiede la costruzione di un piccolo sistema operativo partendo da zero.

L'emulatore è stato sviluppato tenendo a mente il principio della "Completeness over Complexity", bilanciando il rapporto tra realismo e complessità dell'interazione con

l'hardware, implementando un processore reale e dei dispositivi verosimili e funzionanti, ma semplificando il protocollo di comunicazione con questi ultimi.

Allo stesso modo, il progetto JaeOS è strutturato in modo da dare un assaggio della maggior parte delle tecniche utilizzate durante lo sviluppo del nucleo di un sistema operativo, senza concentrarsi troppo sull'utilizzo di algoritmi ottimizzati o tecniche avanzate, guidando, tuttavia, gli studenti attraverso lo sviluppo di un sistema completo e funzionante.

1.1 Stato dell'arte

La maggior parte dei corsi di Sistemi Operativi prevede, nel proprio materiale, un progetto pratico basato su un sistema operativo orientato alla didattica. Nel corso degli anni sono stati proposti progetti di vario genere in ambito accademico, questi possono essere raggruppati in quattro categorie, in base al tipo di lavoro che viene richiesto agli studenti:

Studio di un sistema operativo completo

Un esponente storico, appartenente a questa categoria, identifica l'approccio meno pratico allo studio dei sistemi operativi:

Minix [7] è un intero sistema operativo open source che può essere eseguito su hardware reale, accompagnato da un manuale completo; l'attività proposta, in questo caso, è quella di comprendere il funzionamento di un sistema operativo complesso, partendo dal manuale teorico e poi studiando il codice sorgente.

Questa categoria di attività didattiche è quasi interamente teorica, approccio diametralmente opposto al Costruzionismo che sta alla base del presente lavoro.

Estensione di un sistema operativo funzionante

Il primo degli approcci più pratici è quello dell'estensione di un sistema funzionante, in questo caso i progetti proposti richiedono, generalmente, di studiare un piccolo sistema operativo didattico ed estenderne le funzionalità, implementando moduli, driver o componenti mancanti. Alcuni esempi celebri sono:

OSP2 [8] Corredo di sistema operativo e macchina simulata, il codice fornito comprende entrambi i componenti scritti in Java, da compilare insieme; di fatto viene a mancare una separazione semantica netta tra l'hardware virtuale ed il sistema operativo, il che rende molto più complesso ottenere una visione di insieme corretta.

PintOS [9] Si tratta di un sistema operativo in grado di eseguire su architettura x86; è dotato di un ambiente speciale di testing per aiutare gli studenti nello sviluppo di funzioni di alto livello.

BabyOS [10] Un piccolo sistema operativo modulare pensato per sistemi embedded, funziona su architettura x86, il compito prevede lo studio del sistema e lo sviluppo di funzionalità di alto livello, come PintOS; oltre al differente obiettivo di impiego, questo sistema fornisce librerie che permettono un accesso molto più semplice alle funzionalità dell'hardware rispetto al precedente.

Xv6 [11] Una re-implementazione ridotta di Unix Versione 6 sviluppata in ANSI C per architettura x86 viene fornita agli studenti, richiedendo di studiare il sistema ed estenderlo implementando moduli aggiuntivi.

Completamento di un sistema operativo parziale

Una variante al metodo precedente, è quella di proporre come esercizio il completamento di un sistema operativo parziale. In questo caso si richiede agli studenti di implementare parti di un sistema operativo necessarie al suo funzionamento e che non sono state fornite insieme alla dotazione per svolgere l'esercizio. Progetti di questo tipo sono i seguenti:

Topsy [12] Un sistema operativo basato su *microkernel* in grado di funzionare su architettura MIPS e i386, mancante in alcune sue parti; questo progetto richiede il completamento del sistema operativo e lo sviluppo di alcuni moduli di alto livello, purtroppo è carente di un corredo curricolare che guidi lo sviluppo del progetto.

OS/161 [13] Viene fornito, insieme all'emulatore di hardware MIPS per cui è progettato, un sistema operativo parziale carente di alcune parti principali. Viene richiesto agli studenti di implementare i moduli mancanti e re-implementare lo scheduler di sistema come attività di progetto.

Awk-Linux [14] È un sistema integrato che contiene un simulatore ed un sistema operativo basato su una versione semplificata del Kernel Linux, interamente sviluppato in Awk. Il compito per gli studenti è quello di implementare le funzionalità mancanti utilizzando algoritmi di alto livello.

GeekOS [15] Viene fornito agli studenti un piccolo sistema operativo progettato per architettura i486 insieme ad un emulatore di questo hardware; il sistema operativo agisce come basso livello di un sistema, di cui gli studenti devono implementare funzionalità avanzate senza dover interagire direttamente con l'hardware, che è invece gestito dal codice fornito. I requisiti del progetto sono la realizzazione di file system, scheduler a priorità, gestione della memoria virtuale e sincronizzazione tra processi.

Le differenze principali tra questa categoria e la precedente sono gli obiettivi da sviluppare, che in questo caso possono essere anche componenti centrali e fondamentali di un sistema operativo, e la maggiore complessità del compito, poiché richiede più confidenza con le tecniche di debug rispetto al precedente, dove si parte da un programma funzionante.

Costruzione di un sistema operativo completo

L'ultima categoria è quella che richiede lo sviluppo di un sistema operativo completo a partire da zero; in questo caso, vengono forniti agli studenti la documentazione tecnica, che dichiara le specifiche di funzionamento del sistema operativo da costruire, ed un ambiente in cui sviluppare e testare il progetto. Fanno parte di questa categoria:

Nachos [16] Viene fornito un simulatore di architettura MIPS per cui va sviluppato un sistema operativo integrato; come per OSP2 è necessario compilare il codice del simulatore insieme a quello del sistema operativo, rendendo poco chiara la distinzione tra hardware e software.

PortOS [17] Progetto mirato alla soluzione di problemi di rete e connettività per dispositivi portatili, richiede lo sviluppo di un sistema operativo su più livelli per architetture StrongARM/x86. Gli studenti devono sviluppare il sistema operativo utilizzando una macchina virtuale, un obiettivo centrale è quello di fornire supporto affidabile al livello applicazione di uno stack di rete in condizioni di connettività instabile.

μMPS2/Kaya [5, 6] Il predecessore dello strumento proposto in questo lavoro; vengono forniti un simulatore di hardware e le specifiche di un sistema operativo multi-livello che gli studenti devono implementare sul simulatore, basato su architettura MIPS.

From NAND to Tetris [18] Partendo dalla costruzione di porte logiche, implementando l'hardware, un assemblatore ed un compilatore, per poi concludere con un piccolo sistema operativo, questo progetto garantisce un'esperienza decisamente completa sul funzionamento dei calcolatori. Ha, però, una portata troppo ampia per essere assegnato durante un corso di sistemi operativi, è invece un ottimo progetto per concludere un percorso di laurea triennale in materie informatiche.

Questo tipo di attività richiede un quantitativo di impegno maggiore da parte degli studenti, ma permette una più semplice e completa comprensione del funzionamento dei sistemi operativi nel loro complesso. L'ampiezza della visione della materia è analoga a quella fornita dalla prima categoria, ma l'approccio più pratico di questo tipo di progetti rende più semplice l'interiorizzazione dei concetti.

1.2 μARM/JaeOS

Il motto che ha guidato lo sviluppo del materiale qui presentato è "Completeness over complexity", completezza piuttosto che complessità, ed è il medesimo che sta alla base dello sviluppo di μMPS2/Kaya. Con questa frase si vuole enfatizzare l'importanza di dare una visione completa della materia, senza necessariamente concentrarsi su dettagli complessi che possono risultare interessanti soltanto per specifici casi d'uso.

La prima conseguenza di questa filosofia è l'impiego di un emulatore di hardware, come base di sviluppo, che sia sufficientemente semplice per essere compreso a pieno dagli studenti. Questo significa che svariati dettagli, come l'interfaccia hardware per i device esterni, sono stati semplificati cercando di mantenere un livello di verosimiglianza sufficiente rispetto all'hardware reale. Le semplificazioni introdotte consentono di comprendere i principi del funzionamento senza dover studiare estesi manuali tecnici relativi a specifiche implementazioni.

Per fornire un'esperienza apprezzabile e realistica, con μMPS/μMPS2 veniva fornito un simulatore hardware composto da un processore MIPS R3000, largamente utilizzato al periodo, modificato per supportare la disattivazione programmata della traduzione di indirizzi virtuali di memoria, caratteristica dell'hardware reale che introduceva una notevole difficoltà nelle fasi iniziali di sviluppo. Il simulatore comprendeva, inoltre, un bus di sistema a cui erano collegati una serie di dispositivi esterni a blocchi (nastri, dischi), a caratteri (terminali, stampanti) e di rete; per semplicità tutti i registri di controllo dei dispositivi erano mappati sullo spazio di indirizzi della memoria. Infine era presente un meccanismo di traduzione degli indirizzi logici basato su TLB, e attraverso il BIOS di sistema veniva implementato un semplice schema di paginamento della memoria.

Secondo la categorizzazione precedentemente proposta (cfr. sec. 1.1), la tipologia di attività progettuale più adatta a dare una visione di insieme del funzionamento di un sistema operativo è l'ultima, cioè la costruzione di un intero sistema operativo partendo dall'hardware (emulato). La componente di completezza a discapito della complessità entra in gioco nella stesura dei requisiti del sistema da sviluppare: invece di richiedere l'impiego di algoritmi e soluzioni avanzate o particolarmente efficienti, seppur note, si propone un sistema di cui sia necessario sviluppare tutti gli aspetti chiave, suggerendo algoritmi non sofisticati e lasciando la scelta dei dettagli implementativi interamente in mano agli studenti. In questo modo, seguendo la filosofia del Costruzionismo, gli studenti saranno in grado di scoprire ed apprendere le nozioni chiave in maniera indipendente, andando a sviluppare loro stessi gli aspetti principali del sistema e tutta l'infrastruttura che fa da collante e permette ai singoli algoritmi e componenti di interagire, andando a formare un sistema operativo. Una volta compresi gli obiettivi delle varie componenti del sistema ed i meccanismi di interazione, gli studenti avranno gli strumenti per capire quali sono le interconnessioni tra i molti argomenti che compongono un corso di Sistemi Operativi, obiettivo difficile da raggiungere senza un'esperienza così completa e diretta.

I due elementi precedentemente introdotti costituiscono le fondamenta del progetto

μMPS2/Kaya, da cui μARM/JaeOS eredita la filosofia ed alcune caratteristiche dell'emulatore e della documentazione del progetto. La principale motivazione che ha portato alla creazione del nuovo strumento è l'età del chip su cui si basava μMPS2, attualmente caduto in disuso. Di conseguenza è stato necessario effettuare una scelta ponderata su quale architettura fosse la più adeguata per un nuovo emulatore.

La scelta è ricaduta sull'ARM7TDMI in quanto l'architettura ARM sta prendendo piede da alcuni anni ed è largamente utilizzata; inoltre la generazione di processori a cui appartiene quello designato è una famiglia di macchine RISC, questo significa che il set di istruzioni che mettono a disposizione ai programmatori è limitato e rapido da imparare, in questo modo, anche senza studiare l'ISA in dettaglio, gli studenti possono comprendere a grandi linee ed in maniera intuitiva il funzionamento del codice assembly generato per questa architettura.

Un altro punto di forza del processore scelto è la vastità di casi d'uso reali, da cui deriva una notevole libertà da vincoli architetturali imposti al simulatore. Questa caratteristica ha permesso di mantenere un'architettura molto simile a quella di μMPS2, uno dei cui punti di forza era il bilanciamento tra semplicità e plausibilità, mantenendo quindi questa ottima proprietà.

Costruendo la nuova macchina μARM, è stata posta attenzione anche su un punto precedentemente non considerato: l'accessibilità del software. Sfruttando tecnologie attuali, è possibile realizzare programmi grafici multi piattaforma in grado di interagire con gli strumenti per l'accessibilità ormai supportati dai maggiori sistemi operativi, permettendo così a studenti diversamente abili di intraprendere l'attività progettuale.

Verranno in seguito discusse le scelte implementative per quanto riguarda l'hardware della macchina emulata μARM, quelle riguardanti l'interfaccia grafica ed il debugger ed infine verrà presentato il progetto proposto come completamento del corredo.

Capitolo 2

Hardware Emulato e Firmware

Un obiettivo di questo lavoro è quello di fornire un ambiente sufficientemente realistico su cui costruire un sistema operativo, contenendo il più possibile le complessità omissibili, quali, ad esempio, device complessi con interfacce estese.

Per bilanciare questi due fattori si è scelto, come base, un esponente di una famiglia di processori particolarmente prolifica e diffusa al giorno d'oggi: *ARM* [19]. Al processore è stato collegato un *bus principale* che permette di scambiare informazioni ed inviare comandi ad un *memory subsystem* e ad un insieme di *device*.

Verranno di seguito accennate le caratteristiche dell'hardware realizzato, si rimanda all'Appendice A per una più accurata descrizione della struttura e del funzionamento dello stesso.

2.1 Processore

Nella filosofia della "*Completeness over Complexity*" [2], il tipo di processore più adeguato per l'emulatore sviluppato è un processore *RISC*: Reduced Instruction Set Computing. Questa sigla indica un metodo di progettazione di processori mirato a limitare il numero di istruzioni conosciute dal processore e diminuire il tempo di esecuzione di ogni singola istruzione, ottenendo migliori performance grazie ad una struttura più semplice e lineare.

I processori ARM (acronimo di Advanced RISC Machines) sono ormai estremamente diffusi ed utilizzati per ogni genere di applicazione, dai sistemi embedded agli smartphone, dalle console portatili ai netbook, è una famiglia che vanta un sempre crescente numero di membri dalle diverse connotazioni e potenzialità. Come il nome stesso suggerisce, si tratta in buona parte di processori che implementano il design RISC[1], quindi perfetti per soddisfare il requisito di semplicità.

[1]Questo è stato generalmente vero fino a pochi anni fa, le ultime versioni delle ISA ARM e Thumb comprendono un set molto esteso di istruzioni ed i modelli più evoluti di questi processori hanno raggiunto un livello di complessità non più definibile RISC.

L'utilizzo su larga scala di questi processori è un buon presupposto per quanto riguarda la plausibilità del sistema. Nello specifico si è posta molta attenzione sull'utilizzo di questi processori con sistemi embedded, console portatili e soprattutto *single-board computer* come *Raspberry Pi* [20]. La scelta è in ultimo ricaduta sul modello *ARM7TDMI* [21], che implementa una diffusissima versione dell'ISA ARM e la prima versione dell'ISA Thumb, utilizza registri a 32 bit ed istruzioni a 32 o 16 bit in base all'instruction set selezionato. Questo modello si basa su un'architettura a pipeline a tre stadi, con schema *fetch-decode-execute*, in cui ogni istruzione binaria attraversa i tre stadi della pipeline, che possono essere simultaneamente occupati da diverse istruzioni, ed hanno rispettivamente il ruolo di:

***fetch*:** caricamento dell'istruzione dalla memoria (verifica implicita della validità dell'indirizzo di memoria);

***decode*:** decodifica dell'istruzione (verifica della validità del codice istruzione);

***execute*:** esecuzione effettiva dell'istruzione.

2.1.1 Registri

Una peculiarità del processore scelto (e di buona parte della gamma dei processori ARM) è la struttura dei registri interni: 17 registri sono sempre accessibili, ma una parte di questi è legata alla modalità di esecuzione del processore, permettendo così di alleggerire il costo di un context switch, soprattutto in applicazioni embedded.

Alcuni registri hanno dei ruoli ben precisi, suggeriti dal manuale tecnico ed utilizzati dai compilatori, così da permettere ad applicazioni o librerie a basso livello, programmate in linguaggio *Assembly*, di interagire in maniera sufficientemente lineare con codice scritto in linguaggi di più alto livello, come il *C*.

Queste due caratteristiche hanno permesso di realizzare un semplice firmware ed una libreria di sistema che implementano una versione completa del context switch, salvando e caricando l'intero set di registri visibili al momento della chiamata.

I registri globali (sempre accessibili, a prescindere dalla modalità di esecuzione) sono dieci: i primi otto sono general purpose, di cui i primi quattro sono utilizzati per passare argomenti ed ottenere risultati a/da chiamate di funzione e gli ultimi quattro contengono variabili locali; gli ultimi due sono rispettivamente il *program counter*, che tiene traccia delle istruzioni in esecuzione, e lo *status register*, un registro speciale tramite cui è possibile modificare il comportamento del processore, cambiando livello di privilegi, inibendo interruzioni o selezionando la modalità di interpretazione del codice binario.

L'insieme di registri è, in realtà, ulteriormente diviso nel set accessibile in modalità ARM e quello disponibile in modalità Thumb, quest'ultimo è composto dall'insieme di registri globali più due registri dedicati per ogni modalità di esecuzione.

Si indica con *stato del processore* l'insieme dei valori contenuti nei registri attivi in un dato momento dell'esecuzione. In seguito si farà riferimento al salvataggio ed al caricamento di stati del processore intendendo rispettivamente: la copia di tutti i valori contenuti nei suoi registri in un'area contigua di memoria ed il caricamento dei valori da una area di memoria nei registri del processore.

2.1.2 Eccezioni

Le modalità di esecuzione del processore citate in precedenza sono dipendenti dalle varie eccezioni che questo è in grado di riconoscere. Sono disponibili in tutto sette modalità di esecuzione, due sono riservate per il normale funzionamento del processore e hanno diversi livelli di privilegi (*User mode* e *System mode*), le altre cinque modalità vengono attivate automaticamente in risposta al verificarsi di errori o interruzioni.

Per quanto riguarda le interruzioni, il processore è in grado di distinguerne tre tipi:

Interrupt: interruzioni rilevate mediante una linea di connessione esterna del processore;

Fast Interrupt: interruzioni rilevate mediante una seconda linea di connessione esterna del processore;

Software Interrupt: interruzioni richieste attraverso una specifica istruzione dal programma in esecuzione.

I primi due tipi di interruzione sono generati da dispositivi esterni collegati al processore e servono ai dispositivi per notificare un qualche tipo di evento, come la conclusione di un'operazione; la differenza tra questi due tipi sta nel quantitativo di registri dedicati: i fast interrupt possono contare su sette registri indipendenti che permettono ad un gestore di questo tipo di eccezioni di svolgere le proprie mansioni senza effettuare nessun context switch, a patto di essere appositamente progettato; gli interrupt normali invece hanno soltanto due registri indipendenti, come tutte le altre modalità di eccezione (i rimanenti cinque registri sono condivisi), richiedendo un context switch se il gestore delle eccezioni deve svolgere compiti complessi.

Il terzo tipo di interruzione ferma la regolare esecuzione per entrare in modalità *Supervisor* e permette la realizzazione di un gestore di servizi di sistema.

Sono inoltre differenziati tre tipi di errore:

Data Abort: errore di accesso alla memoria con operazioni di load/store;

Prefetch Abort: errore di accesso alla memoria durante il caricamento delle istruzioni da eseguire (fase di *fetch*);

Undefined Exception: istruzione non riconosciuta.

Le prime due tra queste tre eccezioni sono entrambe gestite in modalità *Abort*, ma è possibile associarvi routine separate, l'ultima invece viene gestita in modalità *Undefined*.

Ogni volta che viene sollevata un'eccezione tra quelle precedentemente elencate il processore cambia modalità di esecuzione ed effettua un salto ad un indirizzo specifico, differente per ogni tipologia di errore. Questo meccanismo permette la realizzazione di alcuni gestori delle eccezioni di basso livello; i gestori sono implementati nel BIOS e si occupano di effettuare un context switch, salvare lo stato del processore precedente all'eccezione, indicare in un registro apposito del coprocessore la causa dell'errore, ed infine avviarne la gestione a livello kernel (mediante il caricamento di zone specifiche di memoria che dovrebbero contenere appositi stati del processore, preparati dal sistema operativo in fase di inizializzazione).

2.1.3 ISA

L'*Instruction Set Architecture* (ISA) è la parte dell'architettura di un processore che ne definisce l'insieme di istruzioni eseguibili. I processori ARM, a partire dal modello realizzato, ARM7TDMI, implementano contemporaneamente due insiemi di istruzioni differenti, uno a 32 bit più ricco (ARM ISA) ed uno più ristretto con istruzioni a 16 bit (Thumb ISA). Questo secondo set di istruzioni è stato introdotto per aumentare la densità del codice, permettendo di salvare in un'unica parola di memoria di 32 bit due istruzioni da 16 bit.

Il processore può passare dall'interpretazione di un instruction set all'altro mediante un'apposita istruzione, oppure al variare del valore di un flag nel registro di stato. Questo secondo metodo permette di caricare automaticamente la modalità di interpretazione inseme al resto dello stato del processore, nel caso di un context switch.

ARM

Il primo instruction set storicamente utilizzato dai processori ARM è un insieme di istruzioni a dimensione fissa di 32 bit: ARM ISA. Questo instruction set si basa su un'architettura di tipo *load/store* [22], in cui le operazioni aritmetiche possono essere effettuate solamente sul contenuto dei registri del processore e l'accesso alla memoria avviene unicamente mediante istruzioni dedicate di caricamento/salvataggio.

I 4 bit più significativi di ogni istruzione binaria permettono di specificare una condizione, la cui valutazione è preposta all'esecuzione dell'istruzione stessa e la inibisce, nel caso in cui questa condizione non risulti valida. I seguenti 4 bit individuano nove classi di istruzioni; per molte di queste gli ulteriori 4 bit a seguire indicano l'istruzione specifica. Infine il secondo quartetto di bit meno significativi contiene un ultimo codice utilizzato per discriminare altre operazioni.

In sunto, esaminando 12 bit dell'istruzione binaria è semplice trovare una corrispondenza immediata con il comando da eseguire; questa struttura estremamente regolare [23]

ha permesso la creazione di una matrice ad accesso diretto che velocizza e semplifica il processo di decodifica delle istruzioni binarie.

Thumb

L'insieme di istruzioni Thumb conta un numero inferiore di istruzioni rispetto al precedente, ognuna descritta da un valore binario di 16 bit, tuttavia mantengono uno schema fisso descritto da 8 bit, del tutto analogo a quello visto in precedenza [23]. È quindi risultato altrettanto semplice utilizzare lo stesso meccanismo per realizzare l'interprete per questo set di istruzioni.

2.1.4 Coprocessore

È possibile collegare fino a 16 coprocessori al processore ARM7TDMI per estenderne le funzionalità, l'unico di cui vengano fornite le specifiche è però un coprocessore speciale, generalmente collegato nell'ultima posizione, chiamato *system control coprocessor* o coprocessore di sistema.

Il coprocessore di sistema è dotato di una serie di registri *special purpose* che permettono di selezionare impostazioni speciali, soprattutto riguardo la gestione della memoria, ed ottenere informazioni sul sistema. I documenti di specifica del processore indicano una struttura generale per il coprocessore di sistema, lasciando la scelta dei dettagli implementativi al costruttore del coprocessore. Questa libertà ha permesso di strutturare il coprocessore di sistema nella maniera più adeguata all'ambiente in cui è installato il processore, cercando comunque di rispettare le indicazioni della documentazione nella maniera più fedele possibile.

Il coprocessore si occupa della gestione della memoria virtuale, di segnalare i codici delle eccezioni e, nel caso di data/prefetch abort, di segnalare indirizzi errati; fornisce inoltre informazioni sul sistema secondo le specifiche ARM e gestisce la sospensione (*WAIT*) e lo spegnimento (*HALT*) della macchina emulata.

Un coprocessore spesso associato al ARM7TDMI è quello che effettua operazioni in virgola mobile, tuttavia queste funzionalità possono essere introdotte anche attraverso librerie di sistema, a patto di un maggior costo computazionale. Si è dunque scelto di non realizzare questo coprocessore, poiché i casi di necessità di questo tipo di operazioni sono molto limitati nello svolgimento dell'esercizio a cui è finalizzato lo strumento oggetto della trattazione, di conseguenza si è optato per la realizzazione di una più semplice libreria contenente alcune implementazioni di algoritmi per le divisioni intere (cfr. sec. 2.3.2).

2.2 Bus

Per costruire un sistema completo, è necessario un meccanismo che permetta a tutte le componenti in gioco di scambiare informazioni e comandi. È stato quindi realizzato un bus di sistema, concettualmente ispirato ad un semplice bus seriale[2], che permette al processore di interagire con tutti i componenti a cui non è direttamente collegato.

Il bus è concepito come uno spazio di indirizzamento diretto, il processore ed i dispositivi collegati possono accedere in scrittura o lettura agli indirizzi del bus (la cui massima dimensione è 32 bit), implementando a tutti gli effetti un meccanismo di comunicazione bidirezionale.

I dispositivi collegati al bus sono noti al suo controller emulato, che effettua un primo controllo di validità degli indirizzi e solleva eccezioni di tipo data/prefetch abort nel caso in cui venga richiesto un indirizzo non associato a nessun dispositivo.

2.2.1 Registri di sistema

Sui primi indirizzi considerati validi sono mappati un insieme di registri speciali che vengono utilizzati dal processore per gestire le eccezioni. Quando la normale esecuzione viene interrotta, viene eseguito il contenuto del registro associato al tipo di eccezione che ha generato l'interruzione. Sfruttando questo meccanismo è stato possibile realizzare i gestori a basso livello delle eccezioni (cfr. sec. 2.1.2).

Inoltre sono presenti un piccolo insieme di registri speciali che forniscono informazioni sulla dimensione della memoria e sull'orologio interno del sistema.

2.2.2 Device

I dispositivi collegati al bus sono un insieme di device derivati da CHIP [24] e presenti nei predecessori dello strumento realizzato [4, 6], ognuno dei quali mira ad essere una buona approssimazione della versione reale del dispositivo stesso.

Analogamente a quanto considerato dai lavori precedentemente citati, non è necessario realizzare versioni estremamente dettagliate dei dispositivi e soprattutto dell'interfaccia che questi espongono al sistema. Tutte le complicazioni derivanti dai dettagli implementativi farebbero passare in secondo piano quello che è il meccanismo base di comunicazione tra processore e dispositivi ed il funzionamento generale delle varie tipologie di device. Si è quindi scelto di utilizzare questi dispositivi, in quanto ben rappresentativi dei classici device generalmente disponibili nelle macchine general purpose, ma che non richiedono attento studio di complessi documenti di specifiche per la realizzazione di driver funzionanti.

[2]Il bus di sistema emulato gestisce le richieste in maniera sequenziale, di conseguenza non è assimilabile ad un bus parallelo ed è intrinsecamente immune ai problemi di disallineamento del clock e crosstalk dei bus fisici.

I dispositivi implementati sono elencati di seguito, ordinati in base alla numerazione utilizzata dal sistema.

Timer

Il primo e più semplice device è un interval timer, il cui valore numerico può essere impostato dal processore e viene costantemente decrementato ad ogni ciclo di clock; il timer genera un interrupt nel momento in cui, aggiornando il valore, si verifica un underflow, quando, cioè, passa da `0` a `-1` (in rappresentazione esadecimale `0xFFFF.FFFF`).

Questo dispositivo è fondamentale per realizzare qualsiasi schema di temporizzazione nel sistema, primo di tutti uno scheduler, componente alla base di ogni sistema operativo multitasking.

Disco

La seconda classe di dispositivi è quella dei dischi magnetici, ogni disco virtuale deve essere creato, prima dell'utilizzo, con il tool `uarm-mkdev`, tramite cui si può impostare la geometria del disco. I controller virtuali dei dischi sono in grado di effettuare trasferimenti di dati con meccanismo DMA [25], che significa che i dati sono spostati direttamente dalla memoria al disco e viceversa in blocchi da 4KB, senza occupare il processore.

Utilizzando i dischi è possibile sperimentare la realizzazione di meccanismi di swapping, la creazione di file system ed il caricamento ed avvio dinamico di programmi da supporti esterni.

Tape

I dispositivi a nastro rappresentano l'intera classe di dispositivi ad accesso sequenziale in sola lettura, come ad esempio i dischi ottici. Questi dispositivi devono essere creati tramite il tool `uarm-mkdev`, che precarica file o immagini binarie sul nastro.

I nastri, essendo dispositivi ad accesso sequenziale, permettono soltanto operazioni di movimento della testina sul nastro e di lettura del blocco di 4KB che si trova sotto la testina, anch'essi con meccanismo DMA.

I nastri, similmente ai dischi ottici, sono ottimi mezzi per trasportare dati o programmi ed eventualmente copiarli su dischi magnetici. Possono essere utilizzati in maniera più rapida dei dischi per sperimentare l'avvio dinamico di programmi da supporti esterni, poiché non è necessario riempirli a run time.

Stampante

Le stampanti appartengono alla classe dei device di sola scrittura, poiché non sono dotate di alcuna capacità di leggere i dati che hanno scritto. Le stampanti implementate scrivono testo codificato secondo lo standard ASCII [26] direttamente su file.

Sono la seconda classe di dispositivi più semplice e permettono di generare output permanente, a differenza dei terminali[3].

Terminale

La classe dei terminali realizza dispositivi di interfaccia a caratteri che permettono all'utente di interagire con il sistema. Poiché ogni terminale è un device bidirezionale, il meccanismo di comunicazione ha una struttura differente rispetto a quelli dei device descritti in precedenza: consta di due sotto-interfacce indipendenti, una per la trasmissione di caratteri ed una per la ricezione.

I terminali sono strumenti importanti che permettono la realizzazione di programmi interattivi, inoltre sono la maniera più comoda per ottenere un riscontro sul funzionamento della macchina emulata attraverso stampe di controllo che vengono visualizzate in tempo reale.

Rete

Le interfacce di rete sono basate su *libvde* [27], implementano quindi interfacce ethernet virtuali che possono essere connesse, utilizzando gli strumenti forniti da VDE, con uno switch virtuale presente sulla macchina host.

Le interfacce di rete possono essere utilizzate per testare protocolli di rete o connettere l'emulatore a vari tipi di reti, tra cui, volendo, anche Internet.

2.2.3 Memoria

Gli ultimi due componenti collegati al bus di sistema sono le memorie ad accesso veloce: una ROM contenente il BIOS di sistema ed una RAM di dimensione variabile.

Entrambe sono mappate sul bus, ciò significa che ad intervalli specifici di indirizzi del bus corrisponde l'accesso ad una o l'altra memoria. Questo meccanismo, seppur riducendo in piccola parte lo spazio di indirizzamento della RAM, è uno dei più comunemente utilizzati nei computer di consumo, inoltre semplifica l'interazione tra il processore ed il sistema, poiché tutte le possibili interazioni sono effettuabili attraverso un unico spazio logico.

ROM

La ROM è una memoria di sola lettura e dimensioni contenute (massimo $\sim$108KB), al cui interno deve essere caricato il BIOS di sistema per poter effettuare un avvio corretto.

[3] I terminali sono stati dotati di funzionalità di scrittura su file per rendere l'analisi dell'output più semplice, ma concettualmente il loro output dovrebbe essere volatile.

Il processore ARM, appena avviato, esegue l'istruzione che si trova nel primo indirizzo del Bus (che corrisponde all'eccezione speciale *reset*); la zona di memoria all'inizio dello spazio degli indirizzi è, però, riservata per la gestione delle eccezioni, non è quindi possibile scrivere l'intero codice di avvio direttamente in quell'area. Per ovviare a questo problema senza rinunciare alle funzionalità di bootstrap e di libreria fornite dal BIOS, è stata collegata la memoria ROM al Bus e l'istruzione nella parola di memoria associata al reset è fissata in hardware in modo da effettuare un salto incondizionato verso l'inizio della ROM.

Nonostante la memoria sia di sola lettura, è possibile selezionare un file binario contente il codice del BIOS prima dell'avvio dell'emulatore, in questo modo si possono sperimentare diverse tecniche di avvio, tra cui l'avvio da dispositivo esterno, per cui è necessario il caricamento in memoria del codice operativo ed il relativo avvio.

Fisica

La memoria principale del sistema è una memoria RAM little-endian di dimensione variabile (massimo 4GB). È presente un controller della memoria che effettua semplici controlli di accesso, implementa il meccanismo DMA e realizza uno schema di memoria virtuale in congiunzione con il coprocessore di sistema.

All'avvio la memoria virtuale non è attiva, di conseguenza il controller della RAM permette l'accesso diretto da parte del processore a tutto il suo contenuto e costruisce una corrispondenza 1:1 tra gli indirizzi logici del bus e gli indirizzi fisici di memoria. Questo è lo schema di avvio più semplice, che permette di preparare le strutture dati necessarie al buon funzionamento della memoria virtuale senza preoccuparsi di dover gestire indirizzamento virtuale non inizializzato.

Virtuale

L'indirizzamento logico della memoria virtuale, per quanto più complesso da gestire, è fondamentale per realizzare sistemi operativi multi processo con gestione trasparente della memoria, in cui ogni processo abbia l'illusione di avere l'intero spazio di indirizzamento di memoria privata a disposizione. In realtà tutti i processi del sistema condividono lo stesso spazio di indirizzi, però sfruttando la tecnica del *Paging* [28] ed un controller hardware in grado di effettuare la traduzione di indirizzi virtuali in indirizzi fisici, si può costruire uno schema di accesso indipendente dai vari spazi logici degli indirizzi, definito memoria virtuale.

Il paging è una tecnica generale, di cui esistono svariate implementazioni proposte, che si occupa della gestione delle *tabelle delle pagine*: strutture dati che associano ad ogni blocco di memoria virtuale (un segmento dello spazio di indirizzamento virtuale) un blocco di memoria fisica ed una serie di attributi necessaria alla gestione di questi blocchi. L'attributo più importante è quello di validità, che indica se l'associazione è da

considerare attendibile, in caso contrario il blocco di memoria si assume essere salvato su un disco esterno e quindi andrà riposizionato in RAM prima di potervi accedere.

Il controller della memoria può accedere alle tabelle delle pagine ed utilizzare le associazioni lì contenute per tradurre ogni indirizzo di memoria virtuale nel corrispondente indirizzo fisico.

Un classico meccanismo utilizzato per migliorare l'efficienza della memoria virtuale è il *Translation Lookaside Buffer*, implementato nel sistema e descritto in seguito.

TLB

Poiché ogni accesso alla memoria, con la traduzione degli indirizzi attiva, necessita che l'elemento della tabella delle pagine relativo al frame acceduto sia caricato nel coprocessore, ogni accesso a zone diverse di memoria richiede la ricerca delle relative associazioni tra indirizzi logici e fisici, processo computazionalmente costoso. Per esempio, si consideri un'istruzione che si trovi al penultimo indirizzo di un frame (quindi il cui indirizzo abbia il valore `0xFF8` nei 12 bit meno significativi) che avvii un caricamento dalla sezione `.data`: all'inizio del ciclo di esecuzione lo stadio di fetch del processore carica l'istruzione all'inizio del frame successivo di memoria, questo richiede la ricerca dell'elemento della page table che descrive il frame ed il caricamento nel coprocessore del suddetto elemento, successivamente il processore esegue l'istruzione corrente, che richiede l'accesso alla sezione `.data` che si trova a svariati frame di distanza dal codice, quindi bisogna cercare di nuovo il descrittore nella tabella delle pagine; al ciclo successivo si dovrà accedere di nuovo al frame che contiene il codice, quindi si deve effettuare un'ulteriore ricerca per ottenere un valore recuperato poco prima.

L'esempio appena descritto rappresenta un caso pessimo di esecuzione che, tuttavia, non è troppo sporadico; di conseguenza è stato aggiunto un meccanismo che consente di memorizzare alcuni elementi della tabella delle pagine in una memoria temporanea ad accesso veloce: il Translation Lookaside Buffer. Questo dispositivo, di dimensione variabile, permette di salvare parte degli elementi delle page table (generalmente si tende a salvare gli elementi più recenti, ma è possibile realizzare schemi più complessi ed efficienti) evitando di eseguire spesso funzioni di ricerca computazionalmente costose.

Il controller della memoria utilizza il TLB in maniera automatica: se l'elemento della tabella delle pagine è presente nel buffer, viene caricato nel coprocessore per effettuare l'accesso; in caso contrario viene sollevata un'eccezione. Il BIOS fornito (cfr. sec. 2.3.1) si occupa di gestire l'eccezione generata dalla mancanza dell'elemento necessario nel TLB, effettuando la ricerca dell'elemento nella tabella delle pagine attiva e segnalando attraverso un'ulteriore eccezione la mancanza dell'elemento cercato.

2.3 BIOS e Libreria di Sistema

Il processore ARM7TDMI ha un funzionamento abbastanza rudimentale per quanto riguarda l'avvio dell'esecuzione e la gestione delle eccezioni, inoltre alcune funzionalità della macchina e l'accesso diretto al contenuto dei registri del processore sono possibili soltanto attraverso specifiche istruzioni assembly. È stato quindi realizzato un firmware di sistema (BIOS) che si occupa dell'avvio e della gestione delle eccezioni di basso livello, il cui codice viene collocato nella memoria ROM (cfr. sec. 2.2.3). Questo è inoltre in grado di fornire alcuni servizi come il salvataggio e il caricamento dello stato del processore, lo spegnimento della macchina e la notifica di errori gravi non risolvibili.

Oltre al BIOS viene fornita una libreria sviluppata in codice Assembly che permette l'accesso ai servizi forniti dal firmware, ad altre funzionalità hardware e ad una semplice routine di stampa sul primo terminale.

2.3.1 Servizi firmware

Il firmware di sistema implementa tre servizi di basso livello accessibili tramite la libreria di sistema:

`LDST`: carica lo stato del processore che si trova all'indirizzo indicato dal valore del primo registro del processore;

`HALT`: arresta la macchina in maniera corretta mostrando il messaggio "SYSTEM HALTED" sul primo terminale;

`PANIC`: mostra il messaggio "KERNEL PANIC" sul primo terminale e fa entrare la macchina in un ciclo infinito per segnalare un errore grave.

Il BIOS, inoltre, fornisce una serie di gestori delle eccezioni di basso livello che svolgono routine specifiche per ogni tipo di eccezione e, se necessario, passano il controllo ai gestori di livello kernel, ammesso che questi siano correttamente inizializzati. I gestori di interrupt e program trap si limitano a passare il controllo ai gestori di livello superiore; il gestore di system call/breakpoint risponde direttamente ai breakpoint che richiamano i servizi `LDST`, `HALT` e `PANIC`, altrimenti passa il controllo al gestore di livello kernel; il gestore delle eccezioni di memoria controlla la causa dell'eccezione e tenta di effettuare una ricerca sulle tabelle delle pagine per l'elemento che descrive la zona di memoria che ha generato l'eccezione, se questa ricerca fallisce o l'eccezione non era derivante da un elemento mancante nel TLB, passa il controllo al gestore di livello superiore.

Durante la sequenza di bootstrap, il BIOS imposta i propri gestori delle eccezioni in modo che vengano richiamati dal processore in maniera corretta; inoltre, i gestori di alto livello vengono inizializzati in modo da puntare alla funzione `PANIC`, così da segnalare l'errore a tempo di esecuzione ed evitare comportamenti imprevedibili; infine

recupera il punto di ingresso dall'immagine binaria del kernel caricata in memoria ed avvia l'esecuzione del codice presente in RAM.

2.3.2 Funzioni di Libreria

La libreria di sistema mette a disposizione una serie di funzioni, accessibili nel linguaggio C, per accedere a funzionalità dell'hardware e del BIOS. Grazie alle funzioni esposte dalla libreria, è possibile interagire con registri speciali del processore e del coprocessore, inviare istruzioni al controller del TLB, richiedere servizi di tipo system call o breakpoint, salvare o caricare lo stato del processore e cambiare stato alla macchina.

A queste funzionalità è affiancata una procedura di stampa sul terminale 0 ed una libreria secondaria che implementa la divisione intera, operazione altrimenti non supportata dal processore.

Capitolo 3

Interfaccia

La macchina emulata, descritta nel capitolo 2, non permette interazione diretta da parte dell'utente, se non per l'avvio della macchina e la raccolta dell'output dai device al termine dell'esecuzione. Sarebbe possibile, in linea teorica, utilizzare un debugger (ad esempio `gdb`) per esaminare il comportamento della macchina durante l'esecuzione; questa operazione, tuttavia, risulterebbe molto complessa, poiché il debugger espone tutta la struttura dell'emulatore, portando in secondo piano ciò che avviene all'interno della simulazione. Per rendere possibile l'interazione e l'analisi dell'esecuzione a run time, è stata, quindi, realizzata un'interfaccia grafica che mette a disposizione alcune viste sullo stato interno della macchina, l'accesso ai device terminale ed una serie di strumenti di debug dedicati.

Framework Qt

L'interfaccia grafica (o GUI) è stata implementata utilizzando il toolkit *Qt* [29], un potente framework open source per sviluppo di applicazioni grafiche, che possiede tre principali punti di forza:

Supporto multi-architettura nativo: il toolkit è disponibile per i principali sistemi operativi esistenti, permettendo di rendere portabile il codice in maniera semplice e quasi implicita;

Infrastruttura basata su segnali e slot: questo particolare schema di comunicazione tra oggetti permette di utilizzare in maniera intuitiva il design pattern MVC [30], separando chiaramente l'interfaccia grafica dall'emulatore e rendendo più mantenibile il codice prodotto;

Funzioni di accessibilità integrate: Qt implementa un livello di astrazione per accedere in maniera omogenea alle funzioni di accessibilità fornite dai vari sistemi operativi.

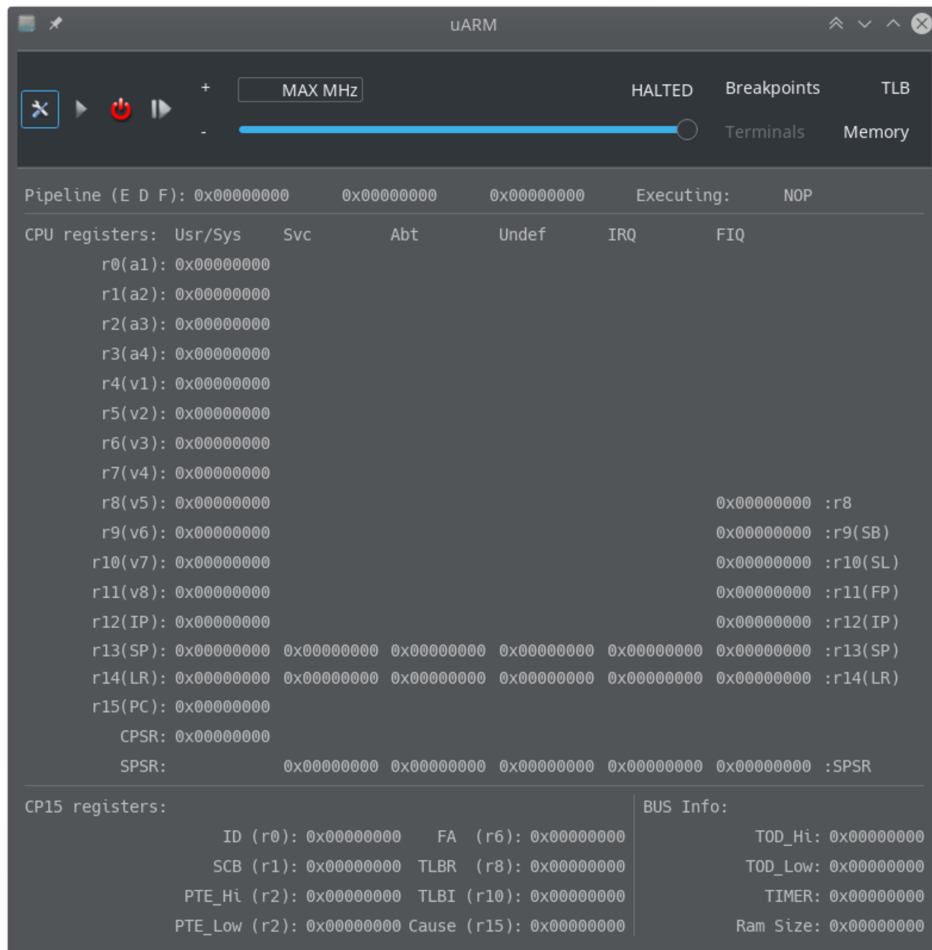


Figura 3.1: Schermata principale: Barra Principale

3.1 Interazione

Attraverso le funzioni dell'interfaccia grafica, l'utente è in grado di governare il comportamento dell'emulatore: ha il controllo dell'esecuzione, può modificare le opzioni ed accedere ai device terminale. Tutte le funzioni della GUI sono attivabili mediante la barra principale (fig. 3.1), questa permette, inoltre, di accede alle funzionalità di debug (cfr. sec. 3.2).

3.1.1 Impostazioni

Attraverso il primo pulsante dell'interfaccia (selezionato in figura 3.1)è possibile raggiungere la finestra delle configurazioni, mostrata in figura 3.2, da qui si accede alle impostazioni dell'emulatore, quali dimensione di RAM e TLB, velocità dei device, im-

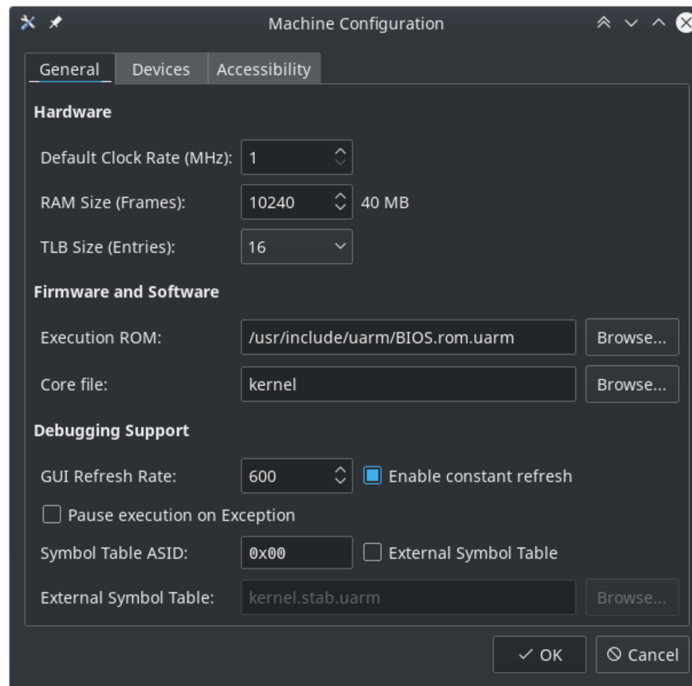


Figura 3.2: Configurazioni: pagina *General*

magine binaria del BIOS da caricare nella ROM ed immagine binaria del kernel o boot loader di alto livello da caricare in RAM. Inoltre è possibile selezionare alcune impostazioni dell'interfaccia, come l'aggiornamento continuo delle viste di debug e la tabella dei simboli utilizzata dal debugger.

La seconda pagina della finestra delle configurazioni fornisce l'accesso alle impostazioni dei device, mostrati un tipo per volta (fig. 3.3). È possibile attivare o disattivare ogni dispositivo e selezionare il file associato; in base alla classe di device, il file associato ha un diverso utilizzo: per le stampanti ed i terminali si tratta di un file di output, in cui viene scritto tutto il testo generato dal device; per quanto riguarda i nastri, i dischi e le interfacce di rete questi file puntano ai descrittori del dispositivo, nei primi due casi generati con il tool `uarm-mkdev`.

L'ultima scheda della finestra (fig. 3.4) fornisce l'accesso alle impostazioni di accessibilità, attualmente l'unica opzione presente è quella che permette l'attivazione della Modalità Accessibilità Aumentata (cfr. sec. 3.3.2).

3.1.2 Esecuzione

L'interfaccia grafica espone due modalità di esecuzione per l'emulatore: la modalità *step by step* e la modalità *continuous play*. La prima delle due esegue un singolo ciclo di clock

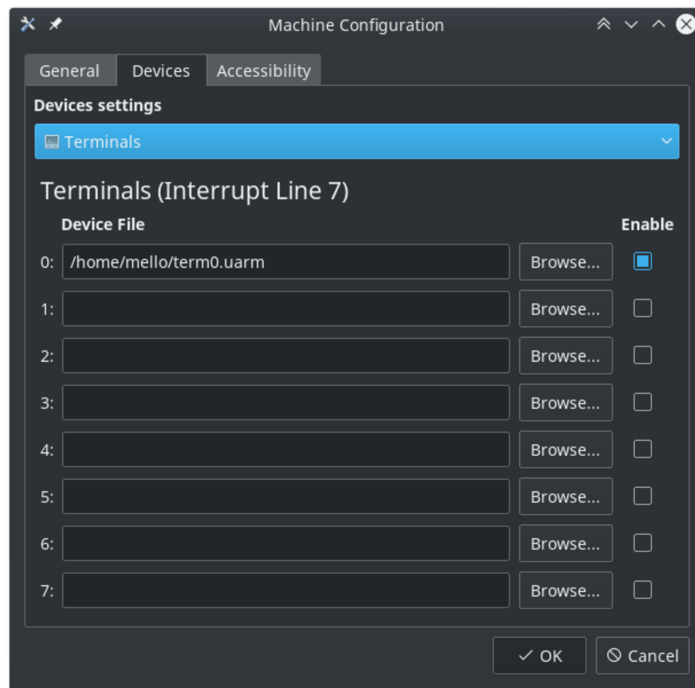


Figura 3.3: Configurazioni: pagina *Devices*

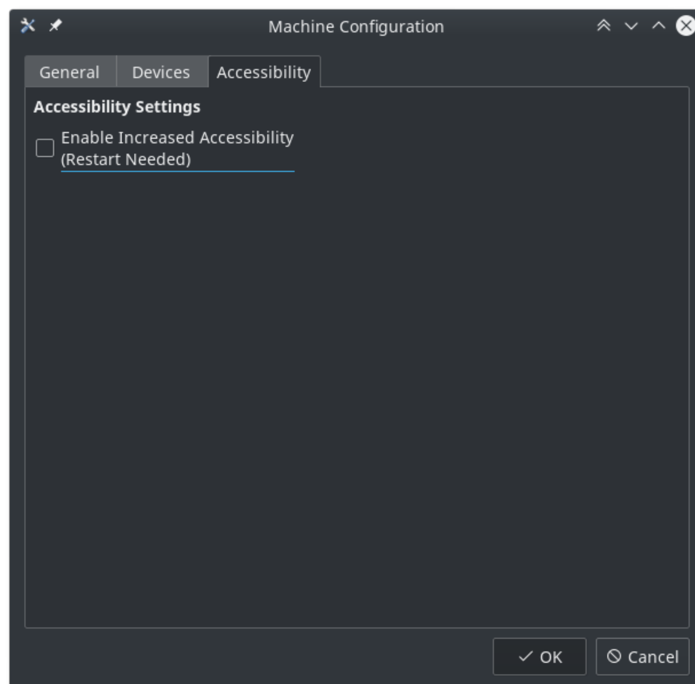


Figura 3.4: Configurazioni: pagina *Accessibility*

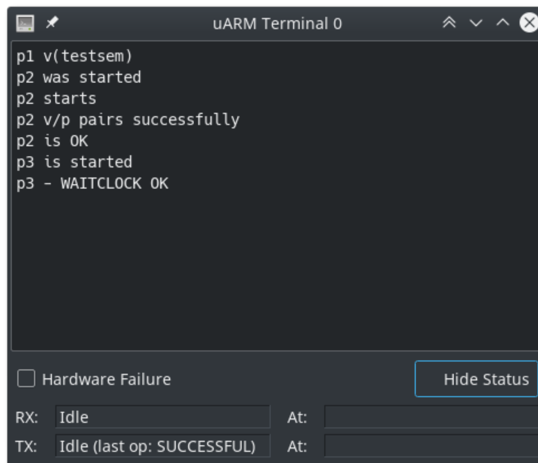


Figura 3.5: Visualizzatore terminale

ad ogni pressione del pulsante, la seconda avvia l'esecuzione continua, la cui velocità è regolata dal cursore che occupa la parte centrale della barra principale.

È inoltre necessario accendere la macchina emulata prima di avviarla, questo passo (come il riavvio, disponibile a macchina accesa) inizializza le memorie, caricando le immagini impostate dal file di configurazione, i device, dove necessario, ed il primo salto incondizionato per eseguire il codice di bootstrap del BIOS.

3.1.3 Terminali

Una volta accesa la macchina, è possibile mostrare una finestra per ogni device terminale attivo (fig. 3.5). Le finestre dei terminali mostrano lo stato interno del dispositivo e permettono l'interazione da parte dell'utente. La parte centrale della finestra mostra il testo stampato dal programma ed inserito dall'utente, è possibile scrivere caratteri utilizzando la tastiera, che verranno ricevuti dal terminale e segnalati al processore. Nella sezione inferiore sono disponibili due pulsanti, uno per simulare guasti hardware, utile per sviluppare driver con tolleranza ai guasti, ed un secondo pulsante che mostra lo stato dei due sotto-device del terminale (cfr. sec. 2.2.2) ed il valore dell'orologio di sistema (cfr. sec. 2.2.1) al momento dell'ultima operazione effettuata.

3.2 Debugger

Oltre a fornire una maniera semplice per interagire con l'emulatore, uno degli scopi principali dell'interfaccia grafica è quello di offrire supporto nella fase di sviluppo del codice attraverso funzionalità di debug, che altrimenti sarebbero eccessivamente complicate da ottenere.

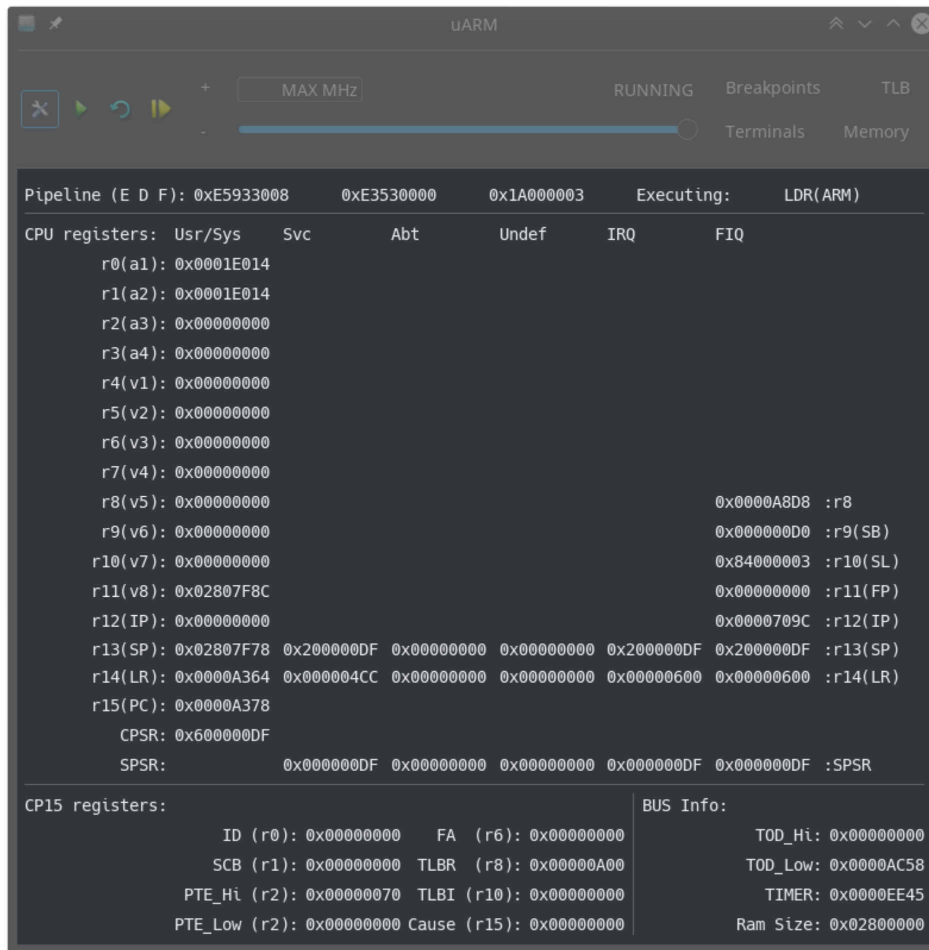


Figura 3.6: Schermata principale: Stato del Processore

Senza strumenti dedicati di debugging, sarebbe necessario utilizzare strumenti generici sull'intera macchina virtuale, il che renderebbe complesso prima di tutto distinguere il codice esecutivo dell'emulatore stesso dal codice che l'emulatore sta eseguendo, allo stesso modo la memoria interna della macchina si potrebbe confondere con la memoria assegnata all'emulatore. Proprio per evitare questo genere di ambiguità e per mantenersi in linea con la filosofia della "Completeness over Complexity" sono state implementate le funzionalità di debug descritte in seguito.

3.2.1 Stato Processore

Il primo insieme di informazioni sulla macchina è presente nella finestra principale dell'interfaccia (fig. 3.6), dove sono mostrati il contenuto della pipeline, l'intero stato del

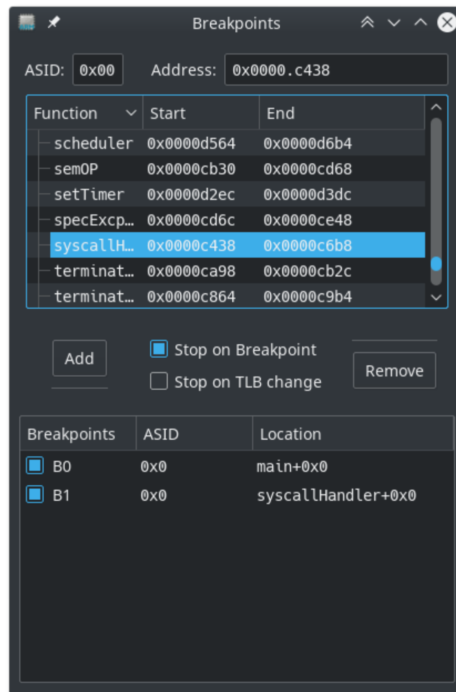


Figura 3.7: Schermata dei Breakpoint

processore ed i registri più interessanti del coprocessore di sistema, oltre ai registri speciali che descrivono lo stato dell'orologio di sistema, dell'interval timer e la dimensione della memoria RAM (cfr. cap. 2).

Le informazioni riportate in questa schermata sono molto utili per una fase di debug avanzata, basata sull'analisi del codice binario/Assembly; una parte dei registri mostra informazioni interessanti anche per una ricerca di errori meno complessa, poiché alcuni registri contengono valori noti in determinati momenti dell'esecuzione, come, ad esempio, i registri **a1-a4**, utilizzati per il passaggio di argomenti nella chiamata di funzione, oppure il registro **r15** e **r6** del coprocessore, contenenti rispettivamente informazioni sulla causa e la locazione in memoria di un eventuale errore.

3.2.2 Breakpoint

La possibilità di interrompere l'esecuzione in punti prestabiliti del codice è un'altra caratteristica molto desiderabile di un sistema di debug. La finestra dei *Breakpoints*, mostrata in figura 3.7, permette di creare punti di arresto in corrispondenza dell'ingresso nel corpo di funzioni o al raggiungimento indirizzi specifici.

Sfruttando i breakpoint, in congiunzione agli altri strumenti di debug, è possibile arrestare l'esecuzione in momenti interessanti e valutare lo stato del sistema, per individuare

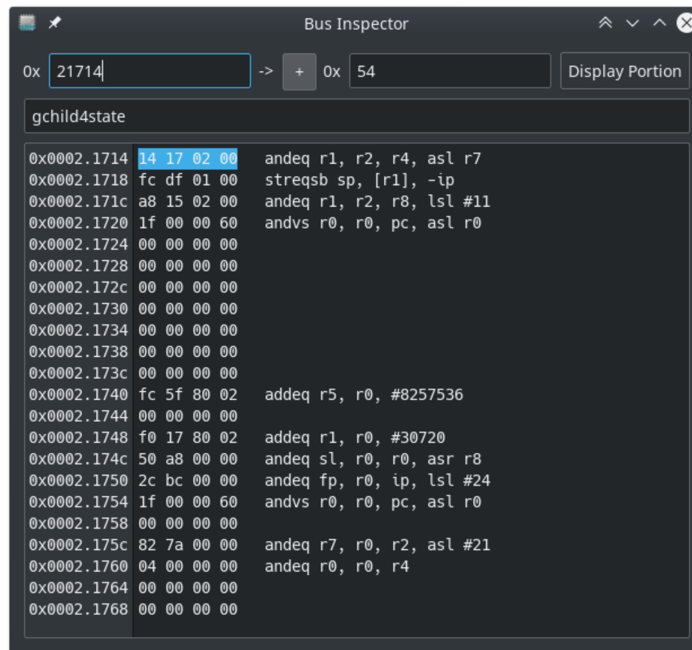


Figura 3.8: Visualizzatore del Bus

il punto di innesco di comportamenti erronei. Poiché non è presente un visualizzatore del codice sorgente, non è immediatamente possibile inserire breakpoint in posizioni arbitrarie all'interno del corpo delle funzioni, ma l'ostacolo è facilmente aggirabile dichiarando una o più funzioni nulle ed inserendo chiamate ad esse, e relativi breakpoint, nelle posizioni desiderate.

Poiché le funzioni elencate nella finestra dei Breakpoints sono desunte dalla tabella dei simboli caricata, è possibile ottenere una lista non riferita al kernel specificando una differente tabella dei simboli nelle impostazioni dell'emulatore, ad esempio nel caso in cui si desideri interrompere l'esecuzione in corrispondenza di una funzione di un processo utente caricato da nastro. Per la stessa ragione, ogni breakpoint ferma l'esecuzione soltanto se l'indirizzo a cui è associato viene acceduto con uno specifico ASID attivo, permettendo di tracciare l'esecuzione di una funzione da parte di un processo specifico (identificato dall'ASID).

3.2.3 Visualizzatore Bus

Molte informazioni interessanti sono salvate in memoria, inoltre svariati registri di controllo dei device ed alcune funzionalità di sistema non sono mostrati esplicitamente nelle schermate già descritte, di conseguenza è presente uno strumento che consente la visualizzazione di porzioni degli indirizzi del bus accessibili al processore (fig. 3.8).

Il visualizzatore del bus permette di indicare gli indirizzi di inizio e fine dell'area da visualizzare, oppure la dimensione dell'area, in base esadecimale, a partire dall'indirizzo di inizio. Una volta impostati i due campi numerici, nella parte centrale della finestra viene mostrato il contenuto dello spazio di indirizzi parola per parola (interi di 32 bit in base esadecimale) e, dove possibile, un'interpretazione in Assembly ARM delle parole di memoria visualizzate.

Spesso può tornare utile tenere sotto controllo diverse porzioni di memoria, di conseguenza è possibile avviare più di un visualizzatore simultaneamente. Per tenere traccia del significato dell'area considerata, è presente un campo di testo liberamente modificabile in cui inserire annotazioni. Il campo di testo ha, inoltre, una funzione speciale: se l'area di memoria corrisponde ad una variabile o struttura globale del programma caricato e l'area di testo contiene il nome della struttura, allora il visualizzatore aggiornerà gli indirizzi di inizio e fine al reset dell'emulatore per continuare a puntare alla struttura indicata. Questa funzione risulta comoda quando si effettuano modifiche al codice e gli indirizzi delle strutture dati subiscono variazioni dopo la compilazione, in questo caso si può continuare ad esaminare l'esecuzione ed i valori delle variabili globali senza doversi preoccupare di ricercarle manualmente in memoria.

3.2.4 Visualizzatore Strutture Statiche

In maniera del tutto similare ai breakpoint, è possibile ottenere una lista delle strutture dati (e delle variabili) globali dalla tabella dei simboli caricata. Questa lista è mostrata in una schermata dedicata (fig. 3.9), permettendo di controllare in maniera veloce l'intero contenuto di una variabile locale o di una struttura, ricercandola per nome.

Senza questa schermata, l'operazione di ricerca delle strutture dati in memoria diventa decisamente più macchinosa, richiedendo la decompilazione del programma e la ricerca degli indirizzi occupati in RAM dalla struttura, per poi inserire manualmente questi indirizzi nel visualizzatore del bus (cfr. sec. 3.2.3). Nell'ottica di agevolare questo processo e permettere la visualizzazione simultanea di più contenuti noti della memoria, il visualizzatore delle strutture statiche espone un pulsante che apre automaticamente un visualizzatore del bus precaricato con gli indirizzi ed il nome della struttura scelta, sfruttando così l'aggiornamento automatico degli indirizzi.

3.2.5 Visualizzatore TLB

Un ultimo componente hardware rimane nascosto dall'interfaccia grafica, il Translation Lookaside Buffer (cfr. sec. 2.2.3); per poterne esaminare il contenuto è stato implementato un visualizzatore dedicato, mostrato in figura 3.10, che espone ogni elemento delle page table caricato nel buffer.

Questo visualizzatore è in grado di scomporre gli elementi caricati nel TLB e mostrare i vari campi delle strutture nel pannello laterale, o in sovrimpressione al *mouse over*.

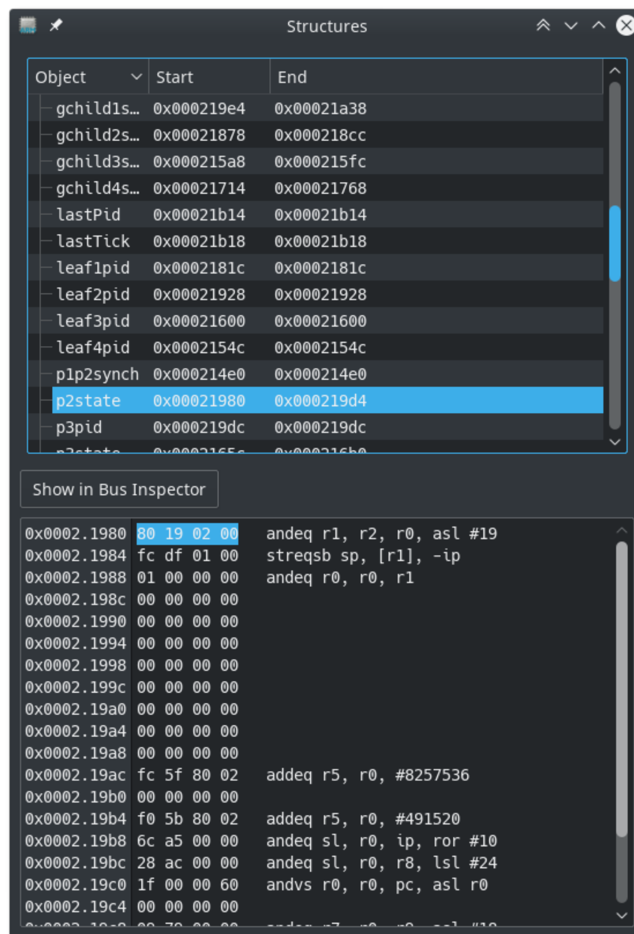


Figura 3.9: Visualizzatore Strutture Statiche

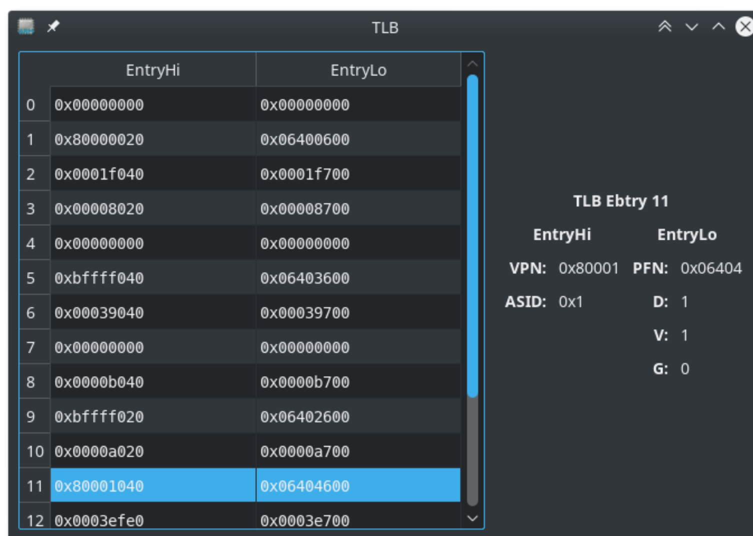


Figura 3.10: Visualizzatore TLB

Permette inoltre di modificare direttamente i valori contenuti nel buffer per effettuare esperimenti con l'indirizzamento logico della memoria.

3.3 Accessibilità

Oggi non sono ancora disponibili dei meccanismi generali per sviluppare applicazioni accessibili, nello specifico esiste una classe di cosiddette *Assistive Technologies* [31] le quali permettono di riorganizzare e presentare all'utente il contenuto semantico delle interfacce grafiche, in modo che possa essere interpretato attraverso meccanismi non completamente grafici (es. screen reader, schermi tattili, ecc..). Le tecnologie assistive non sono, però, regolate da uno standard, diventa quindi complicato sviluppare applicazioni che siano compatibili con più sistemi operativi e librerie di accessibilità, ma Qt implementa un proprio strato di astrazione che permette di scrivere codice portabile anche per quanto riguarda le assistive technologies.

La presenza di uno studente non vedente ha fortemente incentivato l'impiego di tali strumenti durante lo sviluppo dell'interfaccia grafica di μARM, la quale sfrutta le capacità di Qt per essere più accessibile in ogni sua parte.

3.3.1 Annotazioni

La maniera concettualmente più corretta di rendere *accessibile* un'applicazione grafica è quella di aggiungere il supporto per le tecnologie assistive senza modificare l'aspetto estetico. Nello specifico, sviluppando applicazioni Qt, questo significa inserire descrizioni

testuali ulteriori per ogni componente sensibile dell'interfaccia. I due campi di testo in questione si chiamano *Accessible Name* ed *Accessible Description* e rappresentano, appunto, un nome esplicativo del componente grafico ed una sua descrizione più dettagliata per chiarificarne il significato e l'utilizzo.

Sarà poi lo strumento assistivo a riorganizzare l'accesso all'intera interfaccia in modo da fornirne una visione semantica. In questo modo, i contenuti saranno ordinati in una maniera più simile alla struttura gerarchica con cui sono definiti nel codice sorgente, permettendo di "saltare" più semplicemente da una zona logica della schermata all'altra.

3.3.2 Modalità Accessibilità Aumentata

Purtroppo non è sufficiente aggiungere le annotazioni di accessibilità per rendere anche *usabile* tutta l'interfaccia tramite tecnologie assistive. Il problema principale è dato dalla visualizzazione dello stato del processore (cfr. sec. 3.2.1), i cui contenuti risultano particolarmente confusionari quando acceduti attraverso tecnologie assistive.

Per risolvere questo problema, la soluzione più adeguata è risultata essere l'aggiunta di una modalità di rendering della parte principale dell'interfaccia, denominata *Increased Accessibility* (accessibilità aumentata). I registri delle varie modalità di esecuzione sono raggruppati in svariate aree di testo, una per modalità di esecuzione. Le aree di testo risultano essere una delle tipologie di widget più semplici da navigare con strumenti assistivi.

La revisione accessibile ha avuto un buon riscontro da parte dello studente non vedente, che ha offerto un valido supporto per migliorare l'usabilità generale dell'interfaccia.

Capitolo 4

Progetto

Il presente lavoro propone, oltre allo strumento descritto nei Capitoli 2 e 3, una serie di esercizi da assegnare ad una classe di studenti di un Corso di Laurea in Informatica. L'insieme dei compiti prende il nome di *Progetto JaeOS*, acronimo di "Just Another Educational Operating System", ed appartiene alla famiglia di progetti prodotti nell'ambito di *Virtual Square* [3, 5].

Il progetto didattico nasce per essere svolto in piccoli gruppi ed è diviso in tre fasi principali, ognuna con obbiettivi specifici, corredate da una documentazione che ne dichiara i requisiti ed alcuni file sorgente per verificare il corretto funzionamento del codice prodotto. In base al livello dell'insegnamento, il progetto può essere assegnato in maniera sequenziale durante lo svolgimento dell'anno accademico, oppure è possibile somministrare una versione ridotta del progetto in modo da bilanciare il carico di lavoro. Tale versione ridotta può essere composta dalle prime due fasi, da sviluppare per intero, oppure dal più complesso compito di realizzare la terza fase a partire dalle precedenti due già implementate. I documenti di specifiche del progetto sono allegati in versione integrale in appendice a questo documento.

Le tre fasi descritte nei documenti di specifica rappresentano tre livelli di un sistema operativo, derivato dal *THE Multiprogramming System* [32], composto da una serie di strati che hanno rispettivamente accesso alle funzioni offerte dallo strato inferiore e le utilizzano per fornire servizi a quello superiore. In questo schema a livelli, il livello 0 è rappresentato dall'hardware emulato, che espone le proprie funzionalità al livello 1, ovvero il BIOS di sistema, cui segue il livello 2 (la *fase 1* del progetto) che realizza il gestore delle code di strutture utilizzate dal livello superiore (livello 3 - *fase 2*), quest'ultimo sarà il core del sistema operativo vero e proprio, a cui segue il livello di supporto (livello 4 - *fase 3*, l'ultima descritta dai documenti di specifiche) che offre funzionalità necessarie perché generici processi a livello utente possano essere eseguiti. Oltre questi cinque livelli, è possibile creare altri strati di supporto e realizzare funzionalità più complesse, quali, ad esempio, file system, stack di rete, una shell interattiva, etc.

Verranno in seguito discusse le tre fasi proposte dai documenti di specifica.

4.1 JaeOS Phase 1

La prima fase è un compito abbastanza semplice ed ha come scopo quello di far prendere confidenza agli studenti con l'ambiente di lavoro (i.e. μARM) e con il linguaggio di programmazione da utilizzare, nel caso questo non sia ancora stato trattato nel corso di altri insegnamenti[1].

I requisiti di questo primo esercizio si limitano alla realizzazione di due piccole librerie per la gestione di strutture dati, utilizzate nella fase successiva. Viene richiesto di gestire liste circolari ed alberi di Process Control Blocks (cfr. sec. 4.2.1) ed una lista globale di Semaphore Descriptors (cfr. sec. 4.2.2), oltre all'allocazione di questi due tipi di strutture dati.

I tipi di dato appena dichiarati non hanno nessun significato particolare a questo punto dell'esercizio, sono semplici strutture di cui bisogna tenere traccia e non hanno nessun collegamento con l'architettura sottostante; difatti questa fase potrebbe essere sviluppata e testata interamente sulla macchina fisica dello studente, si richiede, però, di utilizzare l'emulatore μARM per poter illustrare le funzionalità di debug e permettere agli studenti di prendere confidenza con l'ambiente di esecuzione/testing, fondamentale per le fasi successive.

Conseguentemente alla consegna, i lavori prodotti vengono corretti in modo da fornire feedback agli studenti sui propri elaborati e poter quindi affrontare in maniera più conscia la fase successiva.

4.2 JaeOS Phase 2

La seconda parte del progetto è quella che getta le basi di un vero e proprio sistema operativo. Durante lo svolgimento di questo compito, gli studenti dovranno sviluppare una serie di funzionalità che permettano al sistema di gestire l'esecuzione concorrente di più processi, realizzando un semplice scheduler, di effettuare operazioni di input/output e richiamare servizi di sistema sotto forma di system call.

Questa fase richiede un più profondo livello di comprensione del funzionamento dell'hardware e dei sistemi operativi per poter essere svolta correttamente. Il documento di specifica propone una serie di semplici algoritmi per realizzare le varie funzionalità, poi testate dall'unità di verifica, lasciando agli studenti la libertà di implementarne varianti ottimizzate. Viene inoltre richiesto di intraprendere alcune scelte implementative cardine, tra cui, per esempio, il metodo di gestione dei tempi di esecuzione dei processi o l'organizzazione interna delle strutture dati.

[1] Il linguaggio consigliato per svolgere l'esercizio è il C [33], ma può essere utilizzato qualsiasi linguaggio per cui esista un compilatore in grado di generare file binari eseguibili dal processore ARM7TDMI, a patto di utilizzare, in fase di linking, la mappa della memoria fornita e, possibilmente, implementare meccanismi per accedere alle funzioni di libreria.

4.2.1 Gestione Processi

Questa parte del progetto richiede la realizzazione del core di un sistema operativo multiprocesso, con uno schema a condivisione del tempo, è dunque necessario implementare un algoritmo di scheduling dei processi attivi. Viene proposto di realizzare un semplice scheduler round-robin in modo da garantire ad ogni processo in attesa l'opportunità di prendere il controllo del processore. Questa proprietà è intrinsecamente garantita dalla semplice struttura dell'algoritmo di selezione round-robin, per cui tutti gli elementi della lista di interesse vengono selezionati in un ordine prestabilito, ottenendo tutti il medesimo quantitativo di risorse (in questo caso la risorsa in oggetto è il tempo di esecuzione).

Per gestire l'avvio e la terminazione dei processi, il nucleo deve esporre appositi servizi di sistema, inoltre ad ogni processo viene associato un identificatore univoco generato secondo un algoritmo a scelta degli studenti.

Infine si richiede di implementare un semplice algoritmo di deadlock detection, basato sulla differenza tra semafori (cfr. sec. 4.2.2) "Soft Blocking" e "Hard Blocking". I primi sono semafori associati ai device, per i quali si presuppone che, prima o poi, le risorse richieste vengano liberate a causa della terminazione di un comando su dispositivo; i semafori del secondo tipo sono invece quelli non direttamente controllati dal sistema operativo e che permettono la sincronizzazione inter-processo. Una volta effettuata questa distinzione, risulta chiaro che, se nessun processo è pronto ad eseguire, nessun processo sta aspettando il termine di un'operazione di input/output ed uno o più processi sono in attesa su un semaforo "Hard Blocking", allora quei processi saranno in mutua attesa e si può dichiarare che il sistema è in stato di deadlock.

4.2.2 Active Semaphore List

Per l'interazione tra i processi e con i dispositivi esterni è necessario realizzare un meccanismo di sincronizzazione, viene quindi proposto uno schema di semafori pesati di basso livello.

Il documento di specifica richiede la realizzazione di un modulo, denominato *Active Semaphore List*, per la gestione dei semafori attivi; questi saranno identificati da un intero e conservati in una lista ordinata, in modo da semplificare l'ottimizzazione dell'algoritmo di ricerca. L'intero che definisce ogni semaforo ha doppia valenza: il valore vero e proprio dell'intero rispecchia lo stato interno del semaforo (i.e. il numero di risorse attualmente disponibili/richieste associate a quel semaforo), il puntatore all'intero viene utilizzato come identificativo univoco per fare riferimento ad ogni semaforo.

Oltre ai semafori gestiti dall'Active Sempahore List (abbreviato in ASL), vengono definiti anche una serie di semafori di sistema associati ai device esterni, questi semafori saranno utili per gestire le richieste di input/output da parte dei processi.

4.2.3 Interrupt Handler

Il nucleo del sistema è il componente che si occupa di gestire la comunicazione a più basso livello con i dispositivi emulati collegati a μARM. La comunicazione tra i device ed il processore avviene attraverso due canali: il bus di sistema, attraverso cui passano i flussi di dati, e le linee di interrupt, che i dispositivi utilizzano per richiamare l'attenzione del processore all'avvenire di un cambiamento di stato. La gestione dell'interazione a basso livello con i dispositivi dovrà, quindi, essere implementata in due parti diverse del core: la gestione delle richieste ai device sarà un servizio offerto dal sistema, dunque viene raggruppata con le chiamate di sistema disponibili (cfr. sec. 4.2.4), la gestione delle risposte deve invece avvenire in conseguenza alle interruzioni dell'esecuzione da parte del dispositivo, è quindi inclusa nel gestore degli interrupt.

Un caso particolare è quello dell'interval timer: questo speciale dispositivo è l'unica fonte di temporizzazione disponibile nel sistema, quindi il gestore degli interrupt dovrà utilizzarlo simultaneamente per svolgere due diversi lavori di sincronizzazione: lo scheduling dei processi ed il servizio di attesa. La soluzione a questo problema è uno dei compiti più interessanti lasciato agli studenti.

4.2.4 Low Level System Calls

Come accennato in precedenza, questo livello del sistema operativo deve essere in grado di fornire alcuni servizi di basso livello ai processi privilegiati che li richiedano. La documentazione dichiara la lista di funzioni richiamabili, i loro parametri ed il risultato atteso, lasciando la scelta della maggior parte dei dettagli implementativi come compito per gli studenti.

Poiché le funzionalità offerte da questo livello permettono la creazione e la terminazione di processi con permessi arbitrari, oltre all'accesso diretto alla memoria fisica del sistema ed ai device, le chiamate di sistema devono essere accessibili soltanto a processi che eseguano con privilegi massimi (i.e. System/Kernel Mode). I livelli superiori del sistema forniranno servizi con capacità più limitate richiamabili anche da processi senza privilegi, questi servizi fungeranno in parte da *wrapper* per le chiamate di basso livello, in modo tale da renderle più "sicure".

4.3 JaeOS Phase 3

Il livello 4 del sistema operativo è il "livello di supporto", quello che offre le prime funzionalità utilizzabili da processi generici a livello utente, che possono, quindi, essere caricati da dispositivi esterni ed eseguiti nel sistema.

Per permettere l'esecuzione di programmi arbitrari sul sistema è necessario implementare un primo meccanismo di protezione della memoria, poiché un programma esterno potrebbe, intenzionalmente o per errore, modificare aree di memoria contenenti codice

binario, o dati, appartenenti al sistema operativo o ad un altro processo, minandone la stabilità. Viene quindi richiesto di utilizzare lo schema di indirizzamento virtuale della memoria di μARM.

Questo livello inoltre implementa una serie di chiamate di sistema specifiche per interagire con i dispositivi. Differentemente dal livello inferiore, che offriva una sola chiamata di sistema tramite cui era possibile inviare comandi arbitrari ai dispositivi, in questo strato vengono definite una serie di system call, una per tipo di operazione e device che mascherano i dettagli della comunicazione con il dispositivo stesso (i.e. semplici device driver).

Utilizzando il segmento di memoria virtuale condiviso e le operazioni sui semafori introdotti da questo livello, è possibile far cooperare più processi utente in esecuzione simultanea.

4.3.1 Memoria Virtuale

Una volta attivata la traduzione degli indirizzi virtuali, ogni indirizzo di memoria (logico) richiesto da un processo viene tradotto in un indirizzo fisico assegnato dal gestore della memoria. Il gestore della memoria può, quindi, associare indirizzi fisici diversi allo stesso indirizzo logico, basandosi su un identificatore associato ad ogni processo. In questo modo, il meccanismo dell'indirizzamento logico della memoria garantisce, in maniera implicita, la protezione dello spazio di indirizzamento privato di ogni processo.

Senza questo meccanismo, inoltre, per caricare in memoria ed eseguire un programma esterno al sistema operativo, sarebbe necessario conoscere, a tempo di esecuzione, le aree di memoria sicuramente libere e che non verranno utilizzate, che possano, dunque, essere occupate dal codice e dai dati del programma. Ne segue che, per eseguire più di un programma esterno la situazione si complica non poco. Utilizzando uno schema di indirizzamento virtuale della memoria, invece, tutti i programmi utente possono condividere le stesse zone di memoria virtuale, rendendo la compilazione ed il caricamento di programmi da eseguire sul sistema estremamente più semplice.

Un'ultima caratteristica interessante dell'indirizzamento virtuale è che, a patto di avere il supporto hardware adeguato da parte del controller della memoria, è possibile implementare un algoritmo di *paging* che consenta il salvataggio ed il caricamento di interi frame di memoria su/da disco. Questo schema, generalmente detto *swapping*, permette di estendere la memoria disponibile, salvando temporaneamente le aree di memoria inutilizzate su disco per lasciare spazio ai processi in esecuzione e ripristinandole quando necessario, il tutto a patto di un costo computazionale più alto nelle fasi di salvataggio e ripristino dei blocchi di memoria.

Il controller della memoria di μARM è dotato di un Translation Lookaside Buffer (cfr. sec. 2.2.3) per rendere più efficiente la risoluzione degli indirizzi virtuali, inoltre il coprocessore di sistema ne estende le funzionalità, permettendo di adottare meccanismi di swapping con algoritmi arbitrari. Si richiede, dunque, agli studenti di realizzare tutte

le funzioni di supporto necessarie al corretto funzionamento della memoria virtuale con un semplice algoritmo di paging *first in-first out*.

4.3.2 Gestione Dischi

I dischi sono risorse che possono essere attivamente sfruttate con l'introduzione di questo livello del sistema operativo. Il primo utilizzo, non dipendente dall'utente, è quello di memoria ausiliaria per il meccanismo di swapping (cfr. sec. 4.3.1): in questo caso un disco intero viene riservato dal sistema operativo in modo da preservare i contenuti della memoria temporaneamente spostati sul disco. La seconda possibilità di impiego è quella di dispositivo di storage e scambio di dati per i processi utente.

Implementando le due chiamate di sistema dedicate, gli studenti possono confrontarsi con i metodi di accesso a basso livello alla struttura fisica degli hard disk; saranno gli stessi studenti a dover creare dischi fissi con geometrie specifiche utilizzando il tool `uarm-mkdev`.

4.3.3 Caricamento ed Esecuzione Dinamici

Questo strato del sistema operativo aggiunge la possibilità di eseguire programmi utente arbitrari, caricati a run time da supporti esterni (i.e. nastri, cfr. sec. 2.2.2). Grazie a questa funzionalità, è possibile avviare contemporaneamente più programmi, che eseguano concorrentemente e, potenzialmente, collaborino.

È tuttavia necessario effettuare una serie di operazioni di preparazione del sistema, molto simili a quelle attuate da un moderno sistema operativo, per avviare un programma, indipendentemente dal fatto che questo si trovi su un dispositivo esterno oppure sia salvato sul file system locale.

4.3.4 Temporizzazione

Un'utile funzionalità inclusa in questa fase fornisce servizi di temporizzazione avanzata. Viene richiesto di realizzare un processo speciale che deve eseguire concorrentemente ai processi utente e che gestisce la temporizzazione di questi ultimi.

Una chiamata di sistema permette ad un generico processo di richiedere l'attesa di un numero arbitrario di secondi, questo processo dovrà essere spostato in una speciale zona del sistema e tenuto in attesa fino a che il tempo richiesto non sarà trascorso. Il processo che si occupa della temporizzazione (*delay daemon*), controllerà periodicamente il tempo trascorso in attesa per i processi che hanno richiesto questa funzione e sbloccherà quelli pronti a ripartire.

Questo particolare processo è un esempio di demone di sistema, un componente tipico dei moderni sistemi operativi.

4.3.5 High (User) Level System Calls

Le system call fornite da questo livello sono le prime utilizzabili dai processi a livello utente. L'insieme di chiamate è logicamente diviso in due gruppi: accesso ai dispositivi e controllo dei processi.

Al primo gruppo appartengono tutte le funzioni che permettono di leggere e scrivere dati da/su terminali, stampanti e dischi. Per semplicità si assume che ad ogni processo utente siano assegnati un lettore di nastri (da cui viene letto il programma da eseguire), un terminale ed una stampante, tutti con il medesimo numero di device, quindi l'accesso a terminali e stampanti fa implicitamente riferimento alla coppia assegnata al processo. L'accesso ai dischi invece è libero, a patto di evitare il disco 0, riservato per il meccanismo di swapping (cfr. sec. 4.3.1 e 4.3.2).

Le system call di controllo dei processi danno accesso ad un sistema di semafori "virtuali", alla funzione di terminazione del processo corrente ed al sistema di temporizzazione, permettendo di richiedere il valore dell'orologio di sistema e di sospendere il processo corrente per un tempo specifico (cfr. sec. 4.3.4). I semafori di questo livello vengono chiamati "virtuali" poiché l'indirizzo in memoria dell'intero che li identifica è un indirizzo logico (virtuale), essendo sempre attivo lo schema di indirizzamento virtuale per i processi utente. Lo scopo di questi semafori è quello di permettere schemi di sincronizzazione inter-processo, è quindi proibito effettuare operazioni su semafori che si trovino in un segmento non globale della memoria, poiché si tratterebbe di semafori accessibili unicamente al processo correntemente in esecuzione. Se il processo corrente effettuasse un'operazione bloccante su un semaforo risiedente nel proprio segmento privato di memoria, nessun altro sarebbe in grado di sbloccarlo in un secondo momento, portando il sistema in stato di deadlock.

Capitolo 5

Conclusioni e Futuri Sviluppi

L'emulatore μARM ed il progetto JaeOS, presentati in questa trattazione, rappresentano l'ultima generazione di una prolifica stirpe di strumenti per la didattica dei Sistemi Operativi [4–6] tutti nati nell'ambito del progetto Virtual Square [3], e a loro volta discendenti della storica coppia CHIP/HOCA [24, 34].

L'impostazione del progetto proposto, partendo dalle basi del linguaggio di programmazione e crescendo fino alla costruzione di un sistema operativo completo, permette agli studenti di prendere confidenza con linguaggi di medio/basso livello ed apprendere le basi del funzionamento dei calcolatori e dei sistemi operativi, a partire dall'interazione tra software e hardware e arrivando alla gestione dei processi e della memoria. Poiché il compito si basa su un lavoro pratico da svolgere in piccoli gruppi, l'esperienza ottenuta tramite lo svolgimento di questo esercizio è molto più profonda rispetto a quella maturabile dal solo studio teorico della materia.

Negli anni passati, gli strumenti del gruppo Virtual Square sono stati ampiamente utilizzati nei corsi di Sistemi Operativi dell'Università di Bologna e della Xavier University, μARM è già in uso all'Università di Bologna da due anni e alla Xavier University da uno, mentre JaeOS è stato introdotto l'anno accademico corrente. La coppia μARM/JaeOS è dunque già utilizzata ed impiegabile nel corso di insegnamenti di Sistemi Operativi a livello universitario, ma l'emulatore è adattabile anche ad altre attività pratiche in ambito didattico, come, per esempio, corsi riguardanti le reti o lo sviluppo su piattaforme embedded basate su architettura ARM.

Sono tuttavia possibili svariate modifiche ed estensioni al materiale qui presentato, alcune proposte sono presentate di seguito, raggruppate seguendo la divisione proposta in questo documento.

Hardware

La macchina emulata può essere migliorata in svariati modi, il primo upgrade importante potrebbe essere l'aggiunta di più processori, creando una macchina multicore. In questo caso sarebbe necessario rivedere in parte anche l'interfaccia grafica, oltre ad implementare meccanismi hardware di comunicazione inter-processore.

Sarebbe possibile estendere le ISA in modo da supportare versioni più aggiornate e, quindi, implementare effettivamente un altro modello di processore. Su questa linea di azione, è anche possibile realizzare svariati modelli differenti di processori, selezionabili prima dell'avvio della macchina.

Alcuni ulteriori aggiornamenti potrebbero essere rivolti ai dispositivi: il lettore di nastri può essere sostituito da un lettore ottico, si possono aggiungere bus esterni sulla falsa riga dell'*USB* oppure, più semplicemente, delle memorie esterne di tipo flash ad accesso diretto.

Interfaccia

Come anticipato precedentemente, nel caso dell'implementazione di una versione multi processore, l'interfaccia necessiterebbe una ristrutturazione grafica, in modo da poter mostrare il contenuto dei registri dei vari core attivi. Inoltre, l'implementazione di diversi modelli di processori potrebbe implicare un diverso schema di registri, per cui sarebbe necessario rivisitare ulteriormente il visualizzatore dei contenuti del processore.

Per quanto riguarda le funzionalità di debug, implementare il meccanismo dei *data breakpoint* sarebbe una buona scelta, in quanto si tratta di uno strumento ampiamente presente nei debugger generici e piuttosto utile in fase di sviluppo.

Un secondo componente desiderabile, anche se non del tutto plausibile per un'unità di debug hardware, è il *code browser*.

Un ultimo punto interessante, riguardante l'interfaccia grafica, è la possibilità di rendere modificabile ogni valore mostrato dai vari visualizzatori, in modo da poter effettuare correzioni ed esperimenti "al volo".

Progetto

Per quanto riguarda le attività da assegnare agli studenti, vengono avanzate qui alcune proposte, possibilmente già ipotizzate, che tentano di preservare le caratteristiche dell'esperienza dello sviluppo di un sistema operativo partendo da zero.

Una prima estensione è un set di fasi ulteriori da sviluppare; aggiungere strati del sistema operativo rende il progetto decisamente più lungo e complesso, quindi più adatto a corsi avanzati. Come già ipotizzato in relazione a Kaya [5] (il predecessore di JaeOS), è possibile aggiungere una quarta fase che preveda lo sviluppo di un piccolo stack di

rete per comunicare con la macchina fisica o reti esterne, oppure l'implementazione di un semplice file system, per concludere il tutto con una quinta fase che prevede la costruzione di una shell interattiva.

Una possibilità per adattare un progetto così vasto ad un corso non avanzato è quella di distribuire il progetto tra più insegnamenti nell'ambito del corso di studi. Ad esempio, si potrebbe suddividere il progetto complessivo come segue:

Phase 1 - Corso di Programmazione di base, in cui si utilizza il progetto per far prendere confidenza agli studenti con il linguaggio di programmazione, e non è necessario utilizzare μARM;

Phase 2 - Corso di Architettura dei Calcolatori, si può variare il documento di specifiche per richiedere ulteriore interazione con l'hardware in questa parte e posticipare lo scheduling alla fase successiva;

Phase 3 - Corso di Sistemi Operativi, realizzazione di schemi di memoria virtuale, scheduler ed accesso ai dispositivi attraverso system call;

Phase 4 (stack di rete) - Corso di Reti di Calcolatori, in questo caso si può anche valutare l'idea di realizzare uno stack di rete completo almeno fino al livello di trasporto;

Phase 4 e Phase 5 (file system e shell interattiva) - Corso avanzato di Sistemi Operativi, conclude l'opera realizzando un sistema operativo completo ed utilizzabile.

Una variante adatta a corsi avanzati potrebbe essere la consegna dei soli documenti di specifiche, sprovvisti di unità di test, aggiungendo queste ultime ai requisiti implementativi. In questo caso, per aiutare gli studenti, sarebbe possibile fornire alcuni programmi di esempio certamente funzionanti su un sistema completo che rispetti le specifiche fornite, al posto delle unità di test.

Bibliografia

[1] Seymour Papert and Idit Harel. Situating constructionism. *Constructionism*, 36:1–11, 1991.

[2] Marco Melletti, Michael Goldweber, and Renzo Davoli. The jaeos project and the μarm emulator. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 3–8. ACM, 2015.

[3] Renzo Davoli and Michael Goldweber. Virtual Square (v2) in computer science education. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 301–305, New York, NY, USA, 2005. ACM.

[4] Mauro Morsiani and Renzo Davoli. Learning operating systems structure and implementation through the MPS computer system simulator. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, pages 63–67, New York, NY, USA, 1999. ACM.

[5] Michael Goldweber, Renzo Davoli, and Mauro Morsiani. The Kaya OS project and the μMPS hardware emulator. In *ACM SIGCSE Bulletin*, volume 37, pages 49–53. ACM, 2005.

[6] Michael Goldweber, Renzo Davoli, and Tomislav Jonjic. Supporting operating systems projects using the μMPS2 hardware simulator. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 63–68, New York, NY, USA, 2012. ACM.

[7] Andrew S Woodhull and Andrew S Tanenbaum. Operating systems design and implementation, 1997.

[8] M Kifer and S Smolkaka. OSP 2.

[9] Ben Pfaff, Anthony Romano, and Godmar Back. The PintOS instructional operating system kernel. In *ACM SIGCSE Bulletin*, volume 41, pages 453–457. ACM, 2009.

[10] Haifeng Liu, Xianglan Chen, and Yuchang Gong. BabyOS: a fresh start. *ACM SIGCSE Bulletin*, 39(1):566–570, 2007.

[11] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple Unix-like teaching operating system, 2011.

[12] George Fankhauser, Christian Conrad, Eckart Zitzler, and Bernhard Plattner. Topsy–a teachable operating system. *Computer Engineering and Networks Laboratory, ETH Zürich, Switzerland*, 2000.

[13] David A Holland, Ada T Lim, and Margo I Seltzer. A new instructional operating system. In *ACM SIGCSE Bulletin*, volume 34, pages 111–115. ACM, 2002.

[14] Yung-Pin Cheng and JM-C Lin. Awk-Linux: A lightweight operating systems courseware. *Education, IEEE Transactions on*, 51(4):461–467, 2008.

[15] David Hovemeyer, Jeffrey K Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with GeekOS. In *ACM SIGCSE Bulletin*, volume 36, pages 315–319. ACM, 2004.

[16] Wayne A Christopher, Steven J Procter, and Thomas E Anderson. The Nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 4–4. USENIX Association, 1993.

[17] Benjamin Atkin and Emin Gün Sirer. PortOS: an educational operating system for the post-pc environment. In *ACM SIGCSE Bulletin*, volume 34, pages 116–120. ACM, 2002.

[18] Noam Nisan and Shimon Schocken. *The elements of computing systems: Building a modern computer from first principles.* MIT press, 2005.

[19] Dave Jaggar. *ARM architecture reference manual.* Prentice Hall, 1996.

[20] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide.* John Wiley & Sons, 2013.

[21] ARM7TDMI ARM. Technical reference manual, 2001.

[22] Michael J Flynn. Computer architecture pipelined and parallel processor design, 1995.

[23] Imran Nazar. Arm7 and arm9 opcode map, w/ thumb. `http://imrannazar.com/ARM-Opcode-Map`.

[24] O Babaoglu, M Bussan, R Drummond, and F Schneider. Documentation for the CHIP computer system, 1988.

[25] AF Harvey and DAD Staff. Dma fundamentals on various pc platforms. *National Instruments, April*, 8, 1991.

[26] Coded Character Set. 7-bit american standard code for information interchange. *ANSI X3*, 4, 1986.

[27] Renzo Davoli. VDE: Virtual distributed ethernet. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of NeTworks and COMmunities*, TRIDENTCOM '05, pages 213–220, Washington, DC, USA, 2005. IEEE Computer Society.

[28] James Lyle Peterson and Abraham Silberschatz. *Operating system concepts*, volume 2. Addison-Wesley Reading, MA, 1985.

[29] AS Troll Tech. The QT toolkit.

[30] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[31] Thomas W King. *Assistive technology: Essential human factors.* Pearson, 1999.

[32] Edsger W Dijkstra. The structure of the "the" multiprogramming system. In *Classic operating systems*, pages 223–236. Springer, 2001.

[33] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[34] Ozalp Babaoglu and Fred B Schneider. The HOCA operating system specifications. Technical report, Cornell University, 1983.

Appendices

Appendice A

Emulated Hardware Manual

A.1 Introduction

μARM is an emulator program that implements a complete system with an emulated version of the ARM7TDMI processor as its core component. The processor's specifications have been respected, so instruction set, exception handling and processor structure are the same as any real processor of the same family. Since ARM7TDMI architecture does not detail a device interface and the memory management scheme is a bit too complex, these two components are based on μMPS2 architecture, which in turn takes inspiration from commonly known architectures.

In a more schematic fashion, μARM is composed of:

- An ARM7TDMI processor.
- A system coprocessor, *CP15*, incorporated into the processor.
- Bootstrap and execution ROM.
- RAM memory little-endian subsystem with optional virtual address translation mechanisms based on Translation Lookaside Buffer.
- Peripheral devices: up to eight instances for each of five device classes. The five device classes are disks, tape devices, printers, terminals, and network interface devices.
- A system bus connecting all the system components.

This document will describe the main aspects of the emulated system, taking into exam each of its components and describing their interactions, further details regarding the processor can be found in the *ARM7TDMI Dustsheet* and the *ARM7TDMI Technical Reference Manual*.

Notational conventions:

- Registers and storage units are **bold**-marked.
- Fields are *italicized*.
- Instructions are `monospaced` and assembly instructions are `CAPITALIZED AND MONOSPACED`.
- Field F of register **R** is denoted **R**.*F*.
- Bits of storage units are numbered right-to-left, starting with 0.
- The i-th bit of a storage unit named N is denoted N[i].

- Memory addresses and operation codes are given in hexadecimal and displayed in big-endian format.
- All diagrams illustrating memory are going from low addresses to high addresses using a left to right, bottom to top orientation.

A.2 Processor

The μARM machine runs on an emulated ARM7TDMI processor, which implements both ARM and Thumb instruction sets, is able to perform each operation listed in *ARM7TDMI Data Sheet* (a brief summary is shown below) and to accept painlessly binary programs compiled with the Gnu C Compiler for ARM7 architecture.

A.2.1 Operating modes and Processor registers

The processor can work in seven different modes:

- User mode (usr) - regular user process execution
- System mode (sys) - typical privileged mode execution (e.g. kernel code execution)
- Supervisor (svc) - protected mode kernel execution
- Fast Interrupt (fiq) - protected mode for fast interrupt handling
- Interrupt (irq) - protected mode for regular interrupt handling
- Abort (abt) - protected mode for data/instruction abort exception handling
- Undefined (und) - protected mode for undefined instruction exception handling

In each mode the processor can access a limited portion of all its registers, varying from 16 registers in User/System modes to 17 registers in each protected mode in ARM state, plus the Current Program Status Register (**CPRS**), which is shared by all modes. Only protected modes have the 17th register, which automatically stores the previous value of the **CPSR** when raising an exception.

The first 8 registers, in addition to the Program Counter (**R15**) are common to each "window" of visible registers, each protected mode has its dedicated Stack Pointer and Link Return registers and Fast Interrupt mode has all the last 7 registers uniquely banked, to allow for a fast context switch, while System and User mode share the full set of 16 general purpose registers.

ARM State General Registers

User / System	FIQ	Svc	Abort	IRQ	Undef
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)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= banked register

Even if the global 16 registers are defined as general purpose registers, there are some conventions adopted by the compiler in their use. The following list shows the full set of processor register visible in each mode with their conventional abbreviations and extended meaning:

- **R0** (**a1**) - first function argument / integer result
- **R1** (**a2**) - second function argument
- **R2** (**a3**) - third function argument
- **R3** (**a4**) - fourth function argument
- **R4** (**v1**) - register variable
- **R5** (**v2**) - register variable

- **R6** (**v3**) - register variable
- **R7** (**v4**) - register variable
- **R8** (**v5**) - register variable
- **R9** (**v6/rfp**) - register variable / real frame pointer
- **R10** (**sl**) - stack limit
- **R11** (**fp**) - frame pointer / argument pointer
- **R12** (**ip**) - instruction pointer / temporary workspace
- **R13** (**sp**) - stack pointer
- **R14** (**lr**) - link register
- **R15** (**pc**) - program counter
- **CPSR** - current program status register
- **SPSR_*mode*** - saved program status register

When the processor is in Thumb state the register window is reduced, showing 12 registers in User/System mode and 13 registers in protected modes, in addition to the Current Program Status Register, which is common to all modes.

Only the first 8 registers (**R0** → **R7**) are general purpose, the higher 3 are specialized registers that act as Stack Pointer, Link Return and Program Counter. Each protected mode has its own banked instance of Stack Pointer and Link Return in addition to Saved Program Status Register to allow for faster exception handling.

Thumb State General Registers

User / System	*FIQ*	*Svc*	*Abort*	*IRQ*	*Undef*
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
SP	SP_fiq	SP_svc	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

(shaded) = banked register

A.2.2 System Coprocessor

CP15 gives access to a total of three 64-bit and four 32-bit registers which provide additional information and functionalities to regular processor operations:

Register 0 (IDC) - ID Codes

R0 is a read-only 64-bit register that contains system implementation information such as *Processor ID* and *TLB type*, as required by ARM specifications.

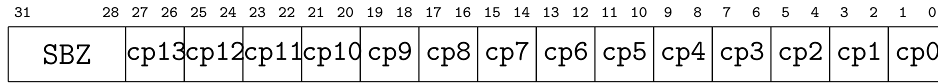
Register 1 (SCB) - System Control Bits

R1.SCB is the System Control Register, this register holds system-wide settings flags. See sec. A.2.3 for further details.

Register 1 (CCB) - Coprocessors Access Register

R1.CCB shows which coprocessors are available. Values can be written to this register to enable/disable available coprocessors a part from CP15.

Coprocessors Access Register




```
SBZ :      Should Be Zero
cp* :  00  Access Denied.
           (Accessing this coprocessor generates Undefined Exception)
       01  Privileged Access Only.
           (Accessing this coprocessor in user mode generates Undefined Exception)
       10  RESERVED.
           (Unpredictable)
       11  Full Access.
```

Register 2 - Page Table Entry

R2 is a 64-bit register which contains the active Page Table Entry when MMU is enabled. Its structure is the same as the Page Table Entry described in sec. A.4.2.

The lower 32 bits of this register are addressed as **CP15.R2.EntryLow** or just **EntryLow** and the higher 32 bits are addressed as **CP15.R2.EntryHi** or simply **EntryHi**.

Register 6 - Faulting Address

R6 is a read-only register that is automatically loaded each time a Page Fault Exception is raised with the memory address that generated the exception.

Register 8 - TLB Random

R8 is a special read-only register used to index the TLB randomly (see sec. A.4.2). This register is also addressed as **Random** or **TLBR**.

Register 10 - TLB Index

R10 is a special register used to index the TLB programmatically (see sec. A.4.2). This register is also addressed as **Index** or **TLBI**.

Register 15 (*Cause*) - Exception Cause

R15.*Cause* contains the last raised exception cause, it can be read or written by the processor.

A scheme of Memory Access exception cause codes is shown below:

Memory Error = 1
Bus Error = 2
Address Error = 3
Segment Error = 4
Page Error = 5

Register 15 (*IPC*) - Interrupt Cause

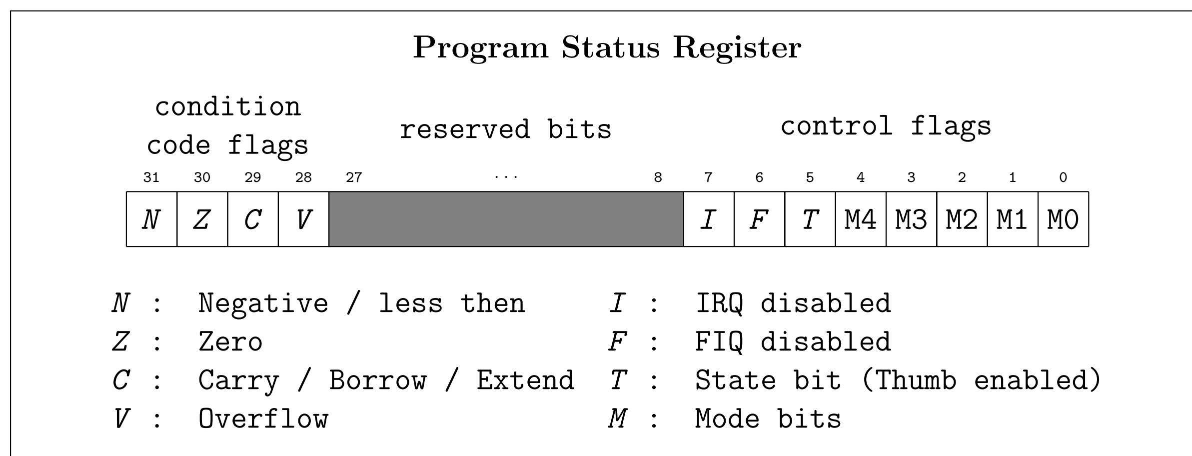
R15.*IPC* shows on which lines interrupts are pending, when interrupts have been handled (i.e. the interrupt request has been acknowledged, see Sec. A.5) the value of this register is updated.

A.2.3 Execution Control

The processor behavior can be set up by modifying the contents of two special registers: the Current Program Status Register (**CPSR**) and the System Control Register (**CP15.SCB**). Each of the two has a special structure and changes the way the system operates.

Program Status Register

The **CPSR** (as well as the **SPSR**, if the active mode has one) is always accessible in ARM state via the special instructions MSR (move register to PRS) and MRS (move PRS to register). This register shows arithmetical instructions' additional results (condition code flags), allows to toggle interrupts and switch states/modes. Its structure is shown below:



The first 5 bits of **CPSR** are used to set processor execution mode, the possible values are:

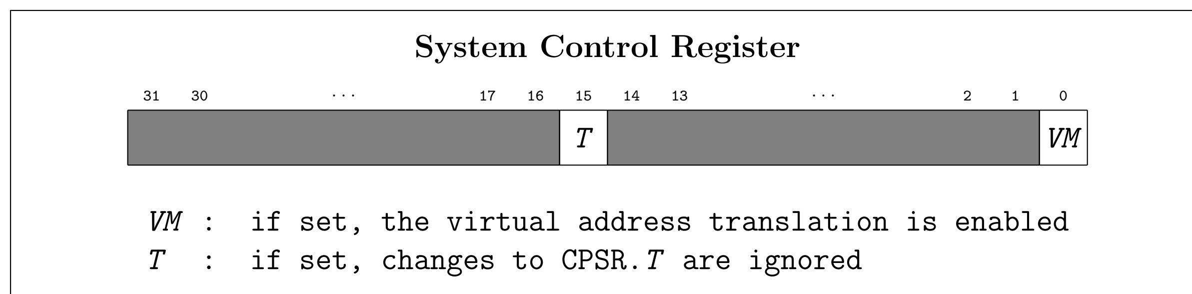
`0x10`	User Mode
`0x11`	Fast Interrupt Mode
`0x12`	Interrupt Mode
`0x13`	Supervisor Mode
`0x17`	Abort Mode
`0x1B`	Undefined Mode
`0x1F`	System Mode

User Mode is the only unprivileged mode, this means that if the processor is running in User Mode, it cannot access reserved memory regions (see System Bus chapter) and it cannot modify CPSR control bits.

System Mode is the execution mode reserved for regular Kernel code execution, all other modes are automatically activated when exceptions are raised (see sec. A.2.5).

System Control Register

System Coprocessor (CP15) holds the System control register (**CP15.R1**), which controls Virtual Memory and Thumb availability (plus some other hardware specific settings that are not implemented in current release):



A.2.4 Processor States

The *T* flag of the Program Status Register shows the state of the processor, when the bit is clear the processor operates in ARM state, otherwise it works in Thumb state. To switch between the two states a Branch and Exchange (BX) instruction is required.

The first difference between the two states is the register set that is accessible (see sec. A.2.1), the other main difference is the Instruction Set the processor is able to decode.

ARM ISA

The main Instruction Set is the ARM ISA, the processor starts execution in this state and is forced to ARM state when an exception is raised.

ARM instructions are 32 bits long and must be word-aligned. The table below shows a brief summary of the instruction set. (For a much detailed description of each instruction refer to *ARM7TMI Data Sheet* and *ARM7TMI Technical Reference Manual*)

ADC	add with carry	ADD	add
AND	logical AND	B	brach
BIC	bit clear	BL	branch with link
BX	branch and exchange	CDP	coprocessor data processing
CMN	compare negative	CMP	compare
EOR	logical exclusive OR	LDC	load coprocessor register from memory
LDM	load multiple registers from memory	LDR	load register from memory
LDRH	load halfword from memory	LDRSB	load signed byte from memory
LDRSH	load signed halfword from memory	MCR	move cpu register to coprocessor register
MLA	multiply accumulative	MLAL	multiply accumulative long
MOV	move register or constant	MRC	move coprocessor register to cpu register
MRS	move PRS status/flags to register	MSR	move register to PRS status/flags
MUL	multiply	MULL	multiply long
MVN	move negative register	ORR	logical OR
RSB	reverse subtract	RSC	reverse subtract with carry
SBC	subtract with carry	STC	store coprocessor register to memory
STM	store multiple	STR	store register to memory
STRH	store halfword	SUB	subtract
SWI	software interrupt	SWP	swap register with memory
TEQ	test bitwiser equality	TST	test bits
UND	undefined instruction		

Thumb ISA

Thumb instruction set is a simpler (smaller) instruction set composed of 16-bit, halfword aligned instructions, which offer less refined functionalities but less memory usage.

Thumb instructions can be seen as "shortcuts" to execute ARM code, as the performed operations are the same but this ISA offers less options for each instruction.

The following table summarizes Thumb instructions. (For a much detailed description of each instruction refer to *ARM7TMI Data Sheet* and *ARM7TMI Technical Reference Manual*)

ADC	add with carry
AND	logical AND
B	unconditonal branch
BIC	bit clear
BX	branch and exchange
CMP	compare
LDMIA	load multiple (increment after)
LDRB	load byte to register
LDRSB	load signed byte to register
LSL	logical shift left
MOV	move from register to register
MVN	move negative register
ORR	logical OR
PUSH	push to stack
SBC	subtract with carry
STR	store register to memory
STRH	store halfword to memory
SWI	software interrupt

ADD	add
ASR	arithmetical shift right
B[cond]	conditioned branch
BL	branch with link
CMN	compare negative
EOR	logical exclusive OR
LDR	load word to register
LDRH	load halfword to register
LDRSH	load signed halfword to register
LSR	logical shift right
MUL	multiply
NEG	negate word
POP	pop from stack
ROR	rotate right
STMIA	store multiple (increment after)
STRB	store byte to memory
SUB	subtract
TST	test bits

A.2.5 Exception Handling

When an exception is raised (e.g. a read instruction is performed on a forbidden bus address), the processor automatically begins a special routine to solve the problem. In addition to low level automatic exception handling facilities, the BIOS code implements a wrapper to simplify OS level handlers setup and functioning.

Hardware Level Exception Handling

There are seven different exceptions handled by the processor, each of those has a specific bus address to which the execution jumps on exception raising (see sec. A.3.1).

When an exception is raised, the processor state is forced to ARM mode, the execution mode and the interrupt flags are set accordingly to the exception type, the **LR** register is filled with a return address and the **PC** is loaded with the correct address of the bus area

Exception Vector. If the Exception Vector has been correctly initialized, the instruction pointed to by the **PC** is a Branch instruction leading to some handler code.

Follows a brief description of the exceptions and the different return addresses.

Reset Exception

This exception is automatically raised each time the machine is started.

This exception is handled in Supervisor mode with all interrupts disabled, Link Return and SPSR registers have unpredictable values and execution starts from bus address `0x0000.0000`.

The first bus word, when the execution starts, is always filled with a fixed Branch instruction that makes the processor jump to the beginning of the bus-mapped ROM (address `0x0000.0300`), where the BIOS is stored.

Undefined Instruction Exception

If a Coprocessor instruction cannot be executed from any Coprocessor or if an UNDEFINED instruction is executed, this exception is raised.

Processor mode is set to Undefined, normal interrupts are disabled and Link Return register points to the instruction right after the one that caused the Undefined Exception.

Software Interrupt Exception

This exception is caused by a SWI instruction and is meant to provide a neat way to implement System Calls.

When handling Software Interrupt Exceptions, the processor switches to Supervisor mode with normal interrupts disabled and the Link Return register points to the instruction after the SWI that caused the exception.

Data Abort Exception

If the processor tries to access a memory address that is not valid or available, this exception is raised.

When handling Data Aborts, the processor switches to Abort mode with normal interrupts disabled and Link Return register is set to the address of the instruction that caused the Abort plus 8.

Prefetch Abort Exception

If the processor tries to execute an instruction that generated a data abort while being fetched, this exception is raised.

When handling Prefetch Aborts, the processor enters Abort mode with normal interrupts disabled and Link Return register points to the address of the instruction after the one that caused the exception.

Interrupt Request Exception

When a connected device requires the processor's attention, it fires an Interrupt Request.

When handling Interrupt Requests, the processor enters Interrupt mode with normal interrupts disabled and Link Return register is set to the address of the instruction that was not executed plus 4.

Fast Interrupt Request Exception

Fast Interrupts have higher priority than normal Interrupts, system Interval Timer is connected to this line of interrupts.

The Interval Timer's value is decreased at each execution cycle, when an underflow occurs (i.e. its value changes from `0x0000.0000` to `0xFFFF.FFFF`), a Fast Interrupt is requested.

When handling Fast Interrupt Requests, the processor enters Fast Interrupt mode with all interrupts disabled and Link Return register points to the address of the instruction that was not executed plus 4.

ROM Level Exception Handling

During the bootstrap process, six of the seven Exception Vector registers must be initialized (the reset exception only occurs at system startup and the relative register has a fixed value). The BIOS code fills these registers with jump instruction opcodes pointing to its internal handler procedures.

The BIOS exception handlers provide a safe and automatic way to enter kernel level handlers by storing the processor state as it was before the exception was raised and loading the kernel handler's processor state from a known memory location inside the Kernel Reserved Frame (see sec. A.3.1). While storing the processor state, all ROM Level Exception Handlers move the value of the **lr** register to the **pc** register, making easier the return to the regular execution.

In addition to this general behavior, some handlers provide other functionalities as described below.

Undefined Instruction Handler

An Undefined Instruction Exception needs no special treatment, its handler stores the old processor state into the PGMT Old Area and loads the processor state stored into the PGMT New Area.

Software Interrupt Handler

Software interruptions recognized by the BIOS handler can be of two types: System Calls or Breakpoints, the foremost being interpreted as a request to the kernel, while the latter can also be a BIOS service request.

This handler is capable of recognizing BIOS service requests and serving them directly, if a System Call or an unrecognized Breakpoint is requested, the exception is handled with the default behavior, the old processor state is stored into the Syscall Old Area and the processor state stored into the Syscall New Area is loaded.

Data Abort and Prefetch Abort Handler

If Virtual Memory is enabled (see sec. A.4.2), both Data and Prefetch Aborts can be raised while accessing a memory frame whose VPN is not loaded into the TLB, event signaled by the memory subsystem through these two exceptions. If this is the case, the BIOS will automatically perform a TLB_Refill cycle, searching the active Page Tables for the required entry; otherwise the exception is treated as a generic exception, storing the old processor state into TLB Old Area and then loading the processor state stored into the TLB New Area.

Since the two different exception types have different return address offsets (see sec. A.2.5), this handler modifies the return address stored within the old processor state in order to be the correct address to jump to. This behavior is necessary, because discerning the exception type from kernel level handler could be quite difficult, or even impossible at times, without the knowledge of the hardware level exception, that is given from processor mode and **CPSR.R15**.*Cause* and could be lost during the pass-up.

Interrupt and Fast Interrupt Handlers

Both these exceptions are treated as generic exceptions and the BIOS handlers will adopt the default behavior, storing the old processor state into the Interrupt Old Area and then loading the processor state stored into the Interrupt New Area.

Notes About Exception Handlers

When writing Exception Handlers code, it is well advised to pay attention to the Program Counter value stored in the Old Area. As described above, each exception leaves a different value in Link Return register and this value is automatically moved to **Old Area.pc** by the ROM Level Exception Handlers, so, for example, when handling an Interrupt, the **Old Area.pc** has to be decreased by 4 to point to the right return instruction.

A.3 System Bus

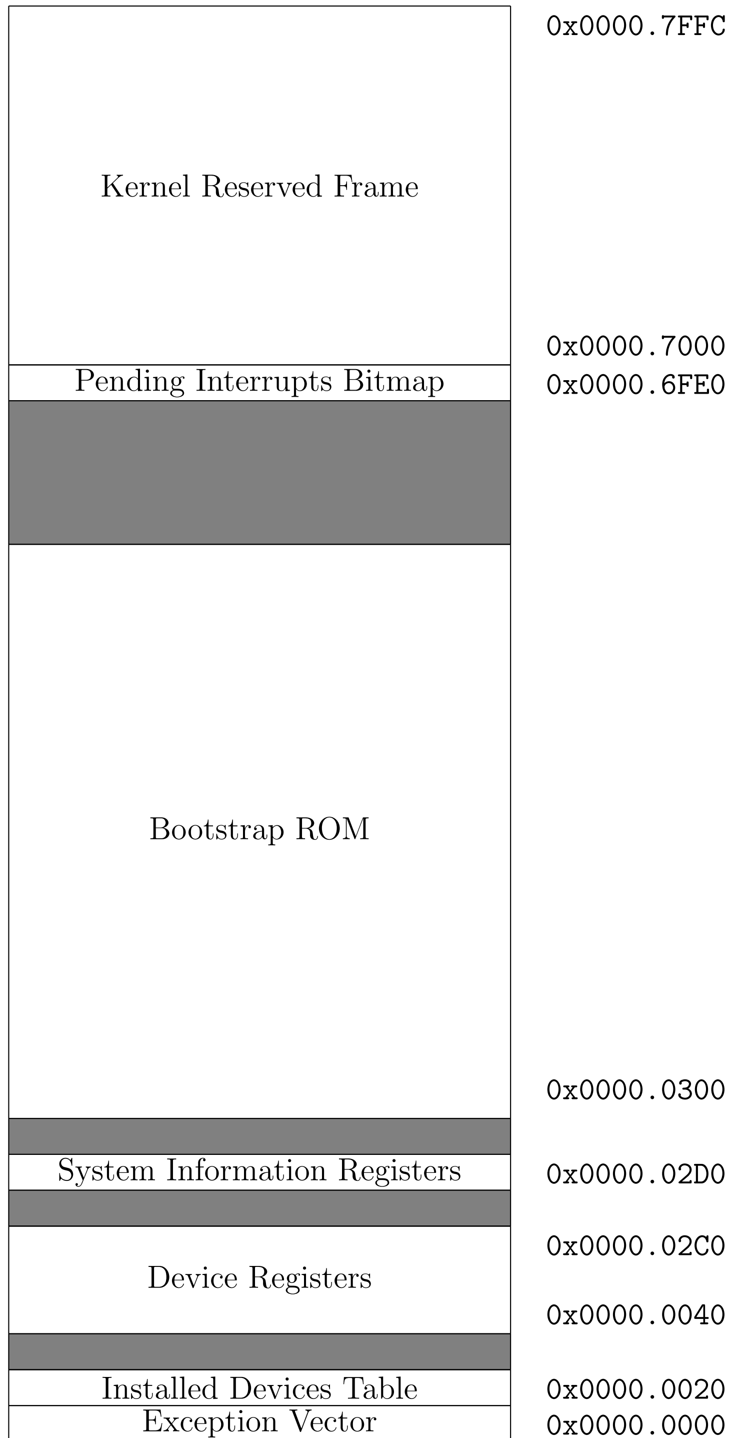
The system bus connects each component in the system and lets processing units access physical memory and devices, as well as some special purpose registers.

The CPU and the System Coprocessor (CP15) can directly access the bus reading or writing values from or to specific addresses.

The lower addresses (below `0x0000.8000`) are reserved for special uses and are accessible under certain conditions.

A.3.1 Reserved address space

The address region between `0x0000.0000` and `0x0000.8000` holds the fast exception vector, device access registers, system information registers, bootstrap ROM and the kernel reserved frame (i.e. the first RAM frame). Any access to this memory area in User mode is (should be) prohibited and treated by the system bus as errors.


Reserved Address Space
Kernel Reserved Frame
0x0000.7FFC
0x0000.7000
Pending Interrupts Bitmap
0x0000.6FE0
Bootstrap ROM
0x0000.0300
System Information Registers
0x0000.02D0
0x0000.02C0
Device Registers
0x0000.0040
Installed Devices Table
0x0000.0020
Exception Vector
0x0000.0000

Exception Vector

The first bus addresses (`0x0000.0000` → `0x0000.001C`) are occupied by the fast exception vectors. Whenever an exception is risen, the processor automatically changes the **PC** register to point to one of these addresses. This way, if the system was correctly set up, a branch instruction will lead execution to the correct exception handler. It is the bootstrap ROM which typically writes a set of branch instructions to exception handlers in these fields.

The exception vector is organized as follows:

Exception Vector

Exception	Address
Fast Interrupt Request	`0x0000.001C`
Interrupt Request	`0x0000.0018`
reserved/unused	`0x0000.0014`
Data Abort	`0x0000.0010`
Prefetch Abort	`0x0000.000C`
Software Interrupt	`0x0000.0008`
Undefined Instruction	`0x0000.0004`
Reset	`0x0000.0000`

Installed Devices Table

Five words, from `0x0000.0020` to `0x0000.0030`, show the status of active devices. Each word represents a device class: for each word, if a specific device i is enabled, i^{th} bit in relative word has value 1.

Installed Devices Table

Device class	Address
Terminals	`0x0000.0030`
Printers	`0x0000.002C`
Network	`0x0000.0028`
Tapes	`0x0000.0024`
Disks	`0x0000.0020`

31–8	7	6	5	4	3	2	1	0
Device Class Bitmap	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Device Registers

Addresses from `0x0000.0040` to `0x0000.02C0` hold device registers. Each device type has its own communication protocol, as described in chapter A.5.

System Information Registers

Six registers, from address `0x0000.02D0` to `0x0000.02E4`, show system specific information:

System Information Registers

Register	Address
Interval Timer	`0x0000.02E4`
Time of day (Low)	`0x0000.02E0`
Time of day (High)	`0x0000.02DC`
Device registers base addr.	`0x0000.02D8`
RAM top address	`0x0000.02D4`
RAM base address	`0x0000.02D0`

These registers are all read-only, except for the interval timer which is a special device register (see sec. A.5.2), and are aimed to provide useful information to the operating system.

Bootstrap ROM

The bootstrap ROM is loaded starting from address `0x0000.0300`, its maximum size is 109 KB. The content of the ROM is actually flashed at each reboot of the emulator copying each byte of the input image starting from the ROM base address, so the BIOS image does not need any special offset set by the linker.

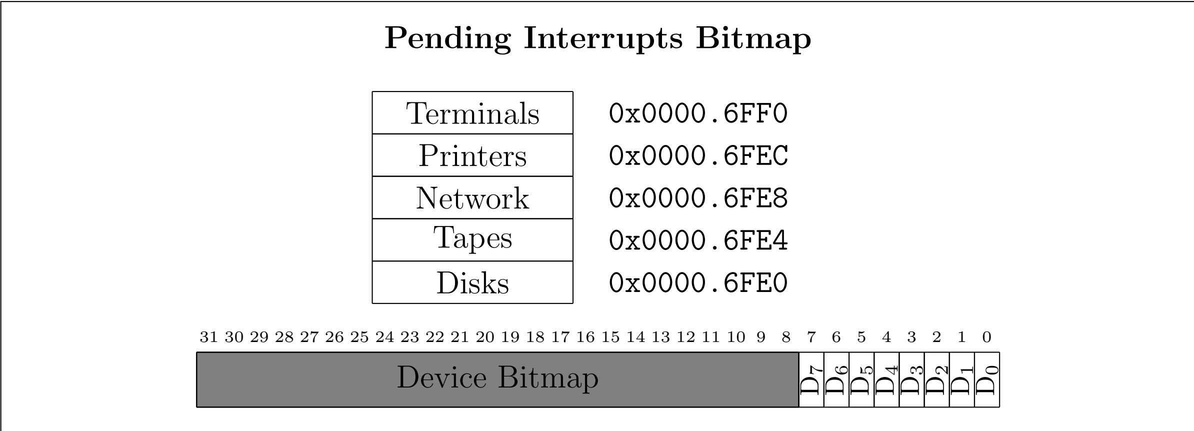
See section A.6.1 for further details regarding provided BIOS implementation.

Pending Interrupts Bitmap

All the devices of one class are connected to the same interrupt line, when a device needs to notify some activity to the processor it sends a message through its interrupt line. To identify which specific device is requesting an interruption, there are five registers from address `0x0000.6FE0` to `0x0000.6FF0` that hold a bitmap of interrupting devices per interrupt line.

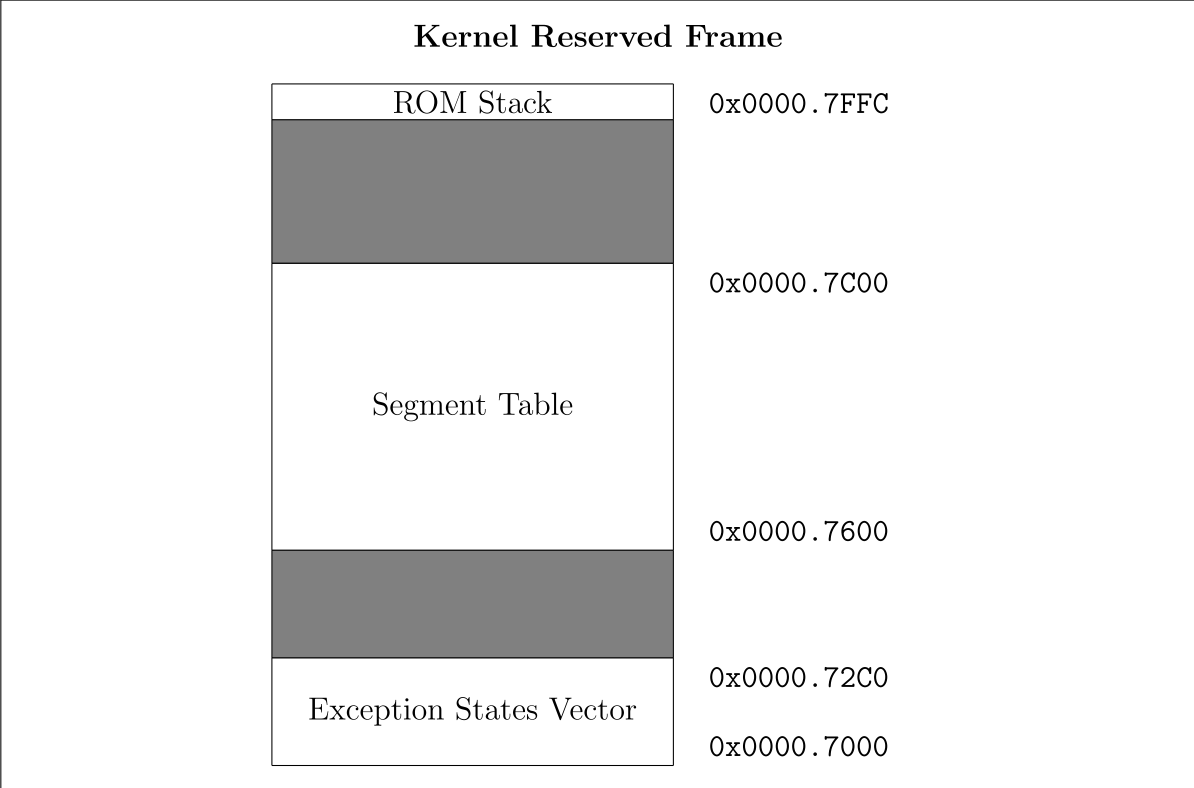
For each word, i bit is set if i^{th} device on that line is requesting for interrupt.

This region is organized exactly as the Installed Device Table:



Kernel Reserved Frame

The first memory frame (0x0000.7000 → 0x0000.7FFC) is reserved for kernel use:



The Exception states vector is a memory area to which processor states are saved and loaded from when entering into/exiting from exception handlers code. The Segment table holds 128 elements describing the virtual address space, for each ASID (corresponding to one entry in the segment table) there are three pointers to ASID's page tables (see

sec. A.4.2). When invoking ROM functions, the four words wide ROM stack (the last four words in the frame) is used as a memory stack and to pass parameters by the low level routines.

Stored Processor States

Processor states are defined by library data structure *state_t*, this structure is composed of 22 unsigned 32-bit integers representing processor registers' values and coprocessor's system control registers' values as well as saving time.

Its structure is shown below:

```
typedef struct {
   unsigned int a1;    //r0
   unsigned int a2;    //r1
   unsigned int a3;    //r2
   unsigned int a4;    //r3
   unsigned int v1;    //r4
   unsigned int v2;    //r5
   unsigned int v3;    //r6
   unsigned int v4;    //r7
   unsigned int v5;    //r8
   unsigned int v6;    //r9
   unsigned int sl;    //r10
   unsigned int fp;    //r11
   unsigned int ip;    //r12
   unsigned int sp;    //r13
   unsigned int lr;    //r14
   unsigned int pc;    //r15
   unsigned int cpsr;
   unsigned int CP15_Control;
   unsigned int CP15_EntryHi;
   unsigned int CP15_Cause;
   unsigned int TOD_Hi;
   unsigned int TOD_Low;
} state_t;
```

These structures take 88 bytes each. Given this value, the BIOS code will look for the Old/New entries at the following addresses:

Exception States Vector

Area	Address
Syscall New	0x0000.7268
Syscall Old	0x0000.7210
PGMT New	0x0000.71B8
PGMT Old	0x0000.7160
TLB New	0x0000.7108
TLB Old	0x0000.70B0
Interrupt New	0x0000.7058
Interrupt Old	0x0000.7000

Each time an exception is risen, the BIOS handlers will store the processor state before the exception into the proper Old area, perform other tasks where required (see sec. A.2.5) and eventually load the processor state stored in the corresponding New area.

The New areas must be filled, during kernel initialization stage, with valid processor states pointing to kernel level exception handlers.

A.3.2 Memory address space

The remaining addresses (`0x0000.8000` $\rightarrow$ `RAMTOP`) are mapped to memory subsystem, this bus region can be directly accessed by the processor and the coprocessor, it is used to store the kernel execution code and data as well as any other program that has to be executed.

The memory is accessible in physical addressing mode or virtual addressing mode, access modes and memory structure are described in detail in chapter A.4.

A.4 Memory Interface

Memory system is controlled by Program Status Register (**CPSR**) and System Co-processor's registers 1 and 2 (**CP15.R1** & **CPSR.R2**). It supports two operating modes:

- physical addressing mode,
- virtual addressing mode.

In addition to address translation modes, the portion of accessible memory is dictated by processor operating mode:

- User mode → User Space
- Privileged mode → All Memory

In each addressing mode these portions have a specific definition.

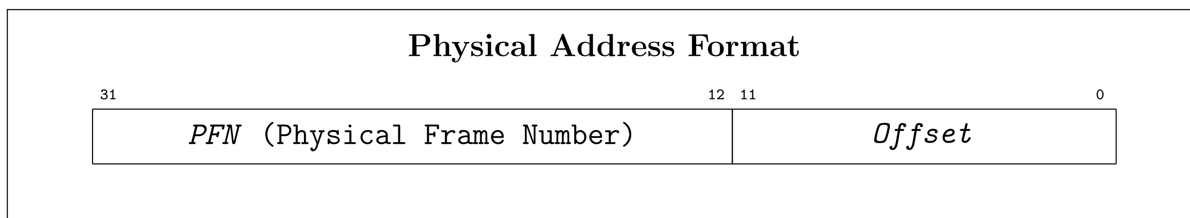
As stated in section A.3.1, addresses below `0x0000.8000` are reserved for hardware/-protected functions and belong to the reserved address space independently from the active addressing mode.

A.4.1 Physical addressing mode

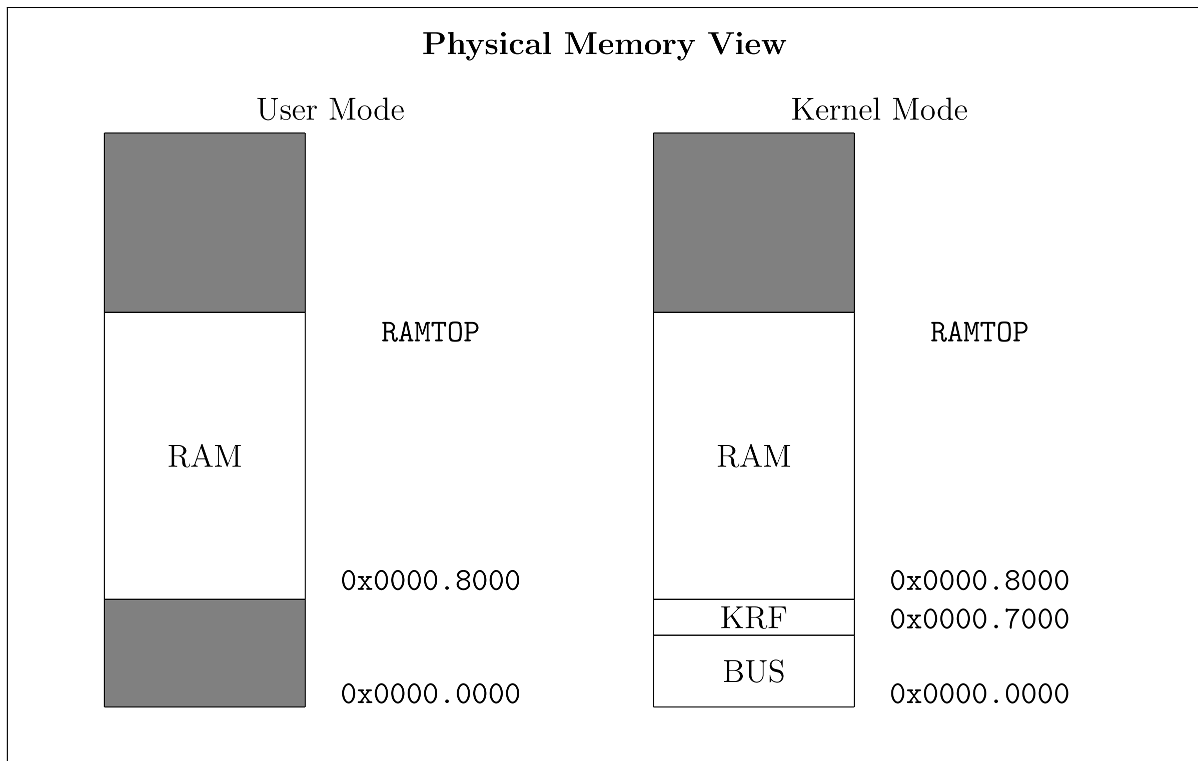
This is the basic memory access scheme, when enabled, any memory access is directed to the physical address specified in the request, without any conversions. The machine begins execution with this mode active.

The physical address space is divided into equal sized frames of 4KB each. Hence a physical address has two components; a 20-bit Physical Frame Number or *PFN*, and a 12-bit *Offset* into the frame. Physical addresses have the following format:

Physical Address Format



All the available memory is directly accessible in Privileged mode and any address over `0x0000.8000` is directly accessible in User mode.



Trying to access an address below `0x0000.8000` while in User Mode will raise an `Address Error` exception.

The installed physical RAM starts at `0x0000.7000` and continues up to `RAMTOP`, this area will hold:

- The operating system code (.text), global variables/structures (.data), and stack(s).
- The user processes' .text, .data and stacks.
- The Kernel Reserved Frame. As detailed in section A.3.1, the BIOS code needs some writable storage to interact with the Kernel. The first 4KB (i.e. the first frame) of physical RAM are reserved for this purpose.

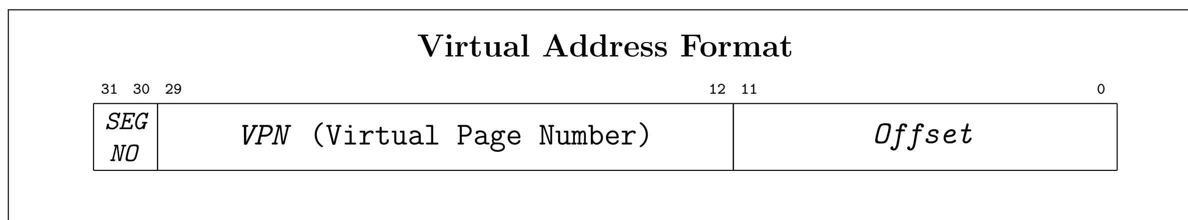
The first 7KB of the physical address space are reserved for Bus functions, as described in section A.3.1. Any attempt to access an unidentified memory area will generate a `BUSERROR` exception.

A.4.2 Virtual addressing mode

When virtual memory is active, each address above `0x0000.8000` is treated as a logical address and translated to the corresponding physical address from the memory subsystem. Addresses below `0x0000.8000` are always treated as physical addresses, as they refer to a reserved address region (see sec. A.3.1).

By setting *M* flag in System Coprocessor's register 1 (**CP15.R1**.*M*), you enable memory address translation.

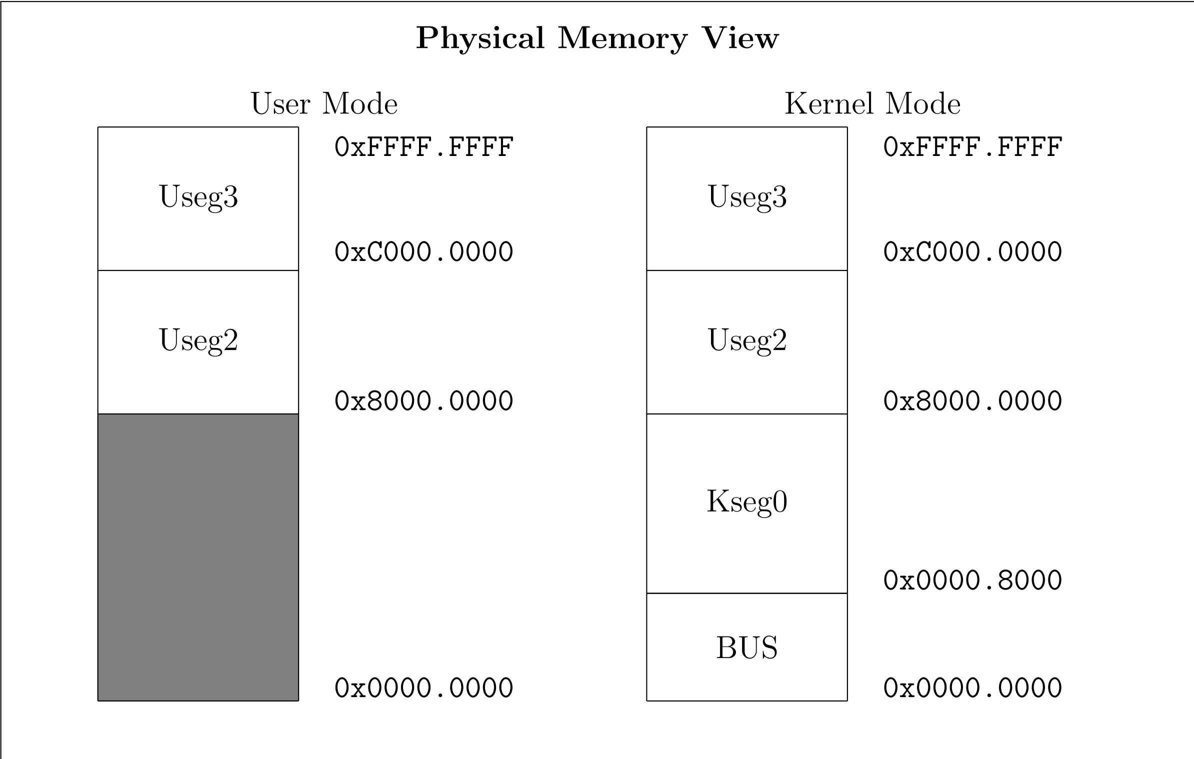
The first two bits of a virtual address are the Segment Number (*SEGNO*). Virtual pages are the same size as physical frames, the final 12-bits indicate an *Offset* into a memory frame. The remaining 18-bits indicate the Virtual Page Number or *VPN*. Virtual addresses have the following format:



The segment number is composed of two bits, the most important one differentiates kernel and user segments, the least important one is meaningful only in user segment and identifies private segment and global segment:

- Kseg0 (*SEGNO* 0 and 1) is the 2GB segment for the OS .text, .data, stacks, as well as the ROM code and device registers that sit at the beginning of this segment.
- Useg2 (*SEGNO* 2) is the 1GB virtual address space for the use of User mode processes as private memory region.
- Useg3 (*SEGNO* 3) is the 1GB virtual address space for the use of User mode processes as shared/global memory region.

In Privileged mode the whole logical memory address space is accessible, while in User mode only User Segments are available and any access to Kseg0 segment will generate an `Address Error` exception.

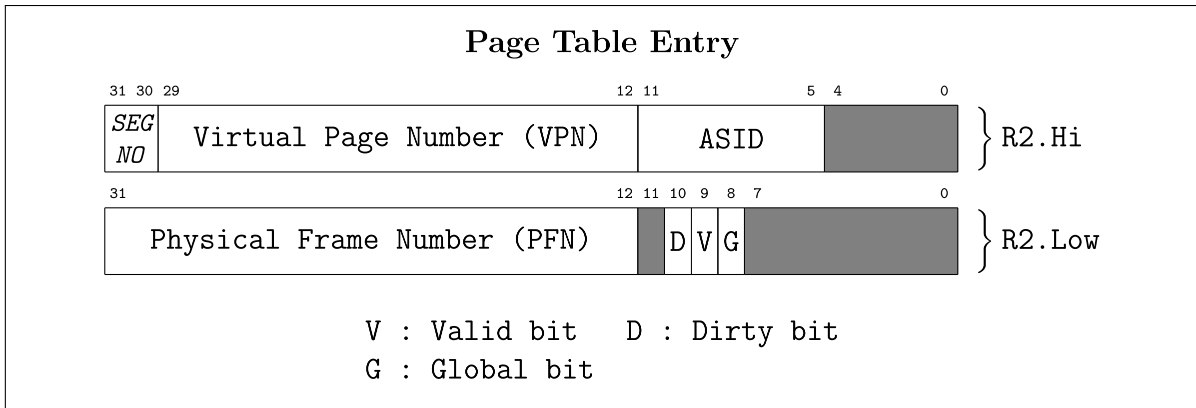


As part of its VM implementation, μARM assigns to each process a 7-bit identifier; hence μARM natively allows up to $2^7 = 128$ concurrent processes. To reflect the fact that each of these processes will run in its own virtual address space, this identifier is called the Address Space Identifier (*ASID*). The "current" *ASID* is part of the processor state and is stored in **EntryHi**.*ASID* (**CP15.R2.EntryHi**.*ASID*).

When the MMU is enabled the user process *ASID* is stored in the **EntryHi** register along with the Virtual Page number (i.e. the 20 most significant bits of the logical address). The **EntryLow** register is filled with the Physical Frame Number from the relevant page table and is kept up to date after each modification of **CP15.R2.EntryHi** value.

Page Table

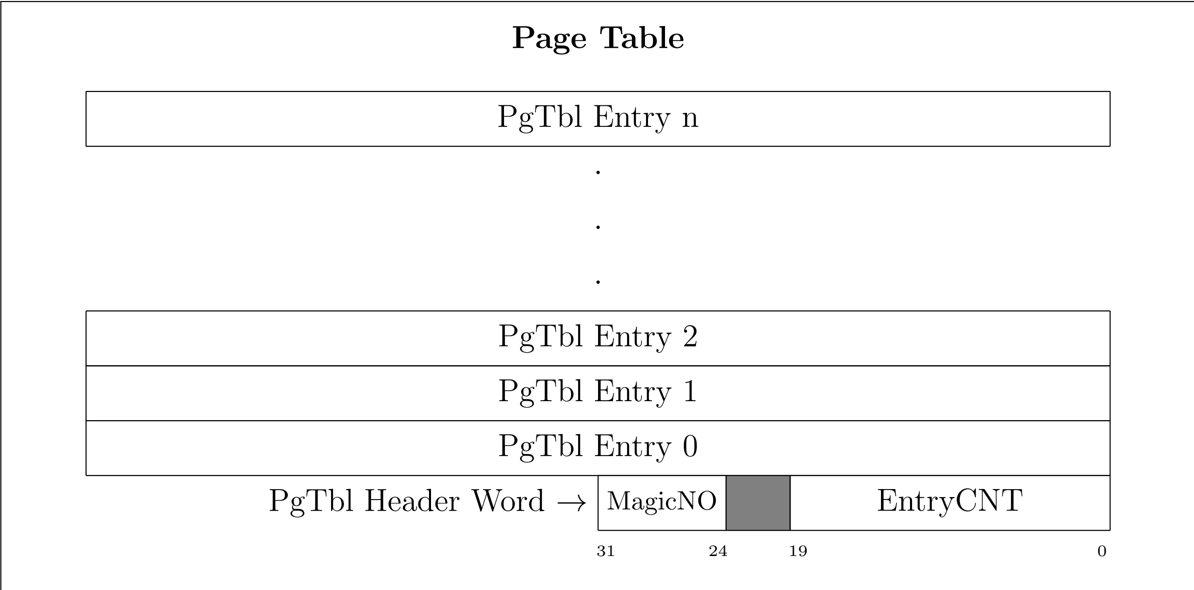
The **CP15.R2** register is organized as a Page Table Entry (PTE):



The Hi half of each entry identifies the logical frame to which the entry refers and the *ASID* of the owning process. The Low half of each entry specifies the physical corresponding frame (if any) and contains 3 flags used for memory protection schemes:

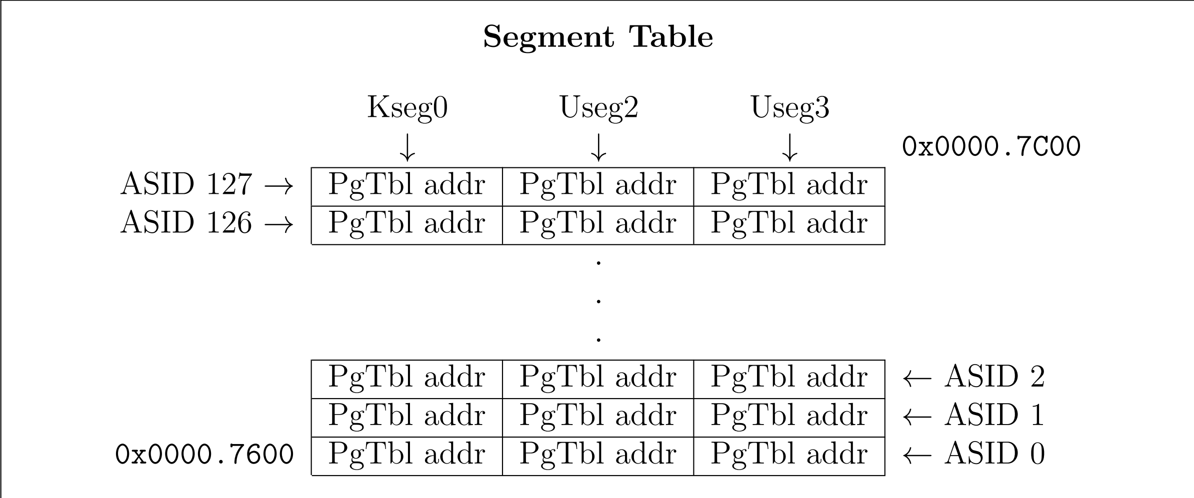
- **D**irty bit: if the flag is clear, any write access to the physical frame locations will rise a `TLB-Modification` exception.
- **V**alid bit: if the flag is set the Page Table Entry is considered valid, otherwise a `TLB-Invalid` exception is raised. This flag should be set only when the *PFN* points to the actual memory frame.
- **G**lobal bit: if the flag is set, the Page Table Entry will match the corresponding *VPN* regardless of the *ASID*.

Page Table Entries are grouped together in Page Tables, each Page Table begins with a special word (PgTbl-Header) composed of the PgTbl Magic Number `0x2A` stored in the most significant 8 bits and the number of page table entries in the least significant 20 bits, as shown below.



Segment Table

The Segment Table specifies the physical addresses of the Page Tables describing the three Segments for each *ASID*, the general structure is shown below:



The segment table is automatically accessed from BIOS code when the Page Table Entry describing a needed memory frame is not present inside the TLB and needs to be retrieved (see sec. A.4.2).

Translation Lookaside Buffer

μARM implements a Translation Lookaside Buffer (TLB) to translate virtual addresses to physical addresses, the buffer contains a specific amount of recently used PTEs and can use a variety of algorithms to select which entry should be replaced with a newly retrieved PTE (the BIOS handler implements a simple random selection). The number of available TLB slots is variable between 4 and 64 elements, it is configurable through the settings window of the emulator and needs a reset of the machine to effectively change.

Each time a memory access is requested, the memory subsystem checks if the requested Virtual Page has a corresponding PTE in the TLB for the current *ASID* or with the *G* flag set. If the necessary PTE is not present in the TLB, a `TLB-Miss` exception is raised and the BIOS reacts with a TLB Refill event, which is composed of the next steps:

1. Retrieve the PgTbl address from the Segment Table for the current ASID and required Segment.
2. Access the PgTbl and check if it is well-formed and well-located:
   - Address must be greater than `0x0000.8000`,
   - Address must be word aligned,
   - PgTbl-Header must be valid (magic number is `0x2A`),
   - PgTbl must not extend outside physical memory (e.g. [PgTbl addr + PgTbl size] < RAMTOP).

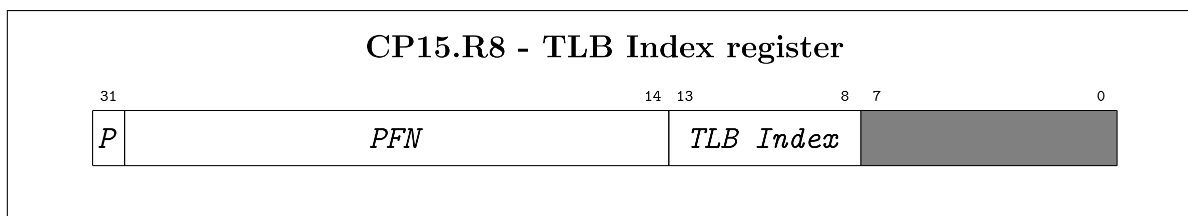
   If one of these checks fails a `Bad-PgTbl` exception is raised.
3. Linearly search the PgTbl for matching Virtual Page with correct *ASID* or *G* flag set.
4. If a matching PTE has been found, write it back in a random slot of the TLB and resume execution from the same instruction that raised the `TLB-Miss` exception, else raise a `PTE-Miss` exception.

At this point, either there is a matching PTE, or an exception has been raised (either an `Address Error`, `Bad-PgTbl`, or `PTE-Miss` exception). If there is a matching TLB entry then the *V* and *D* control bits of the matching PTE are checked respectively. If no `TLB-Invalid` or `TLB-Modification` exception is raised, the physical address is constructed by concatenating the *Offset* from the virtual address to be translated to the *PFN* from the matching PTE.

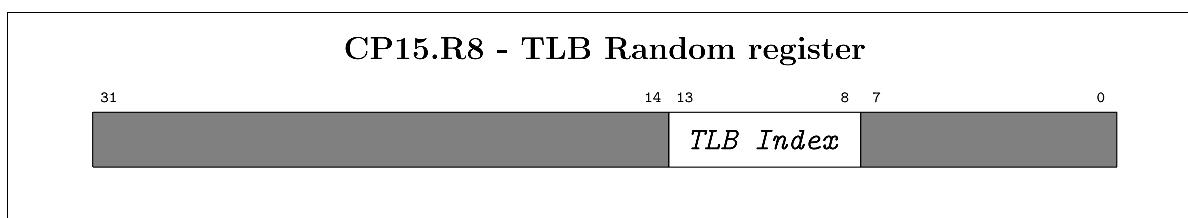
CP15 registers used in address translation

CP15 implements four registers used to support virtual address translation. The first two have already been described, they are **CP15.R1** and **CP15.R2**, used respectively to turn on or off address translation and to cache the active Page Table Entry.

The contents of the TLB can be modified by writing values into the **EntryHi** and **EntryLo** registers and issuing either the TLB-Write-Index (TLBWI) or TLB-Write-Random (TLBWR) CP15 instruction. Which slot in the TLB the entry is written into is determined by which instruction is used and the contents of either the **CP15.R8** (**Random**) or **CP15.R10** (**Index**) register. Both the **Random** and the **Index** registers have a 6-bit *TLB-Index* field which addresses one of the `TLBSIZE` slots in the TLB.



The **Index** register is a read/writable register. When a TLBWI instruction is executed, the contents of the **CP15.R2** register are written into the slot indicated by **Index**.*TLB-Index*.



The **Random** register is a read-only register used to index the TLB randomly; allowing for more effective TLB-refiling schemes. **Random**.*TLB-Index* is initialized to `TLBSIZE`-1 and is automatically decremented by one every processor cycle until it reaches 1 at which point it starts back again at `TLBSIZE`-1. This leaves one TLB "safe" entry (entry 0) which cannot be indexed by **Random**. When a TLBWR instruction is executed, the contents of the **CP15.R2** register are written into the slot indicated by **Random**.*TLB-Index*. (μARM's TLB Refill algorithm uses TLBWR to populate the TLB.)

Three other useful CP15 instructions associated with the TLB are the TLB-Read (TLBR), TLB-Probe (TLBP), and the TLB-Clear (TLBCLR) commands.

- The TLBR command places the TLB entry indexed by **Index**.*TLB-Index* into the **CP15.R2** register. Note, that this instruction has the potentially dangerous affect of altering the value of **EntryHi**.*ASID*.

- The TLBP command initiates a TLB search for a matching entry in the TLB that matches the current values in the **EntryHi** register. If a matching entry is found in the TLB the corresponding index value is loaded into **Index**.*TLB-Index* and the Probe bit (**Index**.*P*) is set to 0. If no match is found, **Index**.*P* is set to 1.

- The TLBCLR command zero's out the "unsafe" TLB entries; entries 1 through `TLBSIZE`-1. This command effectively invalidates the current contents of the TLB cache.

See Sections A.6.2 for more details on the TLBWI, TLBWR, TLBR, TLBP, TLBCLR CP15 instructions and how to access the **CP15.R2** and **Index** registers.

A.5 External Devices

This Section is an adapted revision of μMPS2 - Principles of Operation, Chapter 5.

μARM supports five different classes of external devices: disk, tape, network card, printer and terminal. Furthermore, μARM can support up to eight instances of each device type. Each single device is operated by a controller. Controllers exchange information with the processor via device registers; special memory locations.

A device register is a consecutive 4-word block of memory. By writing and reading specific fields in a given device register, the processor may both issue commands and test device status and responses. μARM implements the full-handshake interrupt-driven protocol. Specifically:

1. Communication with device i is initiated by the writing of a command code into device i's device register.
2. Device i's controller responds by both starting the indicated operation and setting a status field in i's device register.
3. When the indicated operation completes, device i's controller will again set some fields in i's device register; including the status field. Furthermore, device i's controller will generate an interrupt exception by asserting the appropriate interrupt line. The generated interrupt exception informs the processor that the requested operation has concluded and that the device requires its attention.
4. The interrupt is acknowledged by writing the acknowledge command code in device i's device register.
5. Device i's controller will de-assert the interrupt line and the protocol can restart. For performance purposes, writing a new command after the interrupt is generated will both acknowledge the interrupt and start a new operation immediately.

The device registers are located in low-memory starting at `0x0000.0040`. As explained in section A.4.2, regardless of **CP15.R1**.M, all addresses between `0x0000.0200` and `DEVTOP` are interpreted as physical addresses. Furthermore, the device registers can only be accessed when the processor is executing in privileged mode.

The following table details the correspondence between device class/type and interrupt line.

Interrupt Line #	Device Class
0	Inter-processor interrupts
1	Processor Local Timer
2	Bus (Interval Timer)
3	Disk Devices
4	Tape Devices
5	Network (Ethernet) Devices
6	Printer Devices
7	Terminal Devices

Some important issues relating to device management:

- Since there are multiple interrupt lines, and multiple devices attached to the same interrupt line, at any point in time there may be multiple interrupts pending simultaneously; both across interrupt lines and on the same interrupt line.
- The lower the interrupt line number, the higher the priority of the interrupt. Note how fast/critical devices (e.g. disk devices) are attached to a high priority interrupt line while slow devices are attached to the low priority interrupt lines.
- Interrupt lines 3-7 are used for external devices. Interrupt lines 0-2 are for internally generated interrupts. Lines 0-1 are present for future multiprocessor support, but currently unused.
- Disk and tape devices support Direct Memory Access (DMA); that is through cooperation with the bus, these devices are able to transfer whole blocks of data to/from memory from/to the device. Data blocks must be both wordaligned and of multiple-word in size. μARM supports any number of concurrent DMA operations; each on a different device. Care must be taken to prevent simultaneous DMA operations on the same chunk of memory.
- After an operation has begun on a device, its device register "freezes" - becomes read-only - and will not accept any other commands until the operation completes.
- Any device register for an uninstalled device is "frozen" - set to zero - and subsequent writes to the device register have no effect.
- Device registers use only physical addresses; this includes addresses used in DMA operations.
- Each external device in μARM is identified by the interrupt line it is attached to and its device number; an integer in [0..7]. μARM limits the number of devices per interrupt line to eight.

- For performance reasons, devices in the same class are, by default, attached to the same interrupt line.

A.5.1 Device Registers

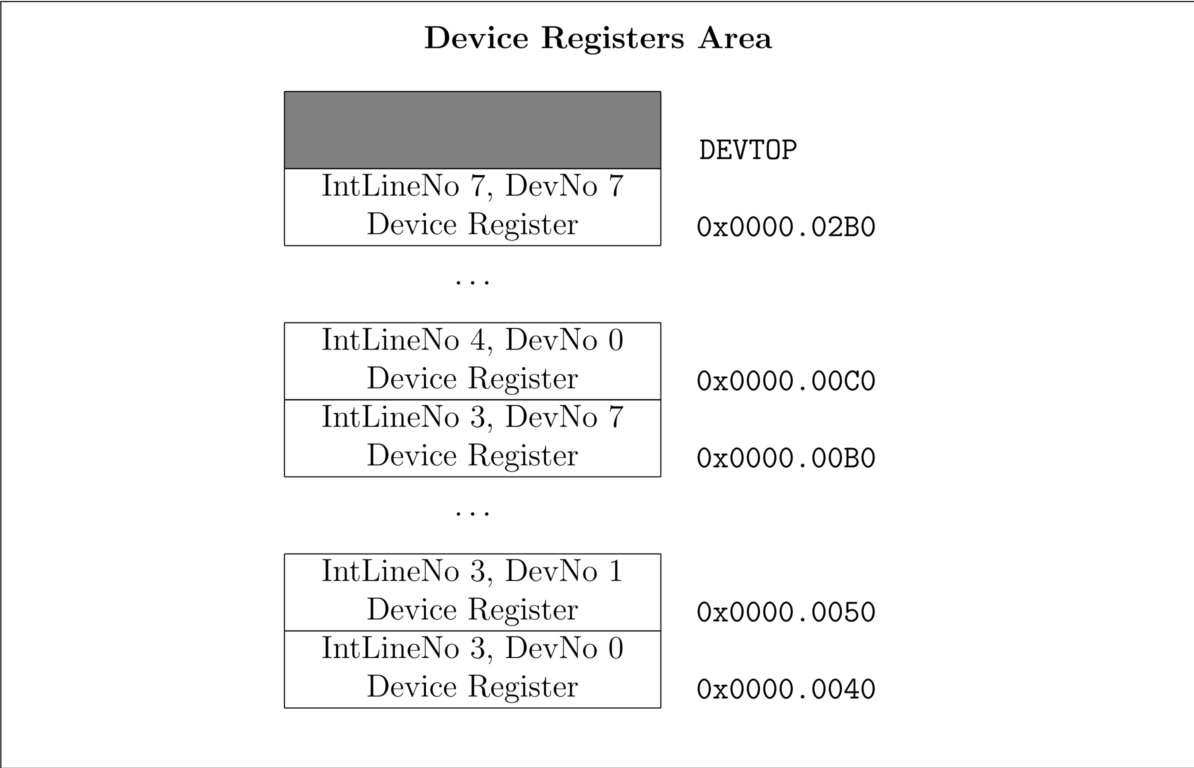
All external devices share the same device register structure.

While each device class has a specific use and format for these fields, all device classes, except terminal devices, use:

- **COMMAND** to allow commands to be issued to the device controller.
- **STATUS** for the device controller to communicate the device status to the processor.
- **DATA0** & **DATA1** to pass additional parameters to the device controller or the passing of data from the device controller.

Field #	Address	Field Name
0	(base) + `0x0`	**STATUS**
1	(base) + `0x4`	**COMMAND**
2	(base) + `0x8`	**DATA0**
3	(base) + `0xC`	**DATA1**

All 40 device registers in μARM are located in low memory starting at `0x0000.0040`.



Given an interrupt line (IntLineNo) and a device number (DevNo) one can compute the starting address of the device's device register:

devAddrBase = `0x0000.0040` + ((IntlineNo - 3) * `0x80`) + (DevNo * `0x10`)

A.5.2 The Bus Device and Interval Timer

The bus acts as the interface between the processor and the RAM, ROM, and all the external devices. In particular the bus performs the following tasks:

1. Management of the time of Day (TOD) clock and Interval Timer.
2. Arbitration among the interrupt lines, the devices attached to each interrupt line and the device registers.
3. Repository of basic system information.

Interval Timer

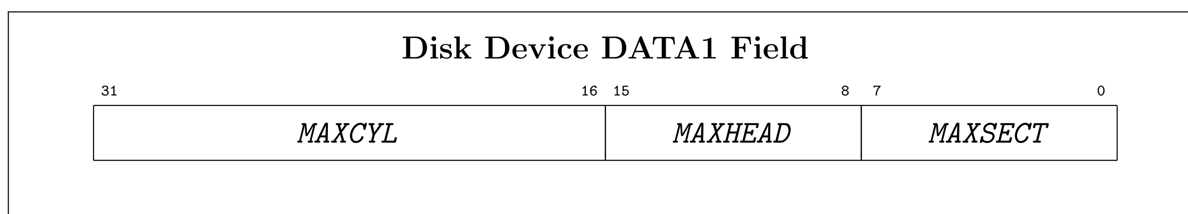
A read/writable unsigned word that is decremented by one every processor cycle and is set by μARM circuitry to `0xFFFF.FFFF` at system boot/reset time. The Interval Timer will generate an interrupt on interrupt line 2 whenever it makes the `0x0000.0000` $\rightarrow$

0xFFFF.FFFF transition. This is the only device attached to interrupt line 2, hence any interrupt on this line may be assumed to be associated with the Interval Timer. Interval Timer interrupts are acknowledged by writing a new value into the Interval Timer register.

The Interval Timer device register is located at 0x0000.02E4 (see sec. A.3.1).

A.5.3 Disk Devices

μARM supports up to eight DMA supporting read/writable hard disk drive devices. All μARM disk drives have a blocksize equal to the μARM framesize of 4KB. Each installed disk drive's device register **DATA1** field is read-only and describes the physical characteristics of the device's geometry.



μARM disk drives can have up to 65536 cylinders/track, addressed [0..(*MAXCYL*-1)]; 256 heads (or track surfaces), addressed [0..(*MAXHEAD*-1)]; and 256 sectors/track, addressed [0..(*MAXSECT*-1)]. Each 4KB physical disk block (or sector) can be addressed by specifying its coordinates: (cyl, head, sect).

A disk drive's device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Seek Error	Illegal parameter/hardware failure
5	Read Error	Illegal parameter/hardware failure
6	Write Error	Illegal parameter/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Status codes 1, 2, and 4-7 are completion codes. An illegal parameter may be an out of bounds value (e.g. a cylinder number outside of [0..(*MAXCYL*-1)]), or a non-existent physical address for DMA transfers.

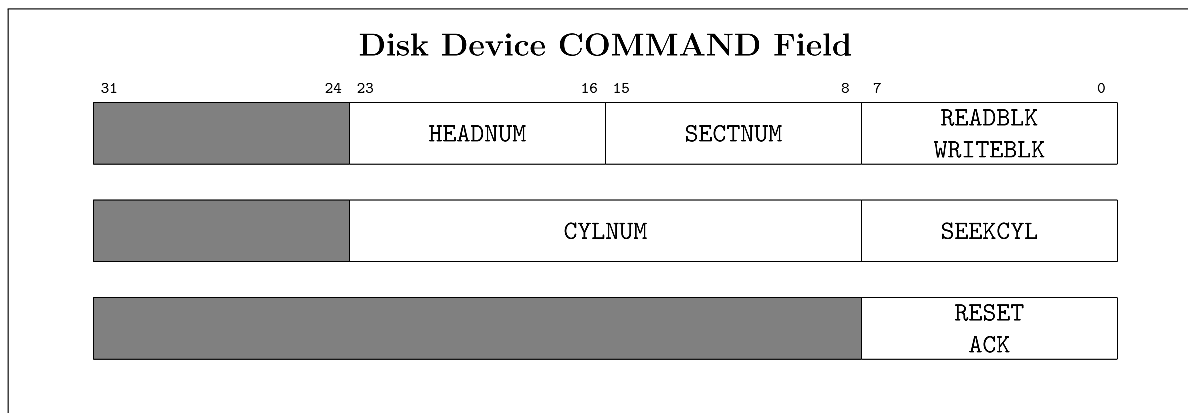
A disk drive's device register **DATA0** field is read/writable and is used to specify the starting physical address for a read or write DMA operation. Since memory is addressed

from low addresses to high, this address is the lowest word-aligned physical address of the 4KB block about to be transferred.

A disk drive's device register **COMMAND** field is read/writable and is used to issue commands to the disk drive.

Code	Command	Operation
0	RESET	Reset the device and move the boom to cylinder 0
1	ACK	Acknowledge a pending interrupt
2	SEEKCYL	Seek to the specified *CYLNUM*
3	READBLK	Read the block located at (*HEADNUM*, *SECTNUM*) in the current cylinder and copy it into RAM starting at the address in **DATA0**
4	WRITEBLK	Copy the 4KB of RAM starting at the address in **DATA0** into the block located at (*HEADNUM*, *SECTNUM*) in the current cylinder

The format of the **COMMAND** field, as illustrated in Figure 5.4, differs depending on which command is to be issued:



A disk operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Disk device performance, because both read and write operations are DMA-based, strongly depends on the system clock speed. While read/write throughput may reach MB's/sec in magnitude, the disk hardware operations remain in the millisecond range.

A.5.4 Tape Devices

μARM supports up to eight tape-removable, DMA supporting, read-only tape devices. All μARM tape devices support a blocksize of 4KB. Each installed tape device's register **DATA1** field is read-only and describes the current marker under the tape head when the device is idle.

Code	Marker	Meaning
0	EOT	End of Tape
1	EOF	End of File
2	EOB	End of Block
3	TS	Tape Start

A tape starts with a TS marker and ends with an EOT marker. It may be viewed as a collection of blocks, delimited by EOB markers, which are divided into files, delimited by EOF markers. An EOF marker acts as the EOB marker for the last block of the file and the EOT marker act as the EOF (and therefore also an EOB) marker for the last file on the tape.

When there is no tape cartridge loaded into the tape device, the **DATA1** field will contain the EOT marker, and the **STATUS** field will contain the Device Ready code. Since there is no tape cartridge present, the **COMMAND** field, though, will not accept any commands. Only when a tape is loaded does the device "wake up" and begin accepting commands. When a tape cartridge is loaded, the tape device rewinds the cartridge back to the TS marker.

A tape drive's device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Skip Error	Illegal command/hardware failure
5	Read Error	Illegal command/hardware failure
6	Back 1 Block Error	Illegal command/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Status codes 1, 2, and 4-7 are completion codes. An illegal parameter may be an attempt to read beyond the EOT marker or a non-existent physical address for DMA transfers.

A tape drive's device register **DATA0** field is read/writable and is used to specify the starting physical address for a DMA read operation. Since memory is addressed from low addresses to high, this address is the lowest word-aligned physical address of the 4 KB block about to be transferred.

A tape drive's device register **COMMAND** field is read/writable and is used to issue commands to the tape drive.

Code	Command	Operation
0	RESET	Reset the device and rewind the tape to TS marker
1	ACK	Acknowledge a pending interrupt
2	SKIPBLK	Forward the tape up to the next EOB/EOT
3	READBLK	Read the current block up to the next EOB/EOT marker and copy it into RAM starting at the address in **DATA0**
4	BACKBLK	Rewind the tape to the previous EOB/EOT marker

A tape operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Tape device performance, because read operations are DMA-based, strongly depends on the system clock speed. Tape read throughput can range from 2 MB/sec when the processor clock is set at 1 MHz, to over 4 MB/sec when the processor clock is bumped up to 99 MHz.

A.5.5 Network (Ethernet) Adapters

μARM supports up to eight DMA supporting network (i.e. Ethernet) adapters. Though these devices are DMA-based, they are not block devices. Network adapters operate at the byte level and transfer into/out of memory only the amount of data called for. Since packets on a network typically follow standard MTU sizes, this data should never exceed (by much) 1500 bytes.

Network adapters share some characteristics with terminal devices; they are simultaneously both an input device and an output device. As an output device, network adapters behave like other peripherals: a write command is issued and when the write (i.e. transmit) is completed, an interrupt is generated.

For packet receipt, there are two modes of operation:

- Interrupt Enabled: Whenever a packet arrives, an interrupt is generated - this interrupt is not the result of an earlier command. After ACK'ing this interrupt one issues a READNET command to read the packet. When the read is completed, another interrupt is generated, which itself must also be ACK'ed. In Interrupt Enabled mode, each incoming packet, when successfully read, is a two-interrupt sequence.

- Interrupt Disabled: When packets arrive, no interrupt is generated. The network adapter must be polled to determine if a packet is available. The READNET command is non-blocking, and returns 0 if there is no packet to be read. The READNET command will still generate an interrupt, which must be ACK'ed, upon its conclusion.

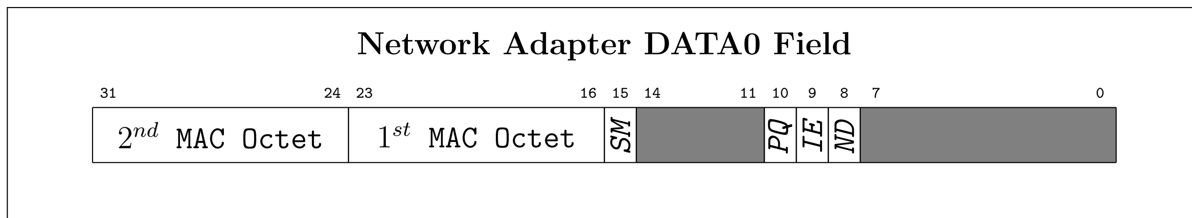
A network adapter's device register **STATUS** field is read-only and will contain one of the following status codes:

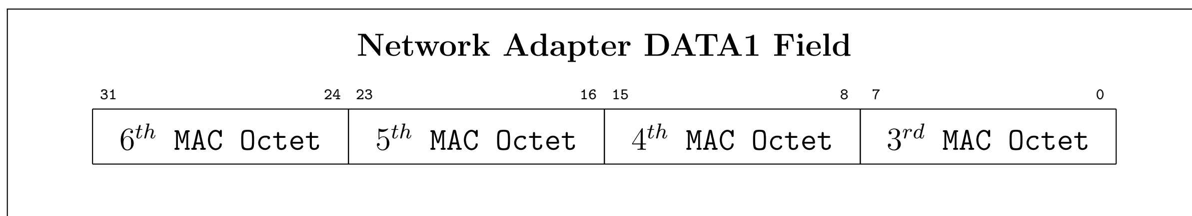
Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Readya	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
5	Read Error	Error reading packet from device
6	Write Error	Error attempt to send packet
7	DMA Transfer Error	Illegal physical address/hardware failure
128	Read Pending	Interrupts Enabled and packet present

Status codes 1, 2, and 5-7 are completion codes. An illegal address may be an out of bounds value or a non-existent physical address for DMA transfers.

Status code 128 is not a distinct status code, it is used in a logical OR fashion with the other status codes. Hence there are actually thirteen status values: 0, (1 & 129), (2 & 130),. . . , (7 & 135). For example, a status code value of 130 indicates that both an illegal operation was requested AND there is a packet pending for reading. The Read Pending status codes are only used when the network adapter is operating Interrupt Enable mode.

Code	Command	Operation
0	RESET	Reset the device and reset all configuration data to defaults
1	ACK	Acknowledge a pending interrupt
2	READCONF	Read configuration data into **DATA0** & **DATA1**
3	READNET	Read the next packet from the adapter and copy it into RAM starting at the address in **DATA0**
4	WRITENET	Send a packet of data starting at the RAM address in **DATA0**, whose length is in **DATA1**
5	CONFIG	Update adapter configuration data from values in **DATA0** & **DATA1**





The **DATA0** fields, during configuration operations (READCONF & CONFIG), are defined as follows:

- *ND* (NAMED, bit 8): When **DATA0**.*ND*=1, the network adapter will automatically fill all outgoing packets' source MAC address field with the network adapter's MAC address.

- *IE* (Interrupt Enable, bit 9): If **DATA0**.*IE*=1, whenever a packet is pending on the device (i.e. waiting to be read), it will immediately generate an interrupt. After ACK'ing this interrupt, one issues a READNET command to facilitate the reading of the packet. The READNET command must then also be ACK'ed.

- *PQ* (PROMISQ, bit 10): If **DATA0**.*PQ*=1 the network adapter will capture and save all packets its receives. When **DATA0**.*PQ*=0, the device will ignore/drop any packets not intended for its MAC address. Broadcast packets will still be received even when **DATA0**.*PQ*=0.

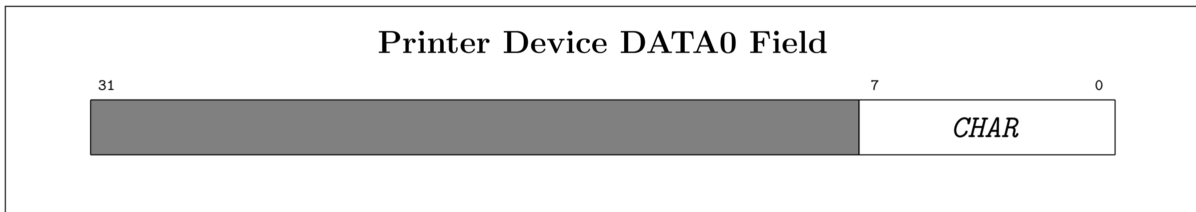
- *SM* (SetMAC, bit 15): When **DATA0**.*SM*=1 and a CONFIG command is issued, the MAC address of the adapter is updated to the values in **DATA0** & **DATA1**. When **DATA0**.*SM*=0 and a CONFIG command is issued, the adapter's MAC address remains unchanged.

As described above, the **DATA0** & **DATA1** fields are overloaded; either containing device status values or DMA addresses and lengths. One uses the CONFIG to set network adapter configuration values. Similarly, after a READNET or WRITENET operation, one can use a READCONF operation to reset the **DATA0** & **DATA1** registers to reflect the current adapter configuration values.

A.5.6 Printer Devices

μARM supports up to eight parallel printer interfaces, each one with a single 8-bit character transmission capability.

The **DATA0** field for printer devices is read/writable and is used to set the character to be transmitted to the printer. The character is placed in the low-order byte of the **DATA0** field. The **DATA1** field, for printer devices is not used.



A printer's device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Print Error	Error during character transmission

Status codes 1, 2, and 4 are completion codes.

A printer's device register **COMMAND** field is read/writable and is used to issue commands to the printer interface.

Code	Command	Operation
0	RESET	Reset the device interface
1	ACK	Acknowledge a pending interrupt
2	PRINTCHR	Transmit the character in **DATA0** over the line

A printer operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

The printer interface's maximum throughput is 125 KB/sec.

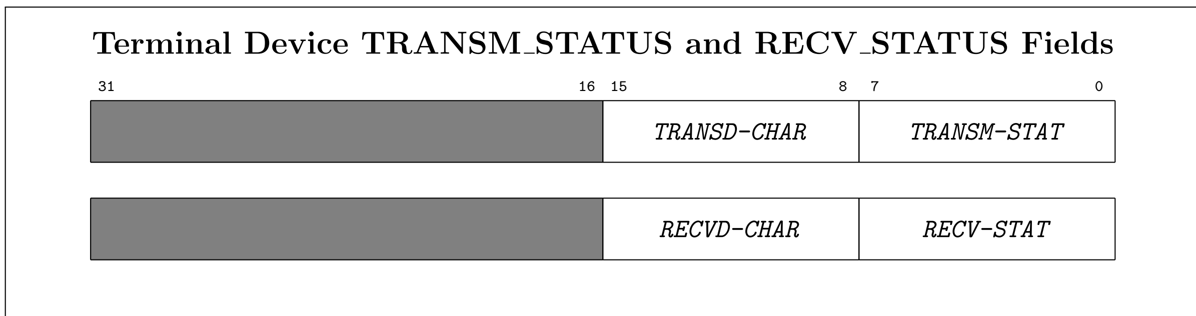
A.5.7 Terminal Devices

μARM supports up to eight serial terminal device interfaces, each one with a single 8-bit character transmission and receipt capability.

Each terminal interface contains two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. To support the two-subdevices a terminal interface's device register is redefined as follows:

Field #	Address	Field Name
0	(base) + 0x0	**RECV-STATUS**
1	(base) + 0x4	**RECV-COMMAND**
2	(base) + 0x8	**TRANSM-STATUS**
3	(base) + 0xc	**TRANSM-COMMAND**

The **TRANSM_STATUS** and **RECV_STATUS** fields (device register fields 0 & 2) are read-only and have the following format.



The *Status* byte has the following meaning:

Code	*RECV-STATUS*	*TRANSM-STATUS*
0	Device Not Installed	Device not installed
1	Device Ready	Device Ready
2	Illegal Operation Code Error	Illegal Operation Code Error
3	Device Busy	Device Busy
4	Receive Error	Transmission Error
5	Character Received	Character Transmitted

The meaning of status codes 0-4 are the same as with other device types. Furthermore:

- The Character Received code (5) is set when a character is correctly received from the terminal and is placed in **RECV_STATUS**.*RECV'D-CHAR*.
- The Character Transmitted code (5) is set when a character is correctlytransmitted to the terminal and is placed in **TRANSM_STATUS**.*TRANS'D-CHAR*.

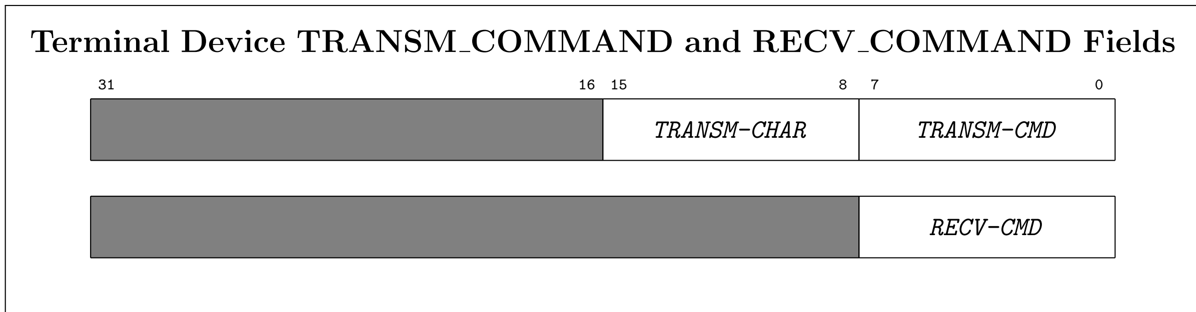
- The Device Ready code (1) is set as a response to an ACK or RESET command.

A terminal's **TRANSM_COMMAND** and **RECV_COMMAND** fields are read/ writable and are used to issue commands to the terminal's interface.

Code	*TRANSM-CMD*	*RECV-CMD*	Operation
0	RESET	RESET	Reset the transmitter or receiver interface
1	ACK	ACK	Ack a pending interrupt
2	TRANSMITCHAR	RECEIVECHAR	Transmit or Receive the character over the line

The **TRANSM_COMMAND** and **RECV_COMMAND** fields have the following format:

Terminal Device TRANSM_COMMAND and RECV_COMMAND Fields



RECV_COMMAND.*RECV-CMD* is simply the command.
The **TRANSM_COMMAND** field has two parts; the command itself (**TRANSM_COMMAND**.*TRANSM-CMD*) and the character to be transmitted (**TRANSM_COMMAND**.*TRANSM-CHAR*).

A character is received, and placed in **RECV_STATUS**.*RECV'D-CHAR* only after a RECEIVECHAR command has been issued to the receiver.

The operation of a terminal device is more complicated than other devices because it is two sub-devices sharing the same device register interface. When a terminal device generates an interrupt, the (operating system's) terminal device interrupt handler, after determining which terminal generated the interrupt, must furthermore determine if the interrupt is for receiving a character, for transmitting a character, or both; i.e. two interrupts pending simultaneously.

If there are two interrupts pending simultaneously, both must be acknowledged in order to have the appropriate interrupt pending bit in the Interrupt Line 7 Interrupting Devices Bit Map turned off.

To make it possible to determine which sub-device has a pending interrupt there are two sub-device "ready" conditions; Device Ready and Character Received/Transmitted. While other device types can use a Device Ready code to signal a successful

completion, this is insufficient for terminal devices. For terminal devices it is necessary to distinguish between a state of successful completion though the interrupt is not yet acknowledged, Character Received/Transmitted, and a command whose completion has been acknowledged, Device Ready.

A terminal operation is started by loading the appropriate value(s) into the **TRANSM_COMMAND** or **RECV_COMMAND** field. For the duration of the operation the sub-device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set in **TRANSM_STATUS** or **RECV_STATUS** respectively; "Character Transmitted/Received" for successful completion or one of the error codes. The interrupt is acknowledged by issuing an ACK or RESET command to which the sub-device responds by setting the Device Ready code in the respective status field.

The terminal interface's maximum throughput is 12.5 KB/sec for both character transmission and receipt.

A.6 BIOS & System Library

A.6.1 BIOS

Bootstrap Function

The bootstrap program bundled with μARM installation initializes hardware facilities and starts execution. The code for the bootstrap routine can be found in the file `BIOS.s`.

The BIOS code performs the following operations at boot time:

1. Populate of the Exception Vector with Branch instructions to ROM Level Exception Handlers;
2. Set default Exception States Vector entries with `Branch to PANIC` instructions;
3. Retrieve Entry Point from core binary code loaded in RAM;
4. Set execution mode to System mode with ARM ISA and all interrupts enabled;
5. Set Exit Point and `RAMTOP` value;
6. Clear all used scratch registers;
7. Jump to Entry Point.

`BIOS.s` source file also contains the code for ROM Level Exception Handlers (see Sec. A.2.5) and ROM Level Services; the default installation directory of this and the other support files is `/usr/include/uarm/`.

ROM Level Services

ROM Level Services are requested by issuing a `SWI` instructions with certain parameters and are served by the BIOS code:

Halt

By executing `SWI #1`, the BIOS will print "`SYSTEM HALTED.`" on Terminal 0 and shut down the virtual machine.

Panic

By executing `SWI #2`, the BIOS will print "`KERNEL PANIC.`" on Terminal 0 and enter an infinite loop.

LDST

A `SWI #3` instructions will begin the loading of the processor state stored at the address loaded into **a1** register to actual processor's registers, checking destination mode and setting only the right register window.

Wait

By executing `SWI #4`, the BIOS will put the machine in IDLE state waiting for an interrupt to wake up the system up.

System Calls / Breakpoints

If a `SWI #8` or `SWI #9` instructions is executed, the syscall handler passes up the call, setting the right cause in **CP15.Cause** register.

A.6.2 System Library

System library is provided by *libuarm*, it offers a set of methods to access low level functionalities from *C* language:

`tprint(char *s)`

Print a '\0' terminated array of chars to Terminal 0. This function uses busy waiting to wait for the device to be ready.

`HALT()`

Request BIOS Halt service.

`PANIC()`

Request BIOS Panic service.

`WAIT()`

Request BIOS Wait service.

`LDST(void *addr)`

Request BIOS LDST service setting **a1** to `addr` value.

`STST(void *addr)`

Stores the actual processor state in state_t structure pointed by *addr*.

`SYSCALL(unsigned int sysNum, unsigned int arg1, unsigned int arg2, unsigned int arg3)`

Generates a software exception leading to kernel defined Syscall handler. **a1** is loaded with `sysNum`, **a2** is loaded with `arg1`, **a3** is loaded with `arg2` and **a4** is loaded with `arg3`

`BREAK(unsigned int arg0, unsigned int arg1, unsigned int arg2, unsigned int arg3)`

Generates a software exception leading to kernel defined Breakpoint handler. **a1** is loaded with `arg0`, **a2** is loaded with `arg1`, **a3** is loaded with `arg2` and **a4** is loaded with `arg3`

`getSTATUS() / setSTATUS()`

Manipulate Current Program Status Register (**CPSR**).

`getCAUSE() / setCAUSE()`

Manipulate Exception/Interrupt Cause register (**CP15.R15**).

`getTIMER() / setTIMER()`

Manipulate Interval Timer.

`getTODHI() / getTODLO()`

Returns the upper/lower part of Time of Day 64-bit register.

`getCONTROL() / setCONTROL()`

Manipulate System Control Register (**CP15.SCB**).

`getTLB_Index()` / `setTLB_Index(unsigned int index)`

Manipulate TLB Index register.

`getTLB_Random()`

Returns TLB Random register.

`getEntryHi()` / `setEntryHi(unsigned int hi)` / `getEntryLo()` / `setEntryLo(unsigned int lo)`

Manipulate Page Table Entry Hi and Entry Low registers (**CP15.R2.EntryHi** and **CP15.R2.EntryLow**).

`getBadVAddr()`

Returns Faulting Address register (**CP15.R6**).

`TLBWR()`

Write the contents of **CP15.R2** to the TLB slot indicated by TLB Random register value.

`TLBWI()`

Write the contents of **CP15.R2** to the TLB slot indicated by TLB Index register value.

`TLBR()`

Read the contents of TLB slot indicated by TLB Index register value to **CP15.R2** register.

`TLBP()`

Scan the TLB searching for a pair that matches *VPN* in and *ASID* in **CP15.R2.EntryHi** or that has *G* flag set in **CP15.R2.EntryLo** and is *V*alid, if a match is found, its index in the TLB cache is stored as TLB Index register value, otherwise that register will have the most significant bit set to 1.

TLBCLR()

Set all TLB contents to 0.

Additional Libraries

Two more simple libraries are provided in addition to the system library:

`ulibuarm`

A simple subset of `libuarm` to be used by user mode programs, only exposes a mean to request system calls through the same `SYSCALL` instruction; see Section A.6.2.

`libdiv`

This library implements integer division and module operations that are not provided by the processor instruction set, it must be linked together with any program that uses divisions.

A.7 Emulator Usage

The following sections will cover the usage of the emulator itself, along with the necessary tools required for compiling the programs to be run, debugging functionalities provided by μARM and the support tool `uarm-mkdev` used to create device files needed for advanced usage.

A.7.1 Compiling and Running the Machine

μARM is an ARM7tdmi-based system emulator; in order to build a program for the correct ARM architecture, an ARM compiler is needed. Once a valid executable file is ready, the machine must be configured to load the proper core file (and optionally BIOS file). At last the machine is run and the output is read from terminal screens or printed files.

C Language Development for μARM

Run time C-library support utilities are obviously not available. This includes I/O statements (e.g. `printf` from `stdio.h`), storage allocation calls (e.g. `malloc`) and file manipulation methods. In general any C-library method that interfaces with the operating system is not supported; μARM does not have an OS to support these calls - unless you write one to do so. The `libuarm` library, described in Section A.6.2, is the only support library available.

The μARM linker requires a small function, named `__start()`. This function is to be the entry point to the program being linked. Typically `__start()` is provided from system library and will initialize some registers and then call `main()`. After `main()` concludes, control is returned to `start()` which should perform some appropriate termination service. Two such functions, written in assembler, are provided:

- `crtso.o`: This file is to be used when linking together the files for the kernel/OS. The version of `__start()` in this file simply runs the `main()` function of the program (i.e. kernel), assuming it is loaded in RAM beginning at `0x0000.8000`.
- `crti.o`: This file is to be used when linking together the files for individual U-proc's. The version of `__start()` in this file assumes that the program's (i.e. U-proc's) header has 0x8000.0000 as its starting (virtual) address. Some registers are stored and restored before and after the `main()` call. `__start()` assumes that the kernel will initialize $SP. The last instruction if the `__start()` routine is the peaceful termination of the program using the dedicated Syscall. As the Syscall numbering is implementation dependent, the value at the beginning of this file needs to be modified accordingly from the OS developer.

Compiling

`arm-none-eabi` toolchain will be taken into example to explain the compiling process, but any ARM cross-toolchain, or any toolchain run on an ARM system should be able to generate proper code for μARM execution.

To be sure the compiler will not include the host system's libraries (see Sec. A.7.1), the `-c` option is necessary while compiling each source file, as well as the `-mcpu=arm7tdmi` to ensure maximum compatibility with the system.

Once each source file has been compiled into an object file, everything has to be linked together using the provided start files (`crtso.o` and `crti.o`) and the `-T` option to select the right memory map:

- `elf32ltsarm.h.uarmcore.x` is the memory map used for kernel binaries, which are meant to be executed with virtual memory turned off;
- `elf32ltsarm.h.uarmaout.x` is the memory map used for Uproc's binaries, which are meant to be executed with virtual memory enabled.

Compiling an Operating System

Take as an example a program composed of the following modules:

- `core.c`, `core.h`: core module, uses `libuarm` library;
- `service.c`, `service.h`: library implementing service functions;
- `test.c`: test module to check program behavior.

To build such a program using `arm-none-eabi` toolchain one should execute the following commands:

```
arm-none-eabi-gcc -c -mcpu=arm7tdmi core.c -o core.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi service.c -o service.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi test.c -o test.o

arm-none-eabi-ld -T \\
/usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x \\
/usr/include/uarm/crtso.o /usr/include/uarm/libuarm.o \\
core.o service.o test.o -o kernel
```

`/usr/include/uarm/` is the default installation directory for support files, in this example, and in the ones that will follow, this location will be considered as valid and filled with the provided support files. The order of the object files in the linking (last) command is important: specifically, the first two support files must be in their respective positions.

Compiling a Uproc file

Take as an example a user mode program that one wishes to run on an already existing μARM operating system: `uproc.c`. This program will use system calls as well as integer divisions, respectively provided by `ulibuarm` and `libdiv` libraries.

To build such a program using `arm-none-eabi` toolchain one should execute the following commands:

```
arm-none-eabi-gcc -c -mcpu=arm7tdmi uproc.c -o uproc.o

arm-none-eabi-ld -T
/usr/include/uarm/ldscripts/elf32ltsarm.h.uarmaout.x \\
/usr/include/uarm/crti.o /usr/include/uarm/ulibuarm.o \\
/usr/include/uarm/libdiv.o uproc.o -o uproc
```

Finally, this executable file can be (optionally) loaded onto a tape cartridge with the following command:

```
uarm-mkdev -t uproc.uarm uproc
```

which produces the preloaded "tape cartridge" file: `uproc.uarm` (for further details see Sec. A.7.2).

Compiling BIOS ROM

ROM code development must be done in ARM assembler. Consider the case where one has a new version of the execution time ROM routines: `testrom.s`.

One should assemble the source file using the command:

```
arm-none-eabi-gcc -mcpu=arm7tdmi -c -fPIC testrom.s -o testrom
```

Note the use of the `-fPIC` option to generate position independent code. (i.e. No relocations)

Settings Breakdown

All the possible configurations for the simulation execution are accessible through the main settings window inside μARM. To get show the settings window simply click on the *Machine Config Button* (the first button in the main bar with the Screwdriver and Wrench icon).

The configurations are split into three categories: *General*, *Devices* and *Accessibility*.

General Settings

The first tab provides configurations for the main simulator features and is divided into three categories:

- *Hardware* - These settings modify core hardware structure
  - Default Clock Rate (MHz): This value represents the simulated clock speed of the processor, it does not influence the simulation speed but the external devices will take this value into account to establish the cycles count needed to complete a task.
  - RAM Size (Frames): This value represents the actual size of installed RAM expressed in ram Frames, a label next to the input field will show the corresponding value in Bytes.
  - TLB Size (Entries): This value represents the number of Entries (i.e. the size) of the Translation Lookaside Buffer (see Sec. A.4.2).
- *Firmware and Software* - These settings specify options regarding the Firmware binaries and the Core/Bootstrap binaries.
  - Execution ROM: The ARM ELF file specified by this setting will be loaded into the Execution ROM and will be executed at boot time, the default value is `/usr/include/uarm/BIOS`.
  - Core file: The ARM ELF file specified by this setting will be loaded into RAM starting from address `0x0000.8000` and will be automatically loaded from default BIOS after the startup routines (see Sec. A.6.1).
- *Debugging Support* - These settings modify the behavior of some debugging tools included into the simulator
  - Enable constant refresh: If enabled, the contents of the debugger components will be automatically updated during the machine running time, otherwise the contents will be updated each time the execution is paused.
  - GUI Refresh Rate: If constant refresh is enabled, this value represents the number of processor cycles that will elapse between each GUI update.
  - Pause execution on Exception: If enabled, the execution will pause each time an exception is raised, i.e. each time the **pc** value is below `0x0000.0020`.
  - Symbol Table ASID: Sets the default ASID for associated with loaded Symbol Table in the Breakpoint Window.
  - External Symbol Table: If enabled, the ARM ELF file specified will be the one from which the Symbol Table loaded in Breakpoint and Data Structures windows is generated.

Devices Settings

Settings for external devices are accessible from this tab. The main drop-down menu is used to select the type of device to configure, then the main contents will change accordingly, showing the settings for each one of the eight devices of the selected type.

Each single device has a relative "Enable" checkbox that will affect its presence into the simulation.

Device specific configurations are:

- *Tapes*
  - Device File: A tape file created with `uarm-mkdev` tool to be loaded into the tape drive
- *Disks*
  - Device File: A hard disk file created with `uarm-mkdev` tool to be attached to the system
- *Terminals*
  - Device File: A text file onto which the simulator will dump all the terminal contents
- *Printers*
  - Device File: A text file onto which the printer will write its output
- *Network Interfaces*
  - Device File: The path of the vde_switch that will be created and will be used from the network interface
  - Fixed MAC address: If enabled, the network interface MAC address will be fixed to a user defined value.
  - MAC address: If fixed MAC address is enabled, this will be the MAC address of the network interface.

Accessibility Settings

Settings related to accessibility features are accessible through this tab. At the time of writing the only available option is *Enable Increased Accessibility.* When enabled and

after a program restart, this setting will change the graphical interface in favor of a set of much accessible widgets.

As an example, the main processor viewer window, that is implemented with a matrix of text labels, will be replaced by an array of text fields, each one displaying a full "window" of registers (see Sec. A.2.1).

Global Settings

Global interface settings are stored in `/etc/default/uarm` in Unix-based systems. This file is used to choose the preferred font face and size to be used by μARM for displaying registers and Bus contents in its graphical user interface.

The default `Monospace` font face should be available in some form in any system, but these settings could be some times necessary to fix display problems. The default size leads to a correct sizing of the emulator windows most of the times, but if there are sizing issues (windows/fonts too big/small), this is the value that needs to be adjusted.

Configuration File Fields

All configurations are stored in a *Json* file, the default location for this file is `$HOME/.config/uarm/machine.uarm.cfg`, where `$HOME` is the current user's home directory. Using the `-c` command line option (see Sec. A.7.1) a different configuration file can be specified. If the configuration file is missing (both the default file or a user selected one), a new file with default settings values is created in its place.

A configuration file can be directly modified outside of μARM with any text editor and will be recognized by the machine if all fields are valid. Any missing field will be initialized by the program with its default value.

The structure of a simple configuration file is shown below:

`machine.uarm.cfg` File Format

```
{
    "accessible-mode": false,
    "clock-rate": 1,
    "core-file": "kernel",
    "devices": {
        "disk0": {
            "enabled": false,
            "file": "disks/disk0.uarm"
        },
        "eth0": {
            "address": "ba:98:76:54:32:10",
            "enabled": false,
            "file": "tap0"
        },
        "terminal0": {
            "enabled": true,
            "file": "term0.uarm"
        }
    },
    "execution-rom": "/usr/include/uarm/BIOS",
    "num-ram-frames": 10240,
    "pause-on-exc": false,
    "pause-on-tlb": false,
    "refresh-on-pause": false,
    "refresh-rate": 600,
    "symbol-table": {
        "asid": 0,
        "external-stab": false,
        "file": ""
    },
    "tlb-size": 16
}
```

Each field of the configuration file refers to a setting that can be reached through the Main Settings Window:

Field	Type	Settings' Tab	Setting Name
`accessible-mode`	`bool`	Accessibility	Enable Increased Accessibility
`clock-rate`	`int`	General	Default Clock Rate
`core-file`	`string`	General	Core file
`devices: {dev}[0-7]: enabled`	`bool`	Devices	Enable
`devices: {dev}[0-7]: file`	`string`	Devices	Device File
`devices: eth[0-7]: address`	`string`	Devices	Fixed MAC address & MAC address
`execution-rom`	`string`	General	Execution ROM
`num-ram-frames`	`int`	General	RAM Size
`pause-on-exc`	`bool`	General	Pause execution on Exception
`pause-on-tlb`	`bool`	Breakpoints[1]	Stop on TLB change
`refresh-on-pause`	`bool`	General	Enable constant refresh
`refresh-rate`	`int`	General	GUI Refresh Rate
`symbol-table: asid`	`int`	General	Symbol Table ASID
`symbol-table: external-stab`	`bool`	General	External Symbol Table (checkbox)
`symbol-table: file`	`string`	General	External Symbol Table (line edit)
`tlb-size`	`int`	General	TLB Size

Where {`dev`} is one of the following:

{`dev`}	Device
`disk`	Disks
`eth`	Network
`printer`	Printers
`tape`	Tapes
`terminal`	Terminals

Terminal Windows

Once the machine is powered on, a dedicated window is accessible for each enabled terminal through the *Terminals* sub menu.

[1]This setting can be found in the Breakpoints window, see Section A.7.3.

Each Terminal Window shows the terminal contents, both input and output, and has a control bar at the bottom. Through the control bar the user can simulate a hardware failure to test special system features and see a hidden status bar through the *Show Status* button.

The terminal status shows the value of the terminal transmitter (`TX`) and receiver (`RX`) sub-devices status words along with the **TOD** value relative to the last update of the status registers.

Command Line Options

A small set of additional options are available by running the program from a command line. The full launch command synopsis is:

```
uarm [-c <config.conf>] [-e [-x]] [--dumpExec <dumpfile>]
```

Where:

- `-c <config.cfg>` is used to load/create a user defined configuration file (see Sec. A.7.1).
- `-e` enables *Autorun*: the machine is powered on and started with the program start.
- `-x` enables *Run and Exit*: if Autorun is enabled, exit the program when the machine reaches the `HALT` instruction (see Sec. A.6.2).
- `--dumpExec <dumpfile>` enables *Execution Dump*: the file `<dumpfile>` is filled with every binary instruction executed by the machine at run time and the relative decompiled assembly code. The file `<dumpfile>` is overwritten if already existent.

Binary Formats

The cross-compiler and cross-linker generate code in the *Executable and Linking Format* (ELF). While the ELF format allows for efficient compilation and execution by an OS it is also quite complex. Using the ELF format would therefore un-necessarily complicate the student OS development process since there are no program loaders or support libraries available until one writes them.

Hence μARM converts on the fly the executable core files in a simpler format, based on the predecessor to the ELF format: *a.out*. User mode programs are converted in a.out format as well as they get loaded on tape by the tool `uarm-mkdev`.

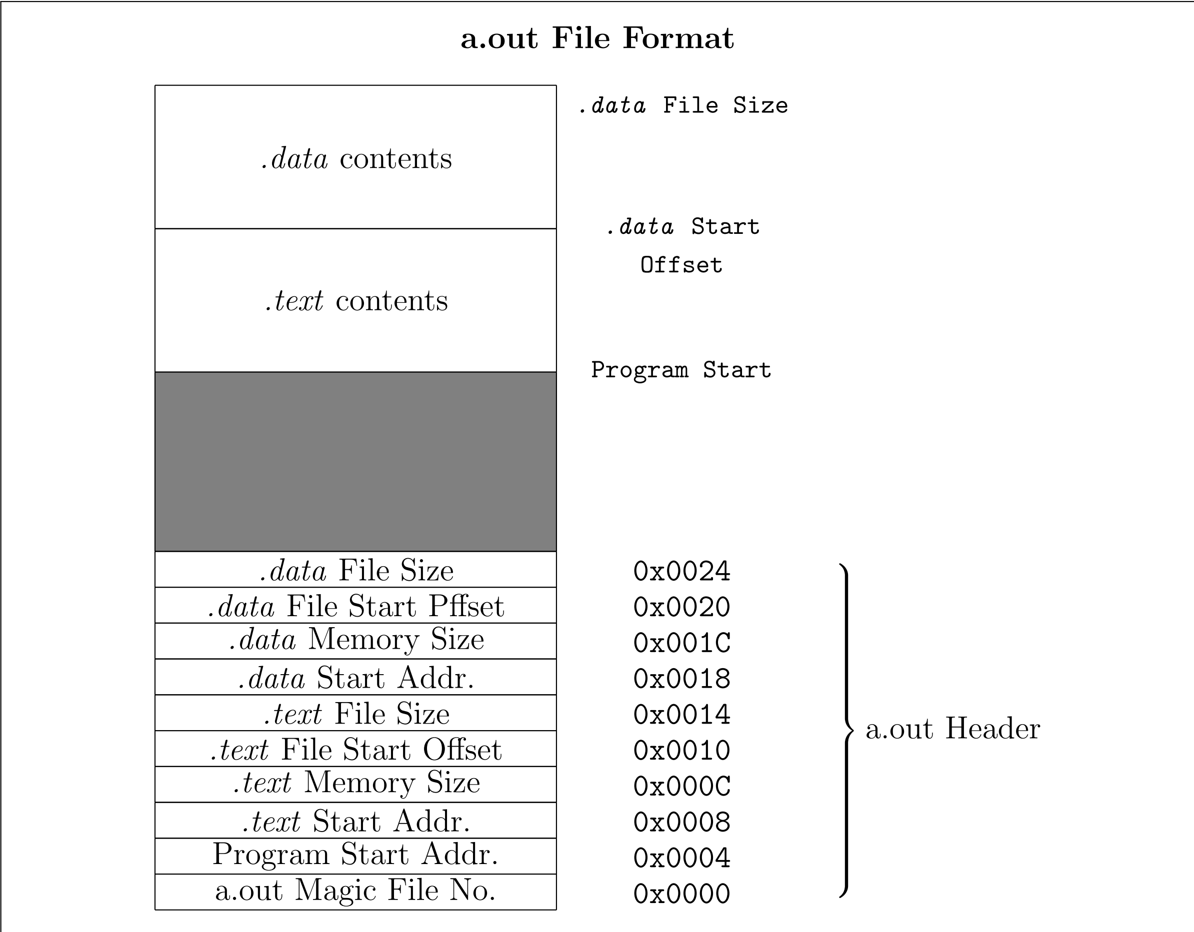
The a.out Format

A program, once compiled and linked may be logically split into two areas or sections. The primary areas are:

- *.text*: This area contains all the compiled code for the executable program. All of the program's functions are placed contiguously one after another in the order the functions are presented to the linker.
- *.data*: This area contains all the global and static variables and data structures. It in turn is logically divided into two sub-sections:
  - *.data*: Those global and static variables and data structures that have a defined (i.e. initialized) value at program start time.
  - *.bss*: Those global and static variables and data structures that do NOT have a defined (i.e. initialized) value at program start time.

Local, i.e. automatic, variables are allocated/deallocated on/from the program's stack, while dynamic variables are allocated from the program's *heap*. A heap, like a stack, is an OS allocated segment of a program's (virtual) address space. Unlike stack management, which is dealt with automatically by the code produced by the compiler, heap management is performed by the OS. The compiler can produce stack management code since the number and size of each function's local variables are known at compile time. Since the number and size of dynamic variables cannot be known until run-time, heap management falls to the OS. Heap management can safely be ignored by OS authors who are not supporting dynamic variables, i.e. there are no `malloc`-type SYSCALLs.

a.out Header		
Field Name	**File Offset**	**Field Description**
a.out Magic File No.	`0x0000`	Special identifier used for file type recognition.
Program Start Addr.	`0x0004`	Address (virtual) from which program execution should begin. Typically this is `0x0000.8074` fore kernel and `0x8000.0094` for user mode programs.
.text Start Addr.	`0x0008`	Address (virtual) for the start of the .text area. It is fixed to `0x0000.0000` for core and `0x8000.0000` for user mode programs.
.text Memory Size	`0x000C`	Size of the memory space occupied by the .text section.
.text File Start Offset	`0x0010`	Offset into a.out file where *.text* begins. Since the header is part of *.text*, this is always `0x0000.0000`
.text File Size	`0x0014`	Size of *.text* area in the a.out file. Larger than *.text* Mem. Size since its padded to the nearest 4KB block boundary.
.data Start Addr.	`0x0018`	Address (virtual) for the start of the *.data* area. The *.data* area is placed immediately after the *.text* area at the start of a 4KB block, i.e. *.text* Start Addr. + *.text* File Size.
.data Memory Size	`0x001C`	Size of the memory space occupied by the full *.data* area, including the *.bss* area.
.data File Start Offset	`0x0020`	Offset into the a.out file where *.data* begins. This should be the same as the *.text* File Size.
.data File Size	`0x0024`	Size of *.data* area in the a.out file. Different from the *.data* Memory Size since it doesn't include the *.bss* area but is padded to the nearest 4KB block boundary.



Important Point: The *.data* area is given an address space immediately after the *.text* address space, aligned to the next 4KB block, insuring that *.text* and *.data* areas are completely separated. The *.bss* area immediately follows the *.data* area and is NOT aligned to a separate 4KB block.

.text and *.data* Memory Sizes are provided for sophisticated memory allocation purposes:

- The size of each U-proc's PgTbl can be determined dynamically, instead of "one size fits all" approach.
- PTE's that represent the *.text* area can be marked as read-only, while entries that represent the *.data* area can be marked as writable.

The program loader which reads in the contents of a U-proc's a.out file, needs to be aware that the *.text* and *.data* areas are contiguous and have a starting virtual address of `0x8000.0000`. The *.bss* area, while not explicitly described in the a.out file, will occupy the virtual address space immediately after the *.data* area. Zero'ing out the *.bss* area

will insure that all uninitialized global and static variables and data structures begin with an initial value of zero. Finally, the loader loads the **pc** with the Program Start Addr.; i.e. the contents of the second word of the a.out program header.

a.out files have padded *.text* and *.data* sections to facilitate file reading/loading. Each section is padded to a multiple of the frame size or disk and tape block size. This allows the kernel/OS to easily load the program and insure that the program's *.text* and *.data* occupy disjoint frame sets.

A.7.2 `uarm-mkdev` tool

While the log files for holding terminal and printer output are standard text files, and which if not present for any active printer or terminal, will be automatically created by μARM at startup time, the disk and tape cartridge files must be explicitly created beforehand. One uses the `uarm-mkdev` device creation utility to create the files that represent these persistent memory devices.

Creating Disk Devices

Disks in μARM are read/write sealed devices with specific performance figures. The `uarm-mkdev` utility allows one to create an empty disk only; this way an OS developer may elect any desired disk data organization.

The created "disk" file represents the entire disk contents, even when empty. Hence this file may be very large. It is recommended to create small disks which can be used to represent a little portion of an otherwise very large disk unit.

Disks are created via:

```
uarm-mkdev -d <diskfile.uarm> [cyl [head [sect [rpm [seekt [datas]]]]]]
```

where:

- `-d` instructs the utility to build a disk file image.
- `<diskfile.uarm>` is the name of the disk file image to be created.
- The following six optional parameters allow one to set the drive's geometry: number of cylinders, heads/surfaces, and sectors, and the drive's performance statistics: the disk rotation speed in rotations per minute, the average cylinder-to-cylinder seek time, and the sector data occupancy percentage.

As with real disks, differing performance statistics result in differing simulated drive performance. e.g. A faster rotation speed results in less latency delay and a smaller sector data occupancy percentage results in shorter read/write times.

The default values for all these parameters are shown when entering the `uarm-mkdev` command alone without any parameters.

Creating Tape Cartridges

Tape devices in μARM are read-only devices which are typically used for the fast loading of large quantities of data into the simulation without having to resort to typing the data directly into a terminal. Tapes are typically used to load user programs (Uproc's).

A tape cartridge file image will contain a properly-formatted copy of the file(s) the user wishes loaded onto it.

Tape cartridge image files are created via:

```
uarm-mkdev -t <tapefile.uarm> <file> [<file>] ... [<file>]
```

where:

- `-t` instructs the utility to build a tape cartridge file image.
- `<tapefile.uarm>` is the name of the tape cartridge file image to be created.
- The concluding space-separated list of `<file>` names are the files that will be included on the tape cartridge file image. These files, of which there must be at least one, executable ARM ELF files. Each file will be zero-padded to a multiple of the 4KB block size and sliced up using the EOB and EOF block markers. The tape's end will be marked with a EOT marker.

A.7.3 Debugging

Some more tools can be accessed through the main interface bar. These tools are meant to be used during run time to debug the program running on μARM.

All the contents shown by viewers gets updated each time the execution is paused or when the emulation speed is limited to low (i.e. non maximum) values. In addition to these updates, the viewers' contents can also be updated programmatically by toggling the *Enable constant refresh* setting (see Sec. A.7.1).

Registers Contents View

The main window shows the contents of all processor registers, along with the relevant coprocessor registers and some memory mapped system information, see Sections A.2.1, A.2.2 and A.3.1 for further explanations of each register.

On top of the processor registers matrix is shown the Execution Pipeline and the assembly translation of the currently executing operation.

This viewer gives some useful information about the current status of the machine, along with the first four arguments of each function call that the compiler usually stores in **a1-a4** registers.

Breakpoints

The *Breakpoint* button in the main bar shows the relative window. Through the breakpoint window the user can set any number of breakpoints by specifying the memory address that the machine will have to look for to pause execution.

Each time the value of the **pc** register equals any of the set breapoints, and the *Stop on Breakpoint* option is enabled, the execution is paused and all the matching breakpoints are highlighted.

Breakpoints can be automatically set at the beginning of a function by selecting the function name from the top list and clicking the *Add* button, or they can be set at any custom address by specifying the desired value by hand in the *Address* field. Breakpoints can be individually disabled by toggling the checkbox near their identifier.

ASID field is used when virtual memory is enabled, in this case the breakpoints will trigger only if the desired virtual address is reached with the specific ASID set in **CPSR.R2.EntryHy**.*ASID*.

Stop on TLB change will enable automatic execution pause each time the content of the TLB (see Sec. A.4.2) changes.

One common strategy for debugging with breakpoints is to add one or more debugging functions with empty bodies to be placed into the suspect function code and set breakpoints on the debugging functions to check the execution flow. A more complex approach, that requires a bit of ARM assembly understanding, is to decompile the target binary using the cross toolchain (an example command is shown below) and manually setting breakpoints at interesting points inside function body.

```
arm-none-eabi-objdump -d kernel | less
```

Bus Inspector

The *Memory* menu gives access to the *Bus Inspector* tool. The inspector permits to investigate the contents of the System Bus, beginning with Exception Vector, up to the actual RAM memory (see Sec. A.3.1 and A.4).

To view the contents of a specific bus region, the bus area can be specified in two ways:

- inserting start and end addresses, or
- inserting start address and area size.

Both input methods require the beginning address of the bus region; to specify a size, the *+* button must be enabled. Addresses and sizes must be inserted in hexadecimal format.

Once the area has been selected, clicking the *Display Portion* button will extend the inspector window showing the bus contents. The bus region maximum size allowed by

the Bus Inspector is 10KB, if a larger area is requested, an alert message will inform the user that it is not possible to show a bus portion bigger than 10KB.

A text field below the address bar is left for the user to note down a friendly name for the shown bus region. If the name and the region match the ones of a known object found in the symbol table, the address will be updated across machine resets (i.e. if the program is modified and built again between two runs and a global data structure changes its position in memory or size, the inspector will change the addresses accordingly).

The last component of the Bus Inspector tool window is the bus area contents viewer: the memory addresses are displayed in the leftmost column, the central column shows the raw hexadecimal contents and on the right is shown the decompiled ARM code.[2]

Unlike the other debugging tools, the number of Bus Inspector windows is not limited to one. Given the nature of this tool, it can be useful for the programmer to keep multiple inspectors open to check different bus regions simultaneously.

Structures Viewer

The second entry in *Memory* menu is *Structures Viewer*. This tool shows a list of all the global data structures[3] present in the loaded symbol table along with their contents.

Once the desired object has been chosen from the top table, its memory contents are shown in the lower section of the window in the same fashion as Bus Inspector (see Sec. A.7.3).

The *Show in Bus Inspector* button opens a new Bus Inspector window preloaded with address, size and name of the active object, this way the Bus Inspector will keep the address and size updated across machine resets (see Sec. A.7.3).

TLB Viewer

The *TLB* button in the main bar leads to the TLB Viewer tool window. This window shows the contents of the Translation Lookaside Buffer (see Sec. A.4.2).

Each entry in the TLB is displayed in one row of the central table, split in two 32-bit words: EntryHi and EntryLo. The right panel shows the breakdown of both words into the composing fields as described in Section A.4.2.

When the execution is paused, the contents of the TLB can be modified through the central table. This feature can be useful to debug Paging Algorithms and TLB Exception Handlers.

---

[2]The Bus Inspector tries to decompile each memory word, without knowing if it is code or data, it is up to the programmer to decide if the decompiled assembly is meaningful or garbage.

[3]All global variables are present in the symbol table, thus not only variables of some `struct` type are listed, but all variables, including base type ones, that aren't local to some function.

Appendice B

JaeOS Specifications

B.1 Phase 1 - Level 2: The Queues Manager

Level 2 of JaeOS instantiates the key operating system concept that active entities at one layer are just data structures at lower layers. In this case, the active entities at a higher level are processes (i.e. programs in execution) and the data structure(s) that represent them at this level are process control blocks (ProcBlk's).

The queue manager will implement four ProcBlk related functions:

- The allocation and deallocation of ProcBlk's.
- The maintenance of queues of ProcBlk's.
- The maintenance of trees of ProcBlk's.
- The maintenance of a single sorted list of active semaphore descriptors each of which supports a queue of ProcBlk's.

```
struct semd_t;
/* process control block type */
struct pcb_t {
    struct pcb_t *p_parent; /* pointer to parent */
    struct semd_t *p_cursem; /* pointer to the semd_t on
                                which process blocked */
    state_t p_s; /* processor state */
    struct clist p_list; /* process list */
    struct clist p_children; /* children list entry point*/
    struct clist p_siblings; /* children list: links to the
                                siblings */
};
```

B.1.1 The Allocation and Deallocation of ProcBlk's

One may assume that JaeOS supports no more that MAXPROC concurrent processes; where MAXPROC should be set to 20 (in the file CONST.H).

Thus this level needs a pool of MAXPROC ProcBlk's to allocate from and deallocate to. Assuming that there is a set of MAXPROC ProcBlk's, the free or unused ones can be kept on a list pointed to by the variable pcbFree (of type `struct clist`).

To support the allocation and deallocation of ProcBlk's there should be the following three externally visible functions:

- ProcBlk's which are no longer in use can be returned to the pcbFree list by using the method:
  `void freePcb(struct pcb_t *p)`
  insert the element pointed to by p onto the pcbFree list.

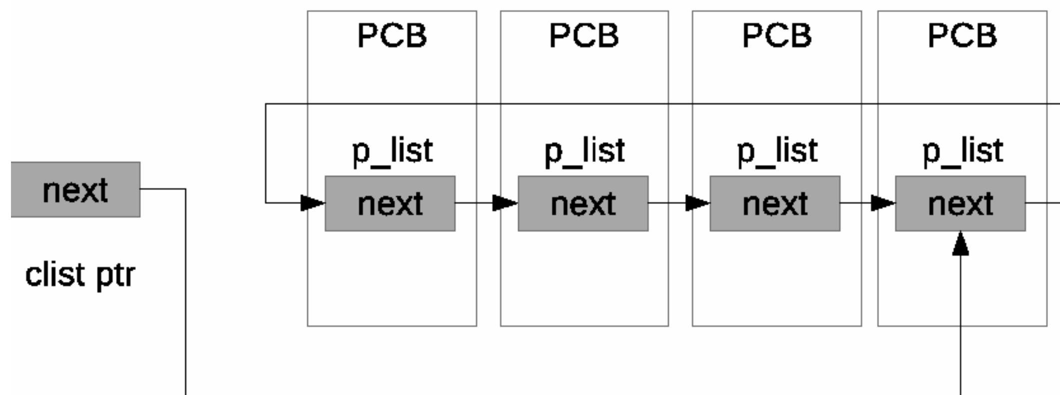
- ProcBlk's should be allocated by using:
  `struct pcb_t *allocPcb()`
  Return NULL if the pcbFree list is empty. Otherwise, remove an element from the pcbFree list, provide initial values for ALL of the ProcBlk's fields (i.e. fill the entire structure by NULL/zero bytes) and then return a pointer to the removed element. ProcBlk's get reused, so it is important that no previous value persist in a ProcBlk when it gets reallocated. There is still the question of how one acquires storage for MAXPROC ProcBlk's and gets these MAXPROC ProcBlk's initially onto the pcbFree list. Unfortunately, there is no malloc() feature to acquire dynamic (i.e. non-automatic) storage that will persist for the lifetime of the OS and not just the lifetime of the function they are declared in. Instead, the storage for the MAXPROC ProcBlk's will be allocated as static storage. A static array of MAXPROC ProcBlk's will be declared in initPcbs(). Furthermore, this method will insert each of the MAXPROC ProcBlk's onto the pcbFree list.

- To initialize the pcbFree List:
  `void initPcbs(void)`
  Initialize the pcbFree list to contain all the elements of the static array of MAXPROC ProcBlk's. This method will be called only once during data structure initialization.

B.1.2 Process Queue Maintenance

The methods below do not manipulate a particular queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the queue upon which the indicated operation is to be performed.

A new process queue is a Linux kernel list (implemented through the macros and inline functions of `listc.h`) thus it can be created using either the macro `LIST_HEAD_INIT` or



the inline function INIT_LIST_HEAD. It is possible to test if a process queue is empty using the list_empty inline function provided in listx.h.

To support process queues there should be the following externally visible functions:

- `void insertProcQ(struct clist *q, struct pcb_t *p)`
  Insert the ProcBlk pointed to by p into the process queue whose list-tail pointer is q.

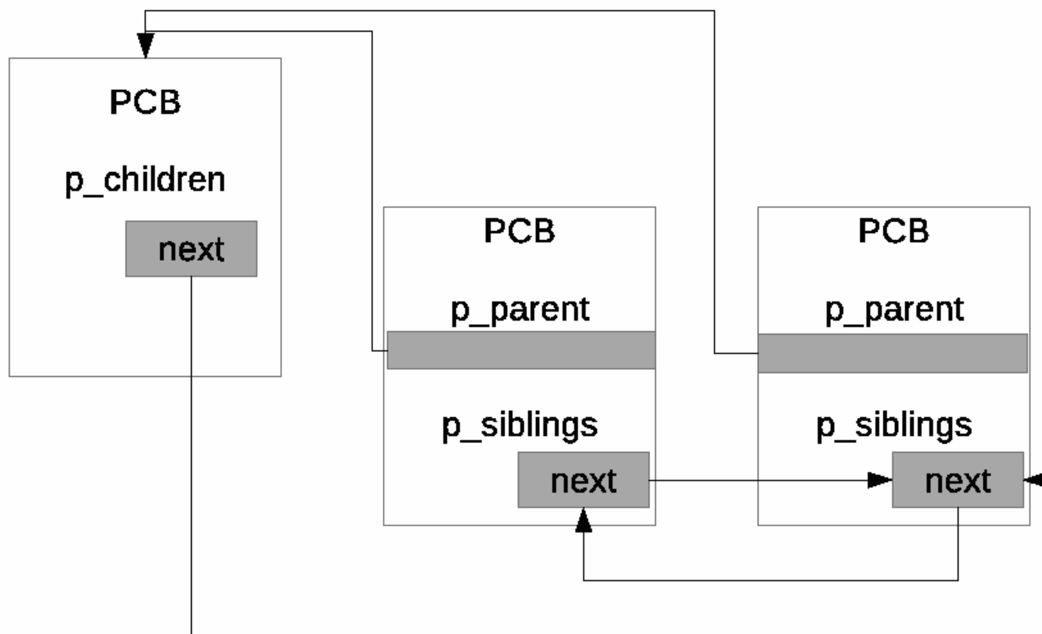
- `struct pcb_t *removeProcQ(struct clist *q)`
  Remove the first (i.e. head) element from the process queue whose list tail pointer is q. Return NULL if the process queue was initially empty; otherwise return the pointer to the removed element.

- `struct pcb_t *outProcQ(struct clist *q, struct pcb_t *p)`
  Remove the ProcBlk pointed to by p from the process queue whose list-tail pointer is q. If the desired entry is not in the indicated queue (an error condition), return NULL; otherwise, return p. Note that p can point to any element of the process queue.

- `struct pcb_t *headProcQ(struct clist *q)`
  Return a pointer to the first ProcBlk from the process queue whose list-tail pointer is q. Do not remove this ProcBlk from the process queue. Return NULL if the process queue is empty.

B.1.3 Process Tree Maintenance

In addition to possibly participating in a process queue, ProcBlk's are also organized into trees of ProcBlk's, called process trees. The p_childred, p_siblings and p_parent



fields of the `struct pcb_t` are used for this purpose. `p_childred` is the entry point of the list of all the children of the process, linked together using the `p_siblings` field. Each child process has a pointer to its parent ProcBlk (`p_parent`). To support process trees there should be the following externally visible functions:

- `int emptyChild(struct pcb_t *p)`
  Return TRUE if the ProcBlk pointed to by p has no children. Return FALSE otherwise.

- `void insertChild(struct pcb_t *parent, struct pcb_t *p)`
  Make the ProcBlk pointed to by p a child of the ProcBlk pointed to by parent.

- `struct pcb_t *removeChild(struct pcb_t *p)`
  Make the first child of the ProcBlk pointed to by p no longer a child of p. Return NULL if initially there were no children of p. Otherwise, return pointer to this removed first child ProcBlk.

- `struct pcb_t *outChild(struct pcb_t *p)`
  Make the ProcBlk pointed to by p no longer the child of its parent. If the ProcBlk pointed to by p has no parent, return NULL; otherwise, return p. Note that the element pointed to by p need not be the first child of its parent.

B.1.4 The Active Semaphore List

A semaphore is an important operating system concept which is needed for Phase 2/Level 3. While understanding semaphores is not needed for this level, this level nevertheless implements an important data structure/abstraction which supports JaeOS's implementation of semaphores. For the purposes of this level it is sufficient to think of a semaphore as an integer. Associated with this integer is its address (semaphores, like all integers, have a physical address) and a process queue. A semaphore is active if there is at least one ProcBlk on the process queue associated it. The following implementation is suggested: maintain a sorted list of semaphore descriptors. This list represents the Active Semaphore List (ASL). Its entry point is the variable named `aslh` (of type `struct clist`). Keep the ASL sorted in ascending order using the address of the semaphore (`s_semAdd` field) as the sort key.

Maintain a second list of semaphore descriptors, the semdFree list, to hold the unused semaphore descriptors. The semaphore descriptors themselves should be declared, like the ProcBlk's, as a static array of size MAXPROC of type `{struct semd_t}`.

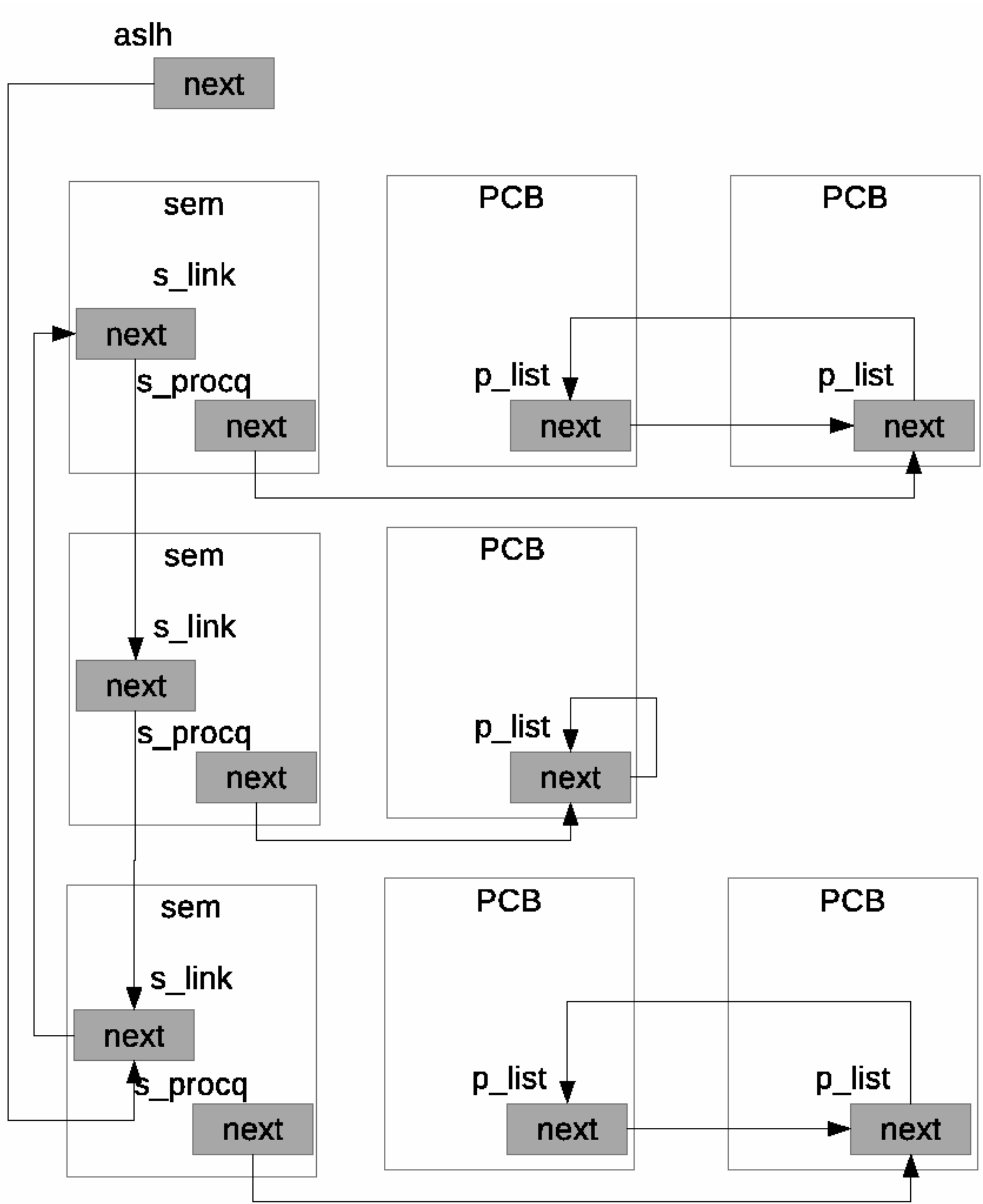
```
struct semd_t {
    int *s_semAdd; /* pointer to the semaphore */
    struct clist s_link; /* ASL linked list */
    struct clist s_procq; /* blocked process queue */
};
```

To support the ASL there should be the following externally visible functions:

- `int insertBlocked(int *semAdd, struct pcb_t *p)`
  Insert the ProcBlk pointed to by p at the tail of the process queue associated with the semaphore whose physical address is semAdd and set the semaphore address of p to semAdd. If the semaphore is currently not active (i.e. there is no descriptor for it in the ASL), allocate a new descriptor from the semdFree list, insert it in the ASL (at the appropriate position), initialize all of the fields (i.e. set `s_semAdd` to `semAdd`, and `s_procq`), and proceed as above. If a new semaphore descriptor needs to be allocated and the semdFree list is empty, return TRUE. In all other cases return FALSE.

- `struct pcb_t *removeBlocked(int *semAdd)`
  Search the ASL for a descriptor of this semaphore. If none is found, return NULL; otherwise, remove the first (i.e. head) ProcBlk from the process queue of the found semaphore descriptor and return a pointer to it. If the process queue for this semaphore becomes empty remove the semaphore descriptor from the ASL and return it to the semdFree list.

- `struct pcb_t *outBlocked(struct pcb_t *p)`
  Remove the ProcBlk pointed to by p from the process queue associated with p's


aslh
next
sem
s_link
next
s_procq
next
PCB
p_list
next
PCB
p_list
next
sem
s_link
next
s_procq
next
PCB
p_list
next
sem
s_link
next
s_procq
next
PCB
p_list
next
PCB
p_list
next

semaphore on the ASL. If ProcBlk pointed to by p does not appear in the process queue associated with p's semaphore, which is an error condition, return NULL; otherwise, return p.

- `struct pcb_t *headBlocked(int *semAdd)`
  Return a pointer to the ProcBlk that is at the head of the process queue associated with the semaphore semAdd. Return NULL if semAdd is not found on the ASL or if the process queue associated with semAdd is empty.

- `void initASL(void)`
  Initialize the semdFree list to contain all the elements of the array
  `static struct semd_t semdTable[MAXPROC]`
  This method will be only called once during data structure in initialization.

B.1.5 Nuts and Bolts

There is no one right way to implement the functionality of this level. One approach is to create two modules: one for the ASL and one for ProcBlk initialization/allocation/-deallocation, process queue maintenance, and process tree maintenance.

The second module, PCB.C, in addition to the public and HIDDEN/private (static) helper functions, will also contain the declaration for the private global variable that points to the head of the pcbFree list, e.g.
`HIDDEN struct clist pcbFree;`

The ASL module, ASL.C, in addition to the public and HIDDEN/private helper functions, will also contain the declarations for aslh and semdFree, e.g.
`HIDDEN struct clist aslh, semdFree;`

Since the ASL module will make calls to the process queue module to manipulate the process queue associated with each active semaphore, this module should `#include "pcb.h"`. This will insure that the ASL can only use the externally visible functions from PCB.C. for maintaining its process queues. Furthermore, the declaration for `struct pcb_t` would then be placed in `types.h`. This is because many other modules will need to access this definition. The declaration for `struct semd_t` is placed in ASL.C because no other module will ever need to access this definition; hence it is local to the module. As with any non-trivial system, you are strongly encouraged to use the make program to maintain your code.

B.1.6 Testing

There will be a provided test file, P1TEST.C that will exercise your code. The test program reports on its progress by writing messages to TERMINAL0. These messages are also added to one of two memory buffers; errbuf for error messages and okbuf for all

other messages. At the conclusion of the test program, either successful or unsuccessful, arm will display a final message. The final message will either be SYSTEM HALTED for successful termination, or KERNEL PANIC for unsuccessful termination.

B.2 Phase 2 - Level 3: The Nucleus

Level 3 (the nucleus) of JaeOS builds on previous levels in two key ways:

- Building on the exception handling facility of Level 1 (the ROM-Excpt handler), provide the exception handlers that the ROM-Excpt handler "passes" exception handling "up" to. There will be one exception handler for each type of exception: Program Traps (PgmTrap), SYSCALL/ Breakpoint (SYS/Bp), TLB Management (TLB), and Interrupts (Ints).
- Using the data structures from Level 2 (Phase 1), and the facility to handle both SYS/Bp exceptions and Interrupts - timer interrupts in particular - provide a process scheduler.

The purpose of the nucleus is to provide an environment in which asynchronous sequential processes (i.e. heavyweight threads) exist, each making forward progress as they take turns sharing the processor. Furthermore, the nucleus provides these processes with exception handling routines, low-level synchronization primitives, and a facility for "passing up" the handling of PgmTrap, TLB exceptions and certain SYS/Bp exceptions to the next level; the VM-I/O support level Level (Phase 3). Since virtual memory is not supported by JaeOS until Level 4, all addresses at this level are assumed to be physical addresses. Nevertheless, the nucleus needs to preserve the state of each process. e.g. If a process is executing with virtual memory on (`CP15_Control[0]=1`) when it is either interrupted or executes a `SYSCALL`, then `CP15_Control` should still be set to 1 when it continues its execution.

B.2.1 The Scheduler

Your nucleus should guarantee finite progress; consequently, every ready process will eventually have an opportunity to execute. Processes in JaeOS belong to four different priority levels: `PRIO_LOW`, `PRIO_NORM`, `PRIO_HIGH` and `PRIO_IDLE`. The latter is reserved for a special Idle Process that "twiddles its thumbs" by endlessly putting the processor in a Wait State, waiting for an interrupt to occur. μARM supports a `WAIT` ROM service expressly for this purpose.

The scheduler uses a 5 millisecond time slice selecting the running process among the highest priority ready processes. The scheduler also needs to perform some simple deadlock detection and if deadlock is detected perform some appropriate action; e.g. invoke the `PANIC` ROM service/instruction. We define the following

- Ready Queues: Four queues of ProcBlk's (`struct list_head`) representing processes that are ready and waiting for a turn at execution (one queue per priority level).
- Current Process: A pointer to a ProcBlk that represents the current executing process.
- Process Count: The count of the number of processes in the system.
- Soft-block Count: The number of processes in the system currently blocked and waiting for an interrupt; either an I/O to complete, or a timer to "expire."

The scheduler should behave in the following manner if the three main Ready Queues are empty (i.e. the Idle Process has to be executed):

1. If the Process Count is zero invoke the `HALT` ROM service/instruction.
2. Deadlock for JaeOS is defined as when the Process Count > 0 and the Soft-block Count is zero. Take an appropriate deadlock detected action. (e.g. Invoke the `PANIC` ROM service/instruction.)
3. Continue execution loading the Idle Process.

B.2.2 Nucleus Initialization

Every program needs an entry point (i.e. `main()`), even JaeOS. The entry point for JaeOS performs the nucleus initialization, which includes:

1. Populate the four New Areas in the ROM Reserved Frame. For each New processor state:
    - Set the `PC` to the address of your nucleus function that is to handle exceptions of that type.
    - Set the `SP` to RAMTOP. Each exception handler will use the last frame of RAM for its stack.
    - Set the `CPSR` register to mask all interrupts and be in kernel-mode (System mode).
2. Initialize the Level 2 (Phase 1 - see Chapter 2) data structures:

```
initPcbs()
initASL()
```

3. Initialize all nucleus maintained variables: Process Count, Soft-block Count, Ready Queues, and Current Process.
4. Initialize all nucleus maintained semaphores. In addition to the above nucleus variables, there is one semaphore variable for each external (sub)device in μARM, plus a semaphore to represent a pseudo-clock timer. Since terminal devices are actually two independent sub-devices (see Section 5.7-pops), the nucleus maintains two semaphores for each terminal device. All of these semaphores need to be initialized to zero.
5. Instantiate a single process and place its ProcBlk in the `PRIO_NORM` Ready Queue. A process is instantiated by allocating a ProcBlk (i.e. `allocPcb()`), and initializing the processor state that is part of the ProcBlk. In particular this process needs to have interrupts enabled, virtual memory off, the processor Local Timer enabled, kernel-mode on, `SP` set to RAMTOP-FRAMESIZE (i.e. use the penultimate RAM frame for its stack), and its `PC` set to the address of `test`. Test is a supplied function/process that will help you debug your nucleus. One can assign a variable (i.e. the `PC`) the address of a function by using

   ```
   . . . = (memaddr)test
   ```

   where `memaddr`, in `const.h` has been aliased to `unsigned int`. Remember to declare the test function as "external" in your program by including the line:

   ```
   extern void test();
   ```

6. Set-up the Idle Process and add it to its proper Ready Queue.
7. Call the scheduler.

Once main() calls the scheduler its task is completed since control will never return to main(). At this point the only mechanism for re-entering the nucleus is through an exception; which includes device interrupts. As long as there are processes to run, the processor is executing instructions on their behalf and only temporarily enters the nucleus long enough to handle the device interrupts and exceptions when they occur. At boot/reset time the nucleus is loaded into RAM beginning with the second frame of RAM; 0x0000.8000. The first frame of RAM is the ROM Reserved Frame. Furthermore, the processor will be in kernel-mode with virtual memory disabled and all interrupts masked. The `PC` is assigned 0x0000.8000 and the `SP`, which was initially set to RAMTOP at boot-time, will now be some value less than RAMTOP due to the activation record for `main()` that now sits on the stack.

B.2.3 SYS/Bp Exception Handling

A SYSCALL or Breakpoint exception occurs when a SYSCALL or BREAK assembler instruction is executed. Assuming that the SYS/Bp New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a SYSCALL or Breakpoint exception is raised, execution continues with the nucleus's SYS/Bp exception handler. A SYSCALL exception is distinguished from a Breakpoint exception by the value of the SWI instruction which raised the exception. SYSCALL exceptions are recognized via an exception code of Sys (8) while Breakpoint exceptions are recognized via an exception code of Bp (9). By convention the executing process places appropriate values in user registers a1-a4 immediately prior to executing a SYSCALL or BREAK instruction. The nucleus will then perform some service on behalf of the process executing the SYSCALL or BREAK instruction depending on the value found in a1. In particular, if the process making a SYSCALL request was in kernel-mode and a1 contained a value in the range [1..10] then the nucleus should perform one of the services described below.

Create Process (SYS1)

When requested, this service causes a new process, said to be a progeny of the caller, to be created. a2 should contain the physical address of a processor state area at the time this instruction is executed and a3 should contain the priority level (PRIO_LOW, PRIO_NORM, PRIO_HIGH). This processor state should be used as the initial state for the newly created process. The process requesting the SYS1 service continues to exist and to execute. If the new process cannot be created due to lack of resources (for example no more free ProcBlk's) or the priority level is not an acceptable value (including PRIO_IDLE), an error code of -1 is placed/returned in the caller's a1, otherwise, return the PID of the newly created process in a1. The SYS1 service is requested by the calling process by placing the value 1 in a1, the physical address of a processor state in a2, the priority level in a3, and then executing a SYSCALL instruction. The following C code can be used to request a SYS1:

```
int SYSCALL (CREATEPROCESS, state t *statep, priority_enum prio)
```

Where the mnemonic constant CREATEPROCESS has the value of 1.

Terminate Process (SYS2)

This services causes the executing process or one process in its progeny to cease to exist. In addition, recursively, all progeny of the designated process are terminated as well. Execution of this instruction does not complete until all progeny are terminated. The SYS2 service is requested by the calling process by placing the value 2 in a1, and the value of the designated process' PID in a2 and then executing a SYSCALL instruction.

When a2 is set to zero, SYS2 terminates the calling process. The following C code can be used to request a SYS2:

```
void SYSCALL (TERMINATEPROCESS, pid_t pid)
```

Where the mnemonic constant TERMINATEPROCESS has the value of 2.

Verhogen (V) (SYS3)

When this service is requested, it is interpreted by the nucleus as a request to perform a weighted V operation on a semaphore. The V or SYS3 service is requested by the calling process by placing the value 3 in a1, the physical address of the semaphore to be V'ed in a2, the weight value (number of resources to be freed) in a3, and then executing a SYSCALL instruction. The following C code can be used to request a SYS3:

```
void SYSCALL (VERHOGEN, int *semaddr, int weight)
```

Where the mnemonic constant VERHOGEN has the value of 3.

Passeren (P) (SYS4)

When this services is requested, it is interpreted by the nucleus as a request to perform a weighted P operation on a semaphore. The P or SYS4 service is requested by the calling process by placing the value 4 in a1, the physical address of the semaphore to be P'ed in a2, the weight value (number of resources to be allocated) in a3, and then executing a SYSCALL instruction. The following C code can be used to request a SYS4:

```
void SYSCALL (PASSEREN, int *semaddr, int weight)
```

Where the mnemonic constant PASSEREN has the value of 4.

Specify Exception State Vector (SYS5)

When this service is requested, the kernel stores the pointers to six processor state areas for exception pass-up. The SYS5 service is requested by the calling process by placing the value 5 in a1, and an array of six processor state pointers in a2. These pointers are the addresses of the OLD area for TLB exceptions, the NEW area for TLB exceptions, the OLD and NEW areas for SYSCALL and the OLD and NEW area for Program Trap, respectively. When an exception occurs for which an Exception State Vector has been specified for, the nucleus stores the processor state at the time of the exception in the area pointed to by the address in the specific OLD area, and loads the new processor state from the area pointed to by the address given in the corresponding NEW area. A process may request SYS5 service at most once. An attempt to request a SYS5 service more than once by any process should be construed as an error and treated as a SYS2 of

the process itself. If an exception occurs while running a process which has not specified an Exception State Vector for that exception type, then the nucleus should treat the exception as a SYS2 as well. The following C code can be used to request a SYS5:

```
void SYSCALL (SPECTRAPVEC, state t **state_vector)
```

Where the mnemonic constant SPECTRAPVEC has the value of 5.

Get CPU Time (SYS6)

When this service is requested, it causes the processor time (in microseconds) used by the requesting process, both as global time and user time, to be returned to the calling process (global time is the time spent by the processor for the calling process, including kernel time and user time). This means that the nucleus must record (in the ProcBlk) the amount of processor time used by each process. The SYS6 service is requested by the calling process by placing the value 6 in `a1`, the address of a `cputime_t` variable in `a2`, the address of a second `cputime_t` variable in `a3` and then executing a `SYSCALL` instruction. At SYS6 completion, the global processor time should be stored at the address specified as `a2` while the processor time in user time should be stored at the address specified as `a3`. The following C code can be used to request a SYS6:

```
void SYSCALL (GETCPUTIME, cputime_t *global, cputime_t *user)
```

Where the mnemonic constant GETCPUTIME has the value of 6.

Wait For Clock (SYS7)

This instruction performs a P operation on the nucleus maintained pseudo-clock timer semaphore. This semaphore is V'ed every 100 milliseconds automatically by the nucleus. The SYS7 service is requested by the calling process by placing the value 7 in `a1` and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS7:

```
void SYSCALL (WAITCLOCK)
```

Where the mnemonic constant WAITCLOCK has the value of 7.

Wait for IO Device (SYS8)

This service performs a P operation on the semaphore that the nucleus maintains for the I/O device indicated by the values in `a2`, `a3`, and optionally `a4`. (P and V operation on I/O semaphores have always weight equal to 1). Note that terminal devices are two independent sub-devices, and are handled by the SYS8 service as two independent devices. Hence each terminal device has two nucleus maintained semaphores for it; one

for character receipt and one for character transmission. As discussed in Section 3.6 the nucleus will perform a V operation on the nucleus maintained semaphore whenever that (sub)device generates an interrupt. Once the process resumes after the occurrence of the anticipated interrupt, the (sub)device's status word is returned in `a1`. For character transmission and receipt, the status word, in addition to containing a device completion code, will also contain the character transmitted or received. As described below in Section 3.6 it is possible that the interrupt can occur prior to the request for the SYS8 service. In this case the requesting process will not block as a result of the P operation and the interrupting device's status word, which was stored off, can now be placed in `a1` prior to resuming execution. The SYS8 service is requested by the calling process by placing the value 8 in `a1`, the interrupt line number in `a2`, the device number in `a3` ([0...7]) , TRUE or FALSE in `a4` to indicate if waiting for a terminal read operation, and then executing a `SYSCALL` instruction. The following C code can be used to request a SYS8:

```
unsigned int SYSCALL (WAITIO, int intlNo, int dnum,
int waitForTermRead)
```

Where the mnemonic constant WAITIO has the value of 8.

Get Process ID (SYS9)

This service returns the Process ID of the caller by placing its value in `a1`. The SYS9 service is requested by the calling process by placing the value 9, in `a1`. The following C code can be used to request a SYS9:

```
pid_t SYSCALL(GETPID)
```

Where the mnemonic constant GETPID has the value of 9.

Get Process Parent ID (SYS10)

This service returns the Process ID of the caller's parent by placing its value in `a1`. The SYS10 service is requested by the calling process by placing the value 10, in `a1`. The following C code can be used to request a SYS10:

```
pid_t SYSCALL(GETPPID)
```

Where the mnemonic constant GETPPID has the value of 10.

SYS1-SYS10 in User-Mode

The above ten nucleus services are considered privileged services and are only available to processes executing in kernel-mode. Any attempt to request one of these services while in user-mode should trigger a PgmTrap exception response. In particular JaeOS should simulate a PgmTrap exception when a privileged service is requested in user-mode. This is done by moving the processor state from the SYS/Bp Old Area to the PgmTrap Old Area, setting `CP15_Cause.ExcCode` in the PgmTrap Old Area to `EXC_RESERVEDINSTR` (Reserved Instruction), and calling JaeOS's PgmTrap exception handler.

Breakpoint Exceptions and SYS11 and Above Exceptions

The nucleus will directly handle all SYS1-SYS10 requests. The ROM-Excpt handler will also directly handle some Breakpoint exceptions; those where the requesting process was executing in kernel-mode and `a1` contained the code for either `LDST`, `PANIC`, or `HALT`. For all other `SYSCALL` and Breakpoint exceptions the nucleus's SYS/Bp exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the SYS/Bp exception should be handled like a SYS2: the current process and all its progeny are terminated.
- If the offending process has issued a SYS5, the handling of the SYS/Bp exception is "passed up." The processor state is moved from the SYS/Bp Old Area into the processor state area whose address was recorded in the ProcBlk as the SYS/Bp Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the SYS/Bp New Area Address is made the current processor state.

B.2.4 PgmTrap Exception Handling

A PgmTrap exception occurs when the executing process attempts to perform some illegal or undefined action. This includes all of the program trap types and reserved instructions. Assuming that the PgmTrap New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a PgmTrap exception is raised, execution continues with the nucleus's PgmTrap exception handler. The cause of the PgmTrap exception will be set in `CP15_Cause.ExcCode` in the PgmTrap Old Area. The nucleus's PgmTrap exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the PgmTrap exception should be handled like a SYS2: the current process and all its progeny are terminated.

- If the offending process has issued a SYS5, the handling of the PgmTrap is "passed up." The processor state is moved from the PgmTrap Old Area into the processor state area whose address was recorded in the ProcBlk as the PgmTrap Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the PgmTrap New Area Address is made the current processor state.

B.2.5 TLB Exception Handling

A TLB exception occurs when μARM fails in an attempt to translate a virtual address into its corresponding physical address. Assuming that the TLB New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when a TLB exception is raised, execution continues with the nucleus's TLB exception handler. The cause of the TLB exception will be set in `CP15_Cause.ExcCode` in the TLB Old Area. The nucleus's TLB exception handler will take one of two actions depending on whether the offending (i.e. Current) process has performed a SYS5:

- If the offending process has NOT issued a SYS5, then the TLB exception should be handled like a SYS2: the current process and all its progeny are terminated.
- If the offending process has issued a SYS5, the handling of the PgmTrap is "passed up." The processor state is moved from the TLB Old Area into the processor state area whose address was recorded in the ProcBlk as the TLB Old Area Address. Finally, the processor state whose address was recorded in the ProcBlk as the TLB New Area Address is made the current processor state.

B.2.6 Interrupt Exception Handling

A device interrupt occurs when either a previously initiated I/O request completes or when the Interval Timer makes a `0x0000.0000` $\rightarrow$ `0xFFFF.FFFF` transition. Assuming that the Ints New Area in the ROM Reserved Frame was correctly initialized during nucleus initialization, then after the processor's and ROM-Excpt handler's actions when an Ints exception is raised, execution continues with the nucleus's Ints exception handler. Which interrupt lines have pending interrupts is set in `CP15_Cause.IP`. Furthermore, for interrupt lines 3–7 the Interrupting Devices Bit Map, as defined in the μARM informal specifications document, will indicate which devices on each of these interrupt lines have a pending interrupt. It is important to note that many devices per interrupt line may have an interrupt request pending, and that many interrupt lines may simultaneously be on. Also, since each terminal device is two sub-devices, each terminal device may have two pending interrupts simultaneously as well. You are strongly encouraged to process only one interrupt at a time: the interrupt with the highest priority. The lower the interrupt line and device number, the higher the priority of the interrupt. When

there are multiple interrupts pending, and the Interrupt exception handler only processes the single highest priority pending interrupt, the Interrupt exception handler will be immediately re-entered as soon as interrupts are unmasked again; effectively forming a loop until all the pending interrupts are processed. As described in Section 5.7-pops, terminal devices are actually two sub-devices; a transmitter and a receiver. These two sub-devices operate independently and concurrently. Both sub-devices may have an interrupt pending simultaneously. For purposes of prioritizing pending interrupts, terminal transmission (i.e. writing to the terminal) is of higher priority than terminal receipt (i.e. reading from the terminal). Hence the processor Interval Timer (interrupt line 2) is the highest priority interrupt and reading from terminal 7 (interrupt line 7, device 7; read) is the lowest priority interrupt. The nucleus's Interrupts exception handler will perform a number of tasks:

- Acknowledge the outstanding interrupt. For all devices except the Interval Timer this is accomplished by writing the acknowledge command code in the interrupting device's `COMMAND` device register. Alternatively, writing a new command in the interrupting device's device register will also acknowledge the interrupt. An interrupt for a timer device is acknowledged by loading the timer with a new value.

- Perform a V operation (weight 1) on the nucleus maintained semaphore associated with the interrupting (sub)device if the semaphore has value less than 1. The nucleus maintains two semaphores for each terminal sub-device. For Interval Timer interrupts that represent a pseudo-clock tick (see Section 3.7.1), perform the V operation on the nucleus maintained pseudo-clock timer semaphore. A V operation on the pseudo clock should unlock all the waiting processes.

- If the SYS8 for this interrupt was requested prior to the handling of this interrupt, recognized by the V operation above unblocking a blocked process, store the interrupting (sub)device's status word in the newly unblocked process's `a1`. If the SYS8 for this interrupt has not yet been requested, recognized by the V operation not unblocking any process, store off the interrupting device's status word until the SYS8 is eventually requested.

B.2.7 Nuts and Bolts

Timing Issues

While μARM has two clocks, the TOD clock and Interval Timer, only the Interval Timer can generate interrupts. Hence the Interval Timer must be used simultaneously for two purposes: generating interrupts to signal the end of processes' time slices, and to signal the end of each 100 millisecond period (a pseudo-clock tick); i.e. the time to V the semaphore associated with the pseudo-clock timer. It is insufficient to simply

V the pseudo-clock timer's semaphore after every 20 time slices; processes may block (SYS4, SYS7, or SYS8) or terminate (SYS2) long before the end of their current time slice. A more careful accounting method is called for; one where some (most) of the Interval Timer interrupts represent the conclusion of a time slice while others represent the conclusion of a pseudo-clock tick. Furthermore, when no process requests a SYS7, the pseudo-clock timer semaphore, if left unadjusted, will grow by 1 every 100 milliseconds. This means that if a process, after 500 milliseconds requests a SYS7, and there were no intervening SYS7 requests, it will not block until the next pseudo-clock tick as hoped for, but will immediately resume its execution stream. Therefore at each pseudo-clock tick, if no process was unblocked by the V operation (i.e. the semaphore's value after the increment performed during the V operation was greater than zero), the semaphore's value should be decremented by one (i.e. reset to zero). The opposite is also true; if more than one process requests a SYS7 in between two adjacent pseudo-clock ticks then at the next pseudo-clock tick, all of the waiting processes should be unblocked and not just the process that was waiting the longest. The CPU time used by each process must also be kept track of (i.e. SYS6). This implies an additional field to be added to the ProcBlk structure. While the Interval Timer is useful for generating interrupts, the TOD clock is useful for recording the length of an interval. By storing off the TOD clock's value at both the start and end of an interval, one can compute the duration of that interval. The Interval Timer and TOD clock are mechanisms for implementing JaeOS's policy. Timing policy questions that need to be worked out include:

- While the time spent by the nucleus handling an I/O or Interval Timer interrupt needs to be measured for pseudo-clock tick purposes, which process, if any, should be "charged" with this time. Note: it is possible for an I/O or Interval Timer interrupt to occur even when there is no current process.

- While the time spent by the nucleus handling a `SYSCALL` request needs to be measured for pseudo-clock tick purposes, which process, if any, should be "charged" with this time.

Returning from a SYS/Bp Exception

`SYSCALL`'s that do not result in process termination return control to the requesting process's execution stream. This is done either immediately (e.g. SYS6) or after the process is blocked and eventually unblocked (e.g. SYS8). In any event the `PC` that was saved is, for this kind of exceptions, the address of the address of the `SYSCALL` assembly instruction that caused the exception.

Loading a New Processor State

It is the job of the ROM-Excpt handler to load new processor states; either as part of "passing up" exception handling (the loading of the processor state from the appropriate New Area) or for `LDST` processing. Whenever the ROM-Excpt handler loads a processor state, the previous state of the running process get stored in the corresponding OLD area. Any information concerning the running process (prior to the exception) should be retrieved from that OLD state, e.g. the CPRS register fields are useful to state the running execution mode.

Process Termination

When a process is terminated there is actually a whole (sub)tree of processes that get terminated. There are a number of tasks that must be accomplished:

- The root of the sub-tree of terminated processes must be "orphaned" from its parents; its parent can no longer have this ProcBlk as one of its progeny.
- If the value of a semaphore is negative, it means that some processes (of which the ProcBlks are blocked on that semaphore) have requested a number of resources. Hence if a terminated process is blocked on a semaphore, the value of the semaphore must be adjusted.
- If a terminated process is blocked on a device semaphore, the semaphore should NOT be adjusted. When the interrupt eventually occurs the semaphore will get V'ed by the interrupt handler.
- The process count and soft-blocked variables need to be adjusted accordingly.
- Processes (i.e. ProcBlk's) can't hide. A ProcBlk is either the current process, sitting on a ready queue, blocked on a device semaphore, or blocked on a non-device semaphore.

Module Decomposition

One possible module decomposition is as follows:

1. `initial.c` This module implements main() and exports the nucleus's global variables. (e.g. Process Count, device semaphores, etc.)
2. `interrupts.c` This module implements the device interrupt exception handler. This module will process all the device interrupts, including Interval Timer interrupts, converting device interrupts into V operations on the appropriate semaphores.

3. `exceptions.c` This module implements the TLB, PgmTrap and SYS/Bp exception handlers.

4. `scheduler.c` This module implements JaeOS's process scheduler and deadlock detector.

B.3 Phase 3 - Level 4: The VM-I/O Support Level

Level 4 (the VM-I/O support level) of JaeOS builds on the nucleus level in four key ways to create an environment for the execution of user-processes (U-proc's):

- Support for virtual memory (VM). Each U-proc will run with **CP15.R1**.VMc=1 in its own virtual memory space using a unique ASID.
- Support for accessing the various I/O devices. In particular, U-proc's will be loaded from tape devices and one of the disk devices will be used as the backing store for the VM implementation. U-proc's will have read/write access to the other disk devices, write access to the printers and terminals and optionally read access to the terminals as well.
- To facilitate U-proc cooperation, this level will provide high-level process synchronization primitives; a P and V service that supports virtual addresses.
- Support for a process delay facility. U-proc's, in addition to being able to access the TOD clock, will have the capability to "sleep" for a specified period of time. i.e. A U-proc can request that it be removed from the Ready Queue for the specified period of time.

Another perspective on the VM-I/O support level is that by building on the exception handling facility of the nucleus, this level provides the U-proc-level exception handlers that the nucleus exception handlers "passes" exception handling "up" to, assuming that an appropriate SYS5 was performed. There will be one exception handler for each type of exception:

- Program Trap (PgmTrap) exceptions: This exception handler will terminate the process in a controlled manner.
- SYSCALL/Breakpoint (SYS/Bp) exceptions: This exception handler will implement the new SYS services the VM-I/O support level exports/ implements.
- TLB Management (TLB) exceptions: This exception handler will implement JaeOS's virtual memory support; i.e. the system Pager.

These exception handlers will run in kernel-mode with **CP15.R1**.VMc=1, while the U-proc's will run in user-mode with **CP15.R1**.VMc=1. Hence each U-proc leads a schizophrenic life, mostly executing in user-mode, but sometimes, after the handling of an exception is "passed" back up to it, executing in kernel-mode. While the nucleus exception and interrupt handlers are system-wide resources that all processes share (in serial fashion), the VM-I/O support level exception handlers are more like VM-I/O support level provided libraries that becomes part of each U-proc.

The overall purpose of the VM-I/O support level is to provide a simple time-sharing system which will support up to eight U-proc's. `UPROCMAX` should be defined to the specific degree of multi-programming to be supported/implemented: [1...8]. Associated with each U-proc will be a terminal, a printer, and a tape device which will hold the U-proc's executable image. U-proc's will each execute with **CP15.R1**.VMc=1 and in their own unique virtual address space. U-proc's, which run in user-mode with interrupts enabled, are considered untrustworthy. Nonetheless U-proc's need a way to request system services in a safe way that does not compromise system security.

This suggests implementing new SYS operations. These are outlined in Section B.3.1. However, several problems need to be addressed when providing this support:

- I/O - a process that can initiate I/O operations could specify any memory location for data transfer and can thus overwrite any location it wishes.
  Solution: make sure the page frame containing the device registers is not accessible to U-proc's. Since the device registers reside in Kseg0 and U-proc's run in user-mode with **CP15.R1**.VMc=1, access to the device registers is prevented by the hardware.

- Nucleus implemented SYS's - by specifying random locations as a semaphore, a process could alter any location it liked.
  Solution: run the U-proc's in user mode, since they are then prohibited from issuing nucleus implemented SYS's. This is why the nucleus specification contained this restriction.

- Arbitrary memory accesses using loads, stores, etc.
  Solution: use μARM's virtual memory support to give each U-proc a private address space mapped to Useg2 using a unique ASID. This will give each U-proc an address space disjoint from both the VM-I/O support level's and the other U-proc's address spaces. Actually, JaeOS provides for some pages to be shared by all U-proc's: the first n pages of Useg3. A malicious U-proc could therefore interfere with other U-proc's that were using Useg3 for synchronization purposes. This however is not a security hole, merely a nuisance.

It needs to be stressed that U-proc's must be assumed to be untrustworthy. Thus, it will be necessary for you to construct the VM-I/O support level so that it protects itself

from U-proc's. For one thing, this will make this level easier to debug; U-proc's will be stopped before they (completely) destroy the integrity of the system.

The VM-I/O support level's functions, which are considered to be trustworthy, will run in kernel-mode with interrupts unmasked. The code and data for these functions will reside in the address space of the kernel. Probably the trickiest aspect of the VM-I/O support level is that the functions of this level must be able to access the private data for each U-proc.

As described in Section 3.2 the nucleus, after initialization, starts the system by creating a single process: user-mode off, interrupts enabled, **CP15.R1**.VMc=0, Local Timer enabled, **SP** set to the penultimate RAM frame, and **PC**=`test`. The VM-I/O support level initialization function should therefore be called `test`.

B.3.1 SYS/Bp Exception Handling

As described in Section 3.3, the nucleus directly handles all SYS1-SYS10 SYSCALL exceptions and Breakpoint exceptions [1...4] (**LDST**, **PANIC**, and **HALT**). For all other SYSCALL and Breakpoint exceptions the nucleus either treats the exception as a SYS2 or if the offending process has issued a SYS5 specifying a handler for SYS/Bp exceptions "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's SYS/Bp New Area was correctly initialized, and a SYS5 for SYS/Bp exceptions was performed during U-proc initialization, execution continues with the VM-I/O support level's SYS/Bp exception handler. The processor state at the time of the exception will be in the U-proc's SYS/Bp Old Area.

A SYSCALL exception is distinguished from a Breakpoint exception by the contents of
Cause.ExcCode in the U-proc's SYS/Bp Old Area. SYSCALL exceptions are recognized via an exception code of *Sys* (8) while Breakpoint exceptions are recognized via an exception code of *Bp* (9).

By convention the executing process places appropriate values in user registers **a1-a4** immediately prior to executing a **SYSCALL** or **BREAK** instruction. The VM-I/O support level SYS/Bp exception handler will then perform some service on behalf of the U-proc executing the **SYSCALL** or **BREAK** instruction depending on the value found in **a1**.

In particular, if a U-proc executes a **SYSCALL** instruction and **a1** contained a value in the range [11..20] then the VM-I/O support level should perform one of the services described below.

Read From Terminal (SYS11)

`int SYS9 (READ FROM TERMINAL, char *addr)` When requested, this service causes the requesting U-proc to be suspended until a line of input (string of characters) has

been transmitted from the terminal device associated with the U-proc.

The SYS11 service is requested by the calling U-proc by placing the value 11 in **a1**, the virtual address of a string buffer where the data read should be placed in **a2**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **a1** if the read was successful. If the operation ends with a status other than "Character Received" (5), the negative of the device's status value is returned in **a1**.

Attempting to read from a terminal device to an address in Kseg0 is an error and should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS11:

```
int SYSCALL (READTERMINAL, char *virtAddr)
```

Where the mnemonic constant `READTERMINAL` has the value of 11.

Write To Terminal (SYS12)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the terminal device associated with the U-proc.

The SYS12 service is requested by the calling U-proc by placing the value 12 in **a1**, the virtual address of the first character of the string to be transmitted in **a2**, the length of this string in **a3**, and then executing a **SYSCALL** instruction. Once the process resumes the number of characters actually transmitted is returned in **a1** if the write was successful. If the operation ends with a status other than "Character Transmitted" (5), the negative of the device's status value is returned in **a1**.

It is an error to write to a terminal device from an address in Kseg0, request a SYS12 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS12:

```
int SYSCALL (WRITETERMINAL, char *virtAddr, int len)
```

Where the mnemonic constant `WRITETERMINAL` has the value of 12.

V Virtual Semaphore (SYS13)

When this service is requested, it is interpreted by the nucleus as a request to perform a weighted V operation on a semaphore.

The V or SYS13 service is requested by the calling U-proc by placing the value 13 in **a1**, the *virtual* address of the semaphore to be V'ed in **a2**, the weight value (number of resources to be released) in **a3** and then executing a **SYSCALL** instruction.

Attempting to perform a V operation on an address in Kseg0 or Useg2 or to release a negative amount of resources are errors and should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS13:

```
void SYSCALL (VSEMVIRT, int *semaddr, int weight)
```

Where the mnemonic constant `VSEMVIRT` has the value of 13.

P Virtual Semaphore (SYS14)

When this service is requested, it is interpreted by the nucleus as a request to perform a weighted P operation on a semaphore.

The P or SYS14 service is requested by the calling U-proc by placing the value 14 in **a1**, the *virtual* address of the semaphore to be P'ed in **a2**, the wight value (number of resources to be allocated) in **a3** and then executing a **SYSCALL** instruction.

Attempting to perform a P operation on an address in Kseg0 or Useg2 or to alloc a negative amount of resources are errors and should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS14:

```
void SYSCALL (PSEMVIRT, int *semaddr, int weight)
```

Where the mnemonic constant `PSEMVIRT` has the value of 14.

Delay (SYS15)

This service causes the executing U-proc to be delayed for n seconds. The requesting U-proc is to be delayed at least n seconds and not substantially longer. Since the nucleus controls low-level scheduling decisions, all the VM-I/O support level can ensure is that the requesting U-proc not be "schedulable" until n seconds has elapsed and that it becomes schedulable shortly thereafter.

The Delay or SYS15 service is requested by the calling U-proc by placing the value 15 in **a1**, the number of seconds to be delayed in **a2**, and then executing a **SYSCALL** instruction.

Attempting to request a Delay for less than 0 seconds is an error and should result in the U-proc begin terminated (SYS20).

The following C code can be used to request a SYS15:

```
void SYSCALL (DELAY, int secCnt)
```

Where the mnemonic constant `DELAY` has the value of 15.

Disk Put (SYS16) and Disk Get (SYS17)

These services provide *synchronous* I/O on the μARM disk devices. When requested, this service causes the requesting U-proc to be suspended until the disk write (or read) operation has concluded.

The SYS16 (SYS17) service is requested by the calling U-proc by placing the value 16 (17) in **a1**, the virtual address of the 4KB block to be written to the disk (to contain the data from the disk) in **a2**, the disk number ([1...7]) in **a3**, the disk sector to be written onto (read from) in **a4**, and then executing a **SYSCALL** instruction. Once the process resumes **a1** is to contain the completion status of the disk operation. If the operation ends with a status other than "Device Ready" (1), the negative of the completion status is returned in **a1**.

From one perspective disk devices are three dimensional devices: cylinders (or tracks), surfaces (or heads) and tracks (or sectors). From another perspective they are only one-dimensional: sectors. A disk device with x cylinders, y surfaces, and z tracks can be thought of being a (one dimensional) device with $sectCnt = x * y * z$ sectors numbered $[0...(sectCnt - 1)]$. The SYS16/SYS17 services, since they only take a disk sector parameter (instead of a disk sector, surface#, and track#) assumes this one dimensional perspective for disk devices.

Attempting to write to (read from) a disk device from (into) an address in Kseg0 is an error and should result in the U-proc being terminated (SYS20). Similarly, attempting to perform a disk operation upon DISK0, which is reserved for use by the VM implementation as the backing store device is an error and should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS16:

```
int SYSCALL (DISK_PUT, int *blockAddr, int diskNo, int sectNo)
```

Where the mnemonic constant `DISK_PUT` has the value of 16. A SYS17 is requested by substituting `DISK_PUT` with `DISK_GET`, where the mnemonic constant `DISK_GET` has the value of 17.

Write To Printer (SYS18)

When requested, this service causes the requesting U-proc to be suspended until a line of output (string of characters) has been transmitted to the printer device associated with the U-proc.

Once the process resumes the number of characters actually transmitted is returned in **a1**.

The SYS18 service is requested by the calling U-proc by placing the value 18 in **a1**, the virtual address of the first character of the string to be transmitted in **a2**, the length of this string in **a3**, and then executing a **SYSCALL** instruction. Once the process

resumes the number of characters actually transmitted is returned in **a1** if the write was successful. If the operation ends with a status other than "Device Ready" (1), the negative of the device's status value is returned in **a1**.

It is an error to write to a printer device from an address in Kseg0, request a SYS18 with a length less than 0, or a length greater than 128. Any of these errors should result in the U-proc being terminated (SYS20).

The following C code can be used to request a SYS18:

```
int SYSCALL (WRITEPRINTER, char *virtAddr, int len)
```

Where the mnemonic constant `WRITEPRINTER` has the value of 18.

Get TOD (SYS19)

When this service is requested, it causes the number of microseconds since the system was last booted/reset to be placed/returned in the U-proc's **a1**.

The SYS19 service is requested by the calling U-proc by placing the value 19 in **a1** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS19:

```
unsigned int SYSCALL (GETTOD)
```

Where the mnemonic constant `GETTOD` has the value of 19.

Terminate (SYS20)

This services causes the executing U-proc to cease to exist.

When all U-proc's have terminated, JaeOS should "shut down". Thus, somehow the "system" processes created in the VM-I/O support level (e.g. the delay daemon process – see Section B.3.3) must be terminated after all the U-proc's have terminated. Since there should then be no dispatchable or blocked processes, the nucleus scheduler will invoke the **HALT** ROM service/instruction. (See Section 3.1.)

The SYS20 service is requested by the calling process by placing the value 20 in **a1** and then executing a **SYSCALL** instruction.

The following C code can be used to request a SYS20:

```
void SYSCALL (TERMINATE)
```

Where the mnemonic constant `TERMINATE` has the value of 20.

B.3.2 PgmTrap Exception Handling

As described in Section 3.4 the nucleus either treats a PgmTrap exception as a SYS2 or if the offending process has issued a SYS5 for PgmTrap exceptions "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's PgmTrap New Area was correctly initialized, and a SYS5 for PgmTrap exceptions was performed during U-proc initialization, execution continues with the VM-I/O support level's PgmTrap exception handler. The processor state at the time of the exception will be in the U-proc's PgmTrap Old Area.

The VM-I/O support level's PgmTrap exception handler is to terminate the process in an orderly fashion; perform the same operations as a SYS20 request.

B.3.3 Delay Facility

The SYS15 Delay facility allows a requesting U-proc to be "put to sleep" for a specified number of seconds. A process that is neither the current process nor sitting on the Ready Queue can be considered to be "sleeping". There are two issues that need addressing for the implementation of this facility: where to place the U-proc while it is sleeping, and how to keep track of which U-proc's are sleeping so they can be awoken (i.e. placed on the Ready Queue) at the appropriate time.

Where to Store Sleeping U-proc's

As mentioned above, access to the nucleus is limited solely to requesting SYS1-SYS10 services. Therefore the only way to put a U-proc to sleep (i.e. keep it off of the Ready Queue) is to block the U-proc on a semaphore. The VM-I/O support level should therefore contain an array of size `UPROCMAX` of semaphores; one for each U-proc. These semaphores are defined as the *U-proc private semaphores*. As part of U-proc initialization each U-proc's private semaphore should be initialized to zero so that a P operation on this semaphore will cause the U-proc to block. Hence the VM-I/O support level should interpret a SYS15 as a request to perform a P operation on the U-proc's private semaphore.

Keeping Track of Sleeping U-proc's

The VM-I/O support level needs to maintain a list of sleeping U-proc's. The following implementation is suggested: Maintain a sorted list of delay-node descriptors whose head is pointed to by the list_head pointer `delayd_h`. The list `delayd_h` points to will represent the list of pending "wake up calls;" the *Active Delay List* (ADL). Keep the ADL sorted in ascending order using the `d_wakeTime` field as the sort key.

Maintain a second list of delay-node descriptors, the *delaydFree* list, to hold the unused delay-node descriptors. This list, whose head is pointed to by the variable `delaydFree_h`, is kept, like the pcbFree and the semdFree lists, as a Linux kernel list.

The delay-node descriptors themselves should be declared, like the ProcBlk's and semaphore descriptors, as a static array of size `UPROCMAX` of type `delayd_t`. In addition to the `d_list` list_head and `d_wakeTime` integer fields, a delay-node descriptor should also contain an ASID integer field as well, to denote the sleeping U-proc's identity.

When a U-proc requests some "quiet time", in addition to performing a P operation on the U-proc's private semaphore, a delay-node needs to be allocated from the delaydFree list, populated with appropriate values, and inserted into the ADL.

Periodically, the VM-I/O support level needs to examine the ADL to determine if a U-proc's wake time has passed. To accomplish this the VM-I/O support level will implement a special VM-I/O support level process (i.e. a daemon); the *delay daemon process*. The delay daemon process will repeat forever:

1. Request a Wait For Clock (SYS7) nucleus service.
2. Upon resumption of execution, examine the ADL, removing all delay-nodes whose wake time has passed. For each delay-node whose wake time has passed, perform a V operation on the indicated U-proc's private semaphore and return the delay-node to the delaydFree list.

Therefore, the delay process will wake every 100 milliseconds (i.e. a pseudo-clock tick event), examine the ADL, waking up U-proc's if their delay has expired, and then return to sleep (SYS7). The delay daemon process will run in kernel-mode using a unique ASID with **CP15.R1**.VMc=1 and all interrupts enabled.

B.3.4 Virtual P and V Service

The SYS13/SYS14 P & V facility for virtual addresses allows a requesting U-proc to request a P or V operation on a semaphore with a virtual address. Since U-proc's run in user-mode and are restricted to only using virtual addresses, the nucleus SYS3/SYS4 service will not be of use to U-proc's wishing to coordinate their operation through use of semaphores. As with the Delay Facility, there are two issues that need addressing for the implementation of this facility: where to place a U-proc blocked on a virtual-addressed semaphore, and how to keep track of which U-proc's are blocked on a given semaphore so that they can be awoken (i.e. placed on the Ready Queue) at the appropriate time; when a V operation is requested for the specified semaphore.

Where to Store Blocked U-proc's

When a U-proc performs a P (SYS14) operation on a virtual-addressed semaphore and the value of the semaphore becomes ≤ 0 (i.e. the U-proc is to be blocked), the VM-

I/O support level should interpret this as a request to perform a P operation on the requesting U-proc's private semaphore.

Keeping Track of Blocked U-proc's

The VM-I/O support level needs to maintain a list of U-proc's blocked because of a SYS14 operation. The following implementation is suggested: maintain a list of virtSem-node descriptors whose head is pointed to by the list_head pointer `virtSemd_h`. The list `virtSemd_h` points to will represent the list of U-proc's blocked because of a SYS14 operation; the *Active Virtual Semaphore List* (AVSL).

Maintain a second list of virtSem-node descriptors, the *virtSemdFree* list, to hold the unused virtSem-node descriptors. This list, whose head is pointed to by the pointer `virtSemdFree_h`, is kept, like the delaydFree, pcbFree, and semdFree lists, as a Linux kernel list.

The virtSem-node descriptors themselves should be declared, like the delay-node, ProcBlk, and semaphore descriptors, as a static array of size `MAXPROC` of type `virtSemd_t`. In addition to the `vs_list` list_head field, a virtSemd-node descriptor should also contain a `vs_semaphore` integer field, and an ASID integer field, to denote the blocked U-proc's identity.

When a U-proc is to be blocked as a result of a SYS14 request, in addition to performing a P operation on the U-proc's private semaphore, a virtSem-node needs to be allocated from the virtSemdFree list, populated with appropriate values, and inserted into the AVSL.

When a U-proc is to be unblocked as a result of a SYS13 request, a search is made of the AVSL for a virtSem-node with a matching `vs_semaphore` field. This node is removed from the AVSL and returned to the virtSemdFree list. Finally a V operation is performed on the unblocked U-proc's private semaphore.

Remember that insertion into and the search for removal from the AVSL should be performed so that when there is more than one U-proc blocked because of a SYS14 on the same semaphore the order of unblocking/removal is first-in first-out.

B.3.5 Implementing Virtual Memory

As described in Section 3.5 the nucleus either treats a TLB exception as a SYS2 or, if the offending process has issued a SYS5 with valid TLB exception handler, "passes up" the handling of the exception. Assuming that the handling of the exception is to be passed up, that the U-proc's TLB New Area was correctly initialized, and a SYS5 was performed during U-proc initialization, execution continues with the VM-I/O support levels TLB exception handler; the *Pager*. The processor state at the time of the exception will be in the U-proc's TLB Old Area.

There are a number of situations that trigger a TLB exception; see Chapter 4-*pops*/Chapter 3-*uarmdoc*. Which of these exception types are to be handled by the Pager and which are to trigger a SYS20 depend on the sophistication of the Pager to be implemented. This section describes a very basic Pager. Throughout the following sub-Sections are suggestions for improving/expanding the Pager.

System Segment Tables

As described in Section 3.2-*uarmdoc*, each ASID's segment table is located in the ROM Reserved Frame. For a given ASID the segment table holds three entries; the addresses of the Kseg0, Useg2, and Useg3 PTE's. These must be initialized during U-proc initialization.

All U-proc ASID's will share the same Kseg0 PTE; there is one Kseg0 PTE and all ASID Kseg0 PTE segment table pointers will point to it. The single Kseg0 PTE is a VM-I/O support level data structure. This segment table entry is only used when the U-proc generates a Kseg0 reference when in kernel-mode, since any reference to Kseg0 while in user-mode generates an Address Error exception.

All U-proc ASID's will share the same Useg3 PTE; there is one Useg3 PTE and all ASID Useg3 segment table pointers will point to it. The single Useg3 PTE is a VM-I/O support level data structure.

Each U-proc ASID will have its own Useg2 PTE. The array of Useg2 PTE's is also a VM-I/O support level data structure.

Finally, the segment table entries for the delay daemon process will have the same Kseg0 entry above and NULL for both the Useg2 and Useg3 entries.

U-proc Page Tables

As described above the VM-I/O support level needs to implement `UPROCMAX`+2 PTE's; one for each U-proc and one each for the Kseg0 and Useg3 segments. Exclusive of the Kseg0 PTE, each PTE should contain `MAXPAGES` entries (where `MAXPAGES` = 32).

For the Useg3 PTE, the `MAXPAGES` entries should describe the first `MAXPAGES` pages of the segment (`0xC000.0000` – `0xC001.F000`). Each page's PTE entry in **CP15.PTE_Hi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **CP15.PTE_Low** should indicate that the entry is global, but invalid (i.e. not present).

For the individual Useg2 PTE's, the `MAXPAGES` entries should describe the first `MAXPAGES`-1 pages of the segment (`0x8000.0000` – `0x8001.E000`) and the last page of the segment (`0xBFFF.F000`). Each page's PTE entry in **CP15.PTE_Hi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to the ASID of the U-proc. Each page's PTE entry in **CP15.PTE_Low** should indicate that the entry is both not global and invalid.

As the above definitions illustrate, JaeOS is designed to only support U-proc's whose combined `.text`, `.data`, and `.bss` areas never grow beyond `MAXPAGES`-1 pages, whose stack needs never exceed one page, and whose utilization of Useg3 is always within the segments first `MAXPAGES` pages.

The Layout of Kseg0 and its PTE

The installed physical RAM begins at `0x0000.7000` and goes up to RAMTOP. The default bootstrap loaders load the OS beginning at `0x0000.8000`; the first frame of RAM is reserved for the ROM Reserved Frame. The OS **.text** and **.data** areas are loaded adjacent to each other starting at `0x0000.8000`. The stack frame for the nucleus is the last frame of RAM, while the stack frame for `test` is the penultimate frame of RAM. Additionally:

- Each U-proc needs two RAM frames, (`UPROCMAX` $*$ 2), for stack space; one each for its VM-I/O support level SYS/Bp and TLB exception handlers. The stack page for when a U-proc is running in user-mode is the last page of Useg2, a virtual page which will get placed somewhere in the page pool.
- The delay daemon process needs one RAM frame for its stack space.
- Each DMA-supporting I/O device needs a RAM frame for its I/O buffer. The reason for this is covered in Section B.3.8.
- (`UPROCMAX` $*$ 2) frames need to be reserved for VM paging. In spite of the additional RAM, the page pool is limited to (`UPROCMAX` $*$ 2) frames to force paging events.

All of the above frames, except the (`UPROCMAX` $*$ 2) frames comprising the page pool and the two stack frames located below RAMTOP need to be located below `0x8000.0000`. The scheme below illustrates one suggested organization.

Memory region	Address
Nucleus Stack	RAMTOP
Test Stack	
Page Pool	
Unused RAM	
Delay Stack	
U-Proc Exception Handler Stacks	
Disk DMA Buffers	
Tape DMA Buffers	
OS Code/Data	0x0000.8000
ROM Reserved Frame	0x0000.7000

The single Kseg0 PTE needs to be set up so that the VM-I/O support level exception handlers can run with **CP15.R1**.VMc=1, but that no page fault ever occurs for an address in Kseg0, and all addresses generated in Kseg0 get translated into the same physical address that would be generated if **CP15.R1**.VMc=0. The VM-I/O support level exception handlers need to run with **CP15.R1**.VMc=1 so that the U-proc's Useg2 (and Useg3) address space(es) are addressable. Furthermore, the VM-I/O support level exception handlers need to run as if **CP15.R1**.VMc=0 since these handlers interact with the nucleus and device registers which only understand physical addresses.

To accomplish this one should allocate a Kseg0 PTE of approximately 50 entries. These entries will describe the 50 physical frames beginning at `0x0000.7000`. Each page's PTE entry in **CP15.PTE_Hi** should indicate the **VPN** and **SEGNO** of the page, and the ASID should be set to zero. Each page's PTE entry in **CP15.PTE_Low** should indicate that the entry is global, writable (i.e. dirty), valid, and located at its

corresponding **PFN**. (e.g. The page at **SEGNO**=0, **VPN**=0x00007 would have a **PFN**=0x00007.)

The first n entries describe the **.text** and **.data** areas of the OS in addition to the ROM Reserved Frame. The next 16 entries describe the 8 DMA disk buffers and the 8 DMA tape buffers. The final (`UPROCMAX` $*$ 2) + 1 entries describe the stack frames for the VM-I/O support level exception handlers (2 per U-proc) and one for the delay daemon process.

Interestingly the Kseg0 PTE entries describing frames that contain nucleus **.text** and **.data** areas can be either omitted from the PTE or at least marked as invalid without affecting the correct performance of the OS. Since the nucleus runs with **CP15.R1**.VMc=0 it makes no use of the Kseg0 PTE. The VM-I/O support level exception handlers do not directly call nucleus functions or access nucleus variables/data structures; all interaction is via the SYS1-SYS8 nucleus services. Individuals wishing to implement a more sophisticated Pager should attempt this experiment.

The Swap Pool

(`UPROCMAX` $*$ 2) physical frames are reserved for paging purposes. While there is probably sufficient available RAM to use more than (`UPROCMAX` $*$ 2) frames for paging, limiting the page pool to (`UPROCMAX` $*$ 2) frames will insure that page faults will occur. The (`UPROCMAX` $*$ 2) frames that are to be used for the page pool can be located anywhere in RAM, excepting of course the first 50 initial RAM frames or the final two RAM frames. It is recommended that the (`UPROCMAX` $*$ 2) contiguous frames preceding `test`'s stack frame be used for the page pool.

The VM-I/O support level needs to implement a data structure describing the page pool. For each frame in the pool, one needs to record whether the frame is in use or not, and if so, by which U-proc (i.e. ASID) and which virtual page is occupying the frame (**SEGNO**, and **VPN**).

Handling a TLB Invalid Exception

As described in Section 4.3.4-*pops*/Section 3.2-*uarmdoc* all TLB-Refill event's are handled by the ROM-TLB-Refill handler. As for TLB exception types, there are four to consider:

- TLB-Modification(*Mod*): This exception should never occur under normal processing since all valid PTE entries should always be marked as dirty.
- Bad-PgTbl(*BdPT*): This exception should never occur under normal processing since it is hoped that all PTE's will be correctly constructed.
- PTE-MISS(*PTMs*): This exception should never occur under normal processing since if correctly constructed, each PTE contains all the necessary entries. Some of

these entries will be for entries that are invalid, but the entry should nonetheless be present in the PTE.

- TLB-Invalid(*TLBL* & *TLBS*): This is the only exception that the Pager should be designed to handle. This exception indicates a "page missing" page fault. All other TLB exception types should trigger a SYS20 response.

Handling a **TLB-Invalid** exception involves:

1. Determining the **SEGNO** and **VPN** of the offending address. This will be found in the U-proc's TLB Old Area.

2. Gain mutual exclusive access to the paging data structures. More on this can be found in Section B.3.8.

3. Determine if the missing page is still missing. A page fault for a page in the shared segment (Useg3) may have been brought into RAM by another U-proc while the current U-proc was waiting for mutual exclusion to be granted. If the missing page is no longer missing, release mutual exclusion and return control to the U-proc's execution stream.

4. Select a frame to be used for this page fault. For our simple Pager it is sufficient to use the "Oldest Page First" frame selection algorithm. Some refer to this algorithm as the "Round Robin" frame selection algorithm; first select frame 0, then 1, 2, ..., 8, 9, 0, 1, etc.

5. If the frame is occupied mark the appropriate PTE entry to indicate that the entry is invalid. A copy of this entry may be sitting in the TLB. It is necessary to mark this entry as invalid as well. The simplest way to accomplish this is to issue a **TLBCLR** instruction. (See Section 6.3.1-*pops*/Section 4.2-*uarmdoc* on how to do this in C.) Those wishing to implement a more sophisticated Pager may optionally perform a TLB-Probe to find the matching entry in the TLB, and if present to alter the entry in the TLB.

   Regardless, altering both the designated PTE entry and the TLB must be done atomically. (Remember that the VM-I/O support level exception handlers run with interrupts enabled.) An error can occur if the U-proc is interrupted after having updated the PTE but before updating the TLB. (Note: Given the load/store nature of RISC architectures like μARM, it is also insufficient to simply update the TLB first.) To alter both atomically one should disable interrupts (i.e. mask them by setting **Status.I**=1 & **Status.F**=1) before the two updates and then re-enable them after the two updates.

6. If the frame was occupied, assume it is dirty and write it out to the backing store device (DISK0). The backing store device needs to reserve `MAXPAGES` sectors (or blocks) for each U-proc plus an additional `MAXPAGES` sectors for pages from Useg3. How the backing store device's sectors are divided up among the U-proc's and Useg3 is left up to the OS author. There is no need to use a DMA buffer for this disk write since the physical RAM address is known and will not change mid-write; the executing U-proc holds mutual exclusion.

7. Read in the missing page from the backing store device. Where it is found on the device is left up to the OS author; see above point. As with the disk write, there is no need to use a DMA buffer for this disk read since the physical RAM address is known and will not change mid-read; the executing U-proc still holds mutual exclusion.

8. Update the swap pool data structure to reflect the new occupant of the given frame in the page pool.

9. Update the appropriate PTE to reflect that the page is now sitting in RAM. (i.e. Mark the **CP15.PTE_Low** field in the page's PTE to indicate that the entry is writable (i.e. dirty), valid, and located at the selected **PFN**.) The TLB also needs to be updated as well. Since a TLB-Invalid exception can only occur when there was a matching entry in the TLB at the time of the exception, not updating the TLB might lead to an infinite loop of TLB-Invalid exceptions.

   Furthermore, as with flushing the previous frame's contents to the backing store device, updating the PTE and the TLB needs to be done atomically.

   The simplest way to update the TLB (i.e. flush the TLB of its invalid entry) is to issue a **TLBCLR** instruction. Those wishing to implement a more sophisticated Pager may optionally perform a TLB-Probe to find the matching entry in the TLB, and if present to alter the entry in the TLB.

10. Release mutual exclusive access to the paging data structures and return control to the U-proc's execution stream; μARM's re-attempt to translate a virtual address into a physical one.

A More Sophisticated Pager

As described above, there are a number of improvements those wishing to implement a more sophisticated Pager can take; mark frames containing nucleus **.text** and **.data** areas as not present and/or update a TLB entry directly instead of invalidating the complete TLB. Orthogonally, one may mark frames containing U-proc **.text** pages as read-only and U-proc **.data** pages as writable.

Another improvement would be to relax the assumption that all pages are dirty and need to be written to the backing store device. Unfortunately, μARM does not automatically update a “dirty” bit whenever a page is written to. One can nonetheless simulate this using an auxiliary data structure. Whenever a page is brought into RAM do not mark its PTE **CP15.PTE_Low** entry as dirty. Therefore, whenever a U-proc attempts to write on such a page a TLB-Modification TLB exception will occur. Now, instead of performing a SYS20 on the offending U-proc, the fact that the page is now dirty must be recorded in the auxiliary data structure. Furthermore, the dirty bit in both the relevant PTE and TLB entries need to be turned on as well (atomically of course). Now when a selected frame is to be vacated for an incoming page, its current contents only need to be written out to the backing store device if indeed the page was dirty.

Finally, the “Oldest Page First” frame selection algorithm can be replaced with something a bit more sophisticated.

B.3.6 VM-I/O Support Level Initialization

After the nucleus concludes its initialization, control passes to `test`. This process/routine has a number of important tasks to complete:

1. Initialize the single Kseg0 PTE.
2. Initialize the single Useg3 PTE.
3. Initialize all VM-I/O support level semaphores.
4. Initialize the `ADL` module.
5. Initialize and launch the delay daemon process; this includes initializing the appropriate entries in its segment table.
6. Initialize the `AVSL` module.
7. Initialize the swap-pool data structure(s).
8. Initialize and launch each U-proc. See Section B.3.7.
9. Go to sleep until all U-proc’s have terminated. One way to accomplish this is to block `test` on a semaphore (i.e. the `masterSem`) until all the U-proc’s have terminated. Since U-proc termination is performed in a controlled manner, SYS20, it is a simple matter to know when the last U-proc has terminated. When this happens, the U-proc termination routine can simply V the `masterSem`, unblocking `test`.

10. Invoke a SYS2 to kill the Delay Daemon.
11. Invoke the SYS20 Terminate service to halt the OS.

B.3.7 U-proc Initialization

Launching a U-proc is a complicated two-step process.

U-proc Initialization: Step 1 - variable initialization

The first step in launching a U-proc is to initialize various data structures and perform a SYS1 operation:

1. Assign the U-proc a unique ASID.
2. Initialize the U-proc's Useg2 PTE.
3. Initialize the U-proc's segment table.
4. Initialize the U-proc's private semaphore.
5. Initialize the U-proc's three (PgmTrap, TLB, and SYS/Bp) New (processor state) Areas. The six processor state areas to be used by each U-proc when it makes its SYS5 request are yet another VM-I/O support level implemented data structure. Each New Area should be initialized to be a state with all interrupts enabled, user-mode off and **CP15.R1**.VMc=1. The **SP** should be set to the appropriate stack page reserved for this U-proc's SYS/Bp or TLB exception handler respectively. Finally set the **PC** to the address of the respective VM-I/O support level exception handler and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID.
6. Since it is imperative that the SYS5 request be made while in kernel-mode and before virtual memory is enabled one needs to initialize a processor state appropriate for this goal. Initialize a processor state such that all interrupts are enabled, user-mode is off and **CP15.R1**.VMc=0. The **SP** should be set to one of the stack pages reserved for this U-proc's SYS/Bp or TLB exception handler. Finally set the **PC** to the address of the U-proc step 2 initialization function and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID.
7. Perform a SYS1 operation using the state in the above step.

U-proc Initialization: Step 2 - SYS5 Requests & Reading From the Tape

The second step in launching a U-proc is to issue the SYS5 request, read in the U-proc's **.text** and **.data** from the tape, and prepare for actual U-proc launch.

1. Determine the running U-proc's ASID; Perform a **getENTRYHI** and extract the ASID value.

2. Perform the SYS5 operation.

3. Read in the U-proc's **.text** and **.data** areas into the U-proc's area on the backing store device from the tape device associated with this U-proc. The contents of the tape are to be read contiguously. The first block is page 0 for the U-proc's Useg2, the second block is page 1, and so on. As described in Section 5.4-*pops* the end of a file on a tape is denoted by an **EOF** marker.

4. Prepare a processor state appropriate for the execution of a U-proc. To do this initialize a processor state such that all interrupts are enabled, user-mode is on and **CP15.R1**.VMc=1. The **SP** should be set to the last page of Useg2. Finally set the **PC** to `0x8000.0054` (i.e. The contents of the second word in Useg2) and **CP15.PTE_Hi.ASID** to the U-proc's assigned ASID. (See Section 8.3-*pops* for an explanation as to why the **PC** are set to this value instead of just the beginning of Useg2.)

5. Perform a **LDST** operation using the state prepared in the above step to finally begin executing the code associated with this U-proc.

   Note: Immediately following this **LDST** the U-proc will experience two page faults. The first one will be for the stack page (the last page in Useg2) and the second will be for the first page of code (the first page of Useg2). The U-proc's PTE was initialized to indicate that neither was "present". The backing store device contains the **.text** and **.data** contents – initialized when the U-proc's tape contents were read in. (i.e. Initial contents of the first n pages of Useg2.) The page on the backing store representing the stack page contains uninitialized junk - which is perfectly fine for a stack page for a newly created process.

   A nice but tricky optimization would be to avoid reading this stack page in from the backing store device for this page fault only. Similarly, when reading in page 0 from tape, in addition to writing it out to the backing store, also place it into a free frame.

B.3.8 Nuts and Bolts

Performing DMA I/O Operations

As described in Chapter 5-*pops* the disk and tape devices utilize DMA to read and write directly from/to RAM. The address for a DMA I/O operation, specified in the device's **DATA0** device register field, must be a physical address. I/O device controllers operate independently of the *CP15* co-processor and by extension the virtual memory address translation facility.

When a U-proc requests to read a block from a disk, the address of a contiguous physical 4KB area must be provided. Even assuming the U-proc supplied virtual address is currently in RAM and that the page containing this address can be locked into its current frame (yet another level of sophistication for the Pager), the 4KB block will likely spill over into the next page in the virtual address space. There is no guarantee that the frame holding the succeeding page, if even present, sits immediately after the frame holding the page containing the beginning of the block.

Instead the VM-I/O support level will provide an individual 4KB buffer for each DMA supporting device. (See Section B.3.5.) The DMA supporting devices will read/write directly from/to their assigned buffer. It is the task of the VM-I/O support level I/O routines to copy the data to/from these buffers from/to the specified virtual address space.

For example for a disk write request the VM-I/O support level I/O routine would, after validating the virtual address, copy the 4KB from the virtual address space into the designated device's assigned buffer. After this is done, the disk write operation can proceed.

Note that the VM-I/O support level's I/O routine (part of the SYS/Bp exception handler) will be running with **CP15.R1**.VMc=1 and its ASID set to the current U-proc. From one perspective the copy operation is from one virtual address (in Useg2 or Useg3) to another virtual address (in Kseg0). Any page faults that occur or the fact that the 4KB source block is not necessarily contiguous in RAM is automatically taken care of by virtual address translation. From another perspective, given the way the Kseg0 PTE was constructed, the copy operation is a from a virtual address to a physical address.

Note: It is not necessary to use the VM-I/O support level DMA buffer for paging related disk I/O. Paging I/O always begins on a frame boundary and because of mutual exclusion held by the process within the Pager, the physical RAM page is effectively locked. Hence paging related disk I/O can be performed directly to/from the physical frames in the page pool. This is also true for U-proc initialization, the buffer for a tape device (filled via a tape read operation) can be used as the source for backing store disk write operation.

Managing Concurrency

Level 4 of JaeOS represents a timeshare system where many U-proc's along with some system daemons run concurrently. These processes can simultaneously be executing code in the same VM-I/O support level routine. There are two questions that need to be addressed: how can two U-proc's be executing code in the same VM-I/O support level routine at the same time; and how to prevent a race condition between two or more processes wishing to access the same VM-I/O support level data structure (e.g. the ADL, the AVSL, or the swap-pool data structure)?

The first question isn't an issue since each VM-I/O support level handler is implemented re-entrantly; each process that executes the code of one of these handlers has its own stack frame and hence its own copy of all local variables.

Race conditions can be explicitly avoided through the use of controlling semaphores. For each shared VM-I/O support level data structure there should be a semaphore defined for it that is initialized to one. Before an attempt to access (both read and write) a shared data structure one must first request a SYS4 operation on the data structure's controlling semaphore. The description in Section B.3.5 provides an example of this for the swap-pool/Pager structures.

There should be one semaphore for the delay facility, the virtual P & V facility, the swap-pool/Pager service and one for each device's device registers. Remember that terminals are actually two independent sub-devices. Hence each terminal has two device register controlling semaphores.

Two Stacks per U-proc At The VM-I/O support level Explained

In the nucleus each exception handler is independent of each other. This is why all four nucleus exception handlers can use the same stack frame. At the VM-I/O support level it is possible for the VM-I/O support level SYS/Bp exception handler to generate a TLB exception (i.e. page fault). Consider a SYS13 (V) request where the increment of the semaphore causes a page fault since the page containing the semaphore is not present in RAM.

To handle these nested exceptions one needs independent stacks for the SYS/Bp and TLB exception handlers for each U-proc. The VM-I/O support level SYS/Bp exception handler generating a TLB exception is the only case where nested exceptions can occur though. The VM-I/O support level TLB exception handler will not make a SYSCALL that gets passed up, and hopefully neither the VM-I/O support level TLB or SYS/Bp exception handler will generate a PgmTrap exception. A separate stack is not needed for the VM-I/O support level PgmTrap exception handler. Either the VM-I/O support level's TLB or SYS/Bp exception handler's stack can be re-used as the VM-I/O support level's PgmTrap exception handler's stack frame.

The VM-I/O support level Exception Identity Question

When control passes to a VM-I/O support level's exception handler, the handler needs to know its ASID. Though there are separate stack frames for each U-proc, making the handlers reentrant, the code (i.e. `text`) is nonetheless the same for each U-proc. If during U-proc initialization (see Section B.3.7) each U-proc initializes its three New (processor state) Areas to contain the U-proc's ASID, each VM-I/O support level exception handler, on entry, can easily learn its ASID. (Perform a **getENTRYHI** and extract the ASID value.)

Module Decomposition

One possible module decomposition is as follows:

- INITPROC.C This module implements `test()` and all the U-proc initialization routines. It exports the VM-I/O support level's global variables. (e.g. swap-pool data structure, mutual exclusion semaphores, etc.)
- ADL.C This module implements the Active Delay List module.
- AVSL.C This module implements the Active Virtual Semaphore List module.
- PAGER.C This module implements the VM-I/O support level TLB exception handler; the Pager.
- SYSSUPPORT.C This module implements the VM-I/O support level SYS/Bp and PgmTrap exception handlers.

Accessing the `libuarm` Library

As described in Chapter 4-*uarmdoc*, accessing the **CP15** registers and the ROM-implemented services/instructions in C is via the `libuarm` library. Simply include the line

```
#include ‘‘/usr/include/uarm/libuarm.h’’
```

The file LIBUARM.H is part of the μARM distribution.
/USR/INCLUDE/UARM/ is the recommended installation location for this file. Make sure you know where it is installed in your local environment and alter this compiler directive appropriately.

Testing

There is a set of provided test U-proc programs that will “exercise” your code.

- HELLOTEST.C - Hello World program for phase 3.

- swapTest.c - Forces the page pool to fill to generate page faults.
- todTest.c - Tests the delay facility.
- diskTest.c - Tests U-proc disk I/O.
- rogueTest.c - Tests the fault tolerance of the OS trying to do forbidden operations.
- tictactoe.c - Tic Tac Toe game, larger executable, tests .text section spanning on multiple pages.
- printerTest.c - Tests writing to the printer.
- fibTest.c - processor intensive job; calculate Fib(24) and Fact(12) recursively.
- readTest.c & multTest.c - Tests for correct terminal input.
- consumerTest.c & producerTest.c - Tests the VM-I/O support level P & V facility. These two programs must be run together.
- consDiskTest.c & prodDiskTest.c - Tests the VM-I/O support level P & V facility that also use Disk1 for synchronization, .data section is large and spans over multiple pages. These two programs must be run together.
- ulib.c - A utility module used by all of the above test programs to facilitate syscall requests and terminal printing.

Additionally, it is easy to write one's own programs to run under JaeOS. Being able to run your own programs under your own OS is half the fun of completing the project anyway.

You should individually compile all the source files from phase1, phase2, and phase3 in addition to the phase3 U-proc test programs using the command:

`arm-none-eabi-gcc -pedantic -Wall -mcpu=arm7tdmi -c filename.c`

All of the OS object files should then be linked together using the command:

```
arm-none-eabi-ld -T
   /usr/include/uarm/ldscripts/elf32ltsarm.h.uarmcore.x
   /usr/include/uarm/crtso.o /usr/include/uarm/libuarm.o
   phase1, phase2 & phase3 .o files -o kernel
```

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μARM. This is done with the command:

`elf2uarm -k kernel`

which produces the file: kernel.core.uarm

Each test program should individually be linked. The following is an example for swapTest.o

```
arm-none-eabi-ld -T
    /usr/include/uarm/ldscripts/elf32ltsarm.h.uarmaout.x
    /usr/include/uarm/crti.o /usr/include/uarm/libuarm.o
    print.o swapTest.o -o swapTest
```

The linker produces a file in the ELF object file format which needs to be converted prior to its use with μARM. This is done with the command:

```
elf2uarm -a swapTest
```

which produces the file: swapTest.aout.uarm
Finally, the linked file can be loaded onto a tape cartridge with the command:

```
uarm-mkdev -t swapTape.uarm swapTest.aout.uarm
```

which produces the file: swapTape.uarm

The files elf32ltsarm.h.uarmcore.x, lf32ltsarm.h.uarmaout.x, crtso.o, crti.o and libuarm.o are part of the μARM distribution. /usr/include/uarm and /usr/include/uarm/ldscripts are the recommended installation locations for these files. Make sure you know where they are installed in your local environment and alter this command appropriately. The order of the object files in the link commands is important: specifically, the first two support files must be in their respective positions.

Finally, your OS can be tested by launching μARM. Entering:

```
uarm
```

without any parameters loads the file kernel.core.uarm by default. Use the settings panel inside the emulator to load tapes and disks.

Hopefully the above will illustrate the benefits for using a Makefile to automate the compiling, linking, and converting of a collection of source files into a μARM executable file.