Thesis for the Degree of Master of Science (20 credits)

A debugger with GUI in OS X

by

Iván A Krizsán, 670308-1411 GU

Supervisor: Per Håkan Sundell

Examiner: Patrik Jansson, Gothenburg University

Department of Computing Science
Chalmers University of Technology and Gothenburg University
2003-2004

Abstract

This thesis describes the process of designing and implementing a debugger with a graphical user interface that is to execute under OS X on Apple Macintosh computers.

Desired functionality have been specified, taking into account the experience of some debuggers I have had previous experience with.

The areas requiring investigation have been specified and investigated one by one and smaller programs that solves the specific problem have been written and tested.

These areas are:

- Launching a program under the control of the debugger.
- Attaching the debugger to an executing program that was not launched by the debugger.
- Suspend and resume execution of the target program.
- Target program signal and exception handling in the debugger.
- Stepwise execution of the target program.
- Breakpoints; points in the target program at which execution is to be stopped.
- Monitoring of the target program reading and/or writing of selected addresses in memory , i.e. watchpoints.
- Managing threads in the target program.
- Reading and writing of object files, i.e. files containing executable code.
- Disassembling machine code.
- Accessing CPU register contents of the target program.
- Accessing target program memory.
- Interpretion of machine code instructions.
- Design that facilitates porting of the debugger to other platforms and future changes.

The work has been limited to finding solutions to in the thesis mentioned issues, not to compare different solutions of the same problem.

The result is that solutions to all above mentioned areas have been found, although the solutions, in some cases, does not perform optimally due to the nature of the problem and the lack of hardware support on the platform in question.

The size of the project and the limited time available for a master's thesis have prevented the completion of the design of the debugger and the implementation of more than very basic functionality.

The work remaining to implement the remaining features of the debugger is sketched out under the conclusion section in this thesis.

Sammanfattning

Denna rapport beskriver design av och implementering av en debugger med ett grafiskt användar-gränssnitt som skall kunna användas under OS X på Apple Macintosh datorer.

Önskad funktionalitet har specificerats, med mina tidigare erfarenheter från användandet av debuggers i åtanke.

Områden som kräver undersökning har specificerats och undersökts ett för ett under det att mindre program som löser det specifika problemet har skrivits och testats.

Problemområden som behandlats är:

- Start av ett program under debuggerns kontroll.
- Koppling av debuggern till ett program som exekverar och som har startats utanför debuggerns kontroll.
- Stanna och återuppta exekveringen av ett program som debuggas.
- Hantering av signaler och undantag från programmet som debuggas.
- Stegvis exekvering av programmet som debuggas.
- Brytpunkter; punkter i programmet som debuggas där exekveringen skall avbrytas.
- Övervakning av programmet som debuggas läsning och/eller skrivning till utvalda minnesaddresser, dvs. watchpoints.
- Hantering av trådar i programmet som debuggas.
- Läsning och skrivning av objekt-filer, dvs. filer som innehåller exekverbar kod.
- Disassemblering av maskinkod.
- Åtkomst av innehåll i CPU register i programmet som debuggas.
- Tillgång till programmet som debuggas minne.
- Tolkning av maskinkods-instruktioner.
- Design för att underlätta överföring av debuggern till andra plattformar samt för att underlätta framtida ändringar i programmet.

Arbetet har begränsats till att hitta lösningar på i rapporten nämnda områden, inte att jämföra olika lösningar av samma problem.

Resultatet är att lösningar till alla ovan beskrivna problemområden har kunnat hittas och implementeras, fastän lösningarna i vissa fall inte är optimalt effektiva på grund av problemets natur och bristen på hårdvaru-stöd på plattformen i fråga.

Projektets storlek och den begränsade tiden tillgänglig för ett examensarbete har förhindrat färdigställande av designen av debuggern och implementering av mer än en grundläggande funktionalitet.

Det arbete som krävs för att implementera återstående funktionalitet i debuggern skisseras under slutlednings-avsnittet i denna rapport.

Preface

This document contains a report of a graduation work for a master's degree in Computing Science at Gothenburg University and Chalmers University of Technology.

The work has been supervised by Per Håkan Sundell from the Institution of Computing Science.

I want to point out that, prior to this project, I had never programmed in Objective-C, I had never used Cocoa (a set of object-oriented frameworks used when developing software for OS X), I had only very basic knowledge and experience of programming under UNIX and I had never developed software to be used under OS X.

Table of Contents

Section 1 – Introduction

The aim of this project is to construct a program for analyzing and debugging compiled object code. Ideally, this program should be able to transparently monitor the execution of another program. A glossary is found at the end of the report.

1.1 Background And Motivation

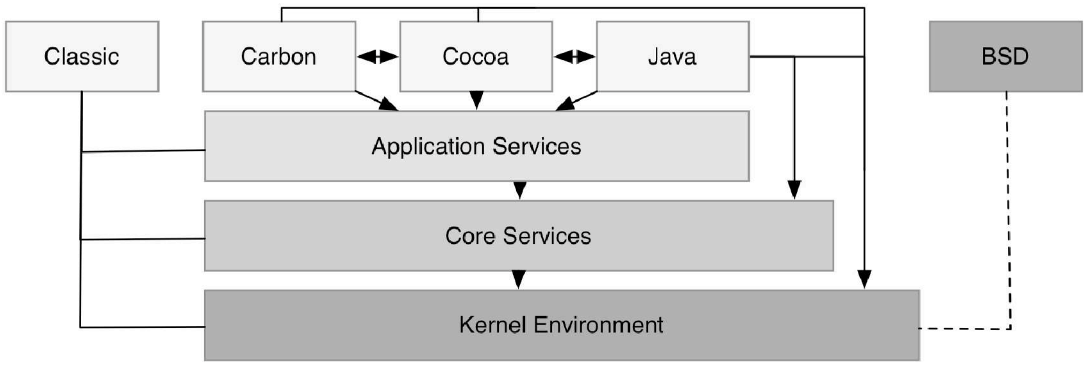
During the time I have been developing software for computers, I have come to appreciate debugging at object code level, since it gives insights about the most fundamental operation of a program. Examining and understanding the object code produced by a compiler also gives me the opportunity to adapt my source code in order to produce a more efficient program. Reading object code has also been very instructive. The tools I have used have usually had text based interface and/or have only had a subset of the desirable functionality. The main motivation for this project is the introduction of GDB – Gnu DeBugger – as the debugger of choice in OS X. GDB is an excellent program, but not very easy to use, in my opinion.

1.2 Presentation Of The Environment

This section contains a very brief presentation of the environment in which the debugger is intended to work, namely the operating system OS X and the PowerPC CPU architecture.

1.2.1 OS X

The following information about the operating system OS X is a brief version of information presented in [0] as well as some additional information found on the webpages of Apple Computer[1].



Simplified overview of the OS X architecture.

The above figure presents a simplified overview of the OS X architecture.

The top layer contains the different environments in which applications execute. Each process process is assigned its own virtual 4GB address space having the base address zero by the kernel.

Classic	- The environment providing backwards compatibility for applications developed for versions of the operating system prior to OS X.
Carbon	- An adaption of the programming environment of the versions of the operating system prior to OS X.
Cocoa	- An object oriented programming environment for developing programs in Objective-C and Java.
Java	- A programming environment for developing programs in Java.
BSD	- An optional environment for executing BSD command line programs.

The application services layer provides services for applications in the Carbon, Cocoa and Java environments, as described above. This layer includes services such as graphics services, windowing services and event handling. These services are provided in dynamically linked libraries, called through stubs.

[1]http://www.apple.com

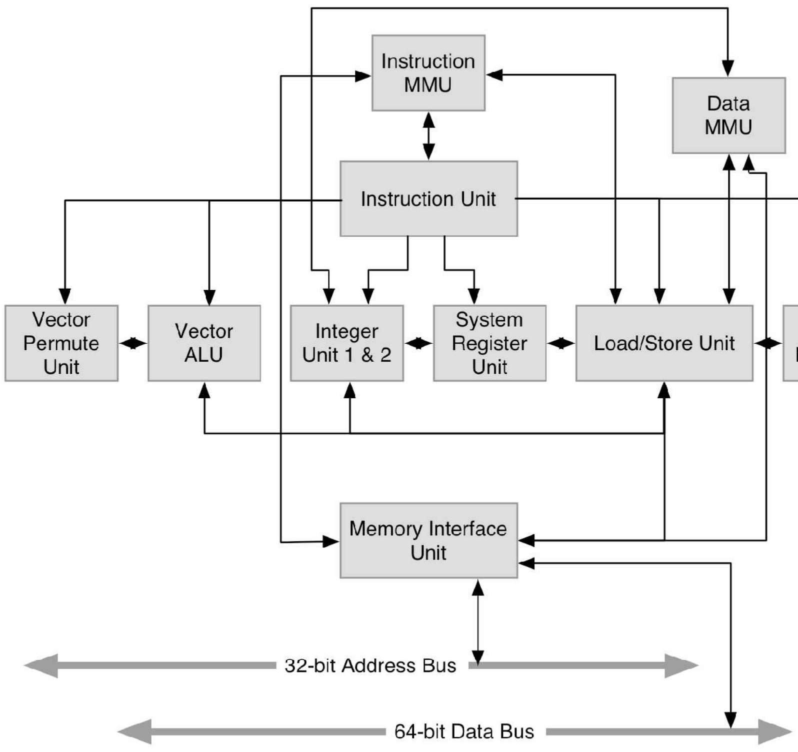
The core services layer provide services that do not fall under the graphical user interface category such as services for basic abstractions, such as strings, different kinds of collections etc., services for the management of processes, threads, virtual memory and other resources. These services are also provided in dynamically linked libraries, called through stubs.

The kernel environment is the lowest level of the system software and provides essential, basic, operating system services to the above layers, hiding the hardware – only code executing in the kernel is allowed to access hardware directly. The different components of the kernel are linked together into a single kernel address space. The kernel address space is "wired" into physical memory, i.e. it cannot be paged out. It occupies approximately 60MB.

The kernel environment consists of the following major parts:

Mach	- A Mach kernel providing fundamental kernel services for process, thread, memory management and interprocess communication.
BSD	- A BSD4.4 kernel supporting the Mach kernel and providing basic networking and file system services.
Device drivers and the IO kit	- An environment that provides an object oriented framework for developing device drivers.
Networking	- Providing networking protocols and services.
File systems	- Provides support for different kinds of file systems.

1.2.2 The PowerPC CPU Architecture



Simplified block diagram of the G4 PowerPC CPU.

The PowerPC CPUs are reduced instruction set computing (RISC) 32-bit CPUs with 32 general purpose registers, 32 floating point registers and additional special purpose registers. It includes, among other units, a floating-point unit (FPU) and a memory management unit (MMU).
The special purpose registers accessible from user mode, i.e. code executing outside of the kernel, are:
CR (condition register), FPSCR (floating point status and control register), XER, LR (link register) and the CTR (count register).
Support for 64-bit addressing is also included, but it is only available in the G5 processor.

In the Apple Macintosh computers, a number of different models of the PowerPC CPUs have been used. The operating system in question, OS X, is currently only able to run on processors commonly referred to as G3, G4 and G5.

1.3 Aim

The intention is to write a debugger for analyzing and debugging of object code that is to execute under OS X, aiming to include not only common functionality of debuggers, but also additional features. Such features include documentation of the object code, which can be stored and retrieved at some later point in time, as well as extended control over the execution of the target program facilitated by interpreting its machine instructions. The functionality of the debugger is to be accessible using a graphical user interface (GUI), in order to facilitate ease of use and flexibility. The intention is also to design the debugger in such a way as to make it easily extendable and to make porting to other platforms possible.

Due to limited time for the project, a only a subset of the functionality will be implemented, however the intention is to investigate all areas requiring investigation.

Some other limitations are:
- The debugger will only be able to debug one program at a time.
- The debugger will only support loading of programs using the Mach-O runtime architecture, as documented in [1].
- The debugger will only use available kernel application programming interfaces (APIs), i.e. the kernel will not be modified or patched in any way.

1.4 Development Method

The program will be developed using an object-oriented approach as described in [2]. This approach was chosen having had positive experiences of it from two larger projects previously undertaken.

Section 2 – Project Definition

2.1 Requirements

2.1.1 Functional Requirements

- Attaching to a process that is currently executing.
- Reading and writing a program object file from/to secondary storage.
- Disassembling of target program object code.
  - Dividing the code in functions.
    Performed automatically by the debugger.
  - Extraction of information present in the target program object file such as symbol names and locations etc.
  - Documentation of disassembled target code.
    User can add comments in the code, change function names and group functions together. This information may be saved, along with additional information about the target program, such as symbols, breakpoints etc., and may be retrieved when the user resumes debugging of the program at some later point in time.
  - Examination and, optionally, modification of the target program object code or data. Modifications made prior execution of the target program being started may be saved.
  - Finding references to a function.

If time available:
- Examination of the call chain of a given function, i.e. references to referring functions etc.
- Analysis of object code from two programs in order to find similarities.
  This may be used as a tool for discovering code-theft.

- Debugging of the target program while executing.
  - Stepwise execution of machine instructions with the ability to treat a function-call as a single machine instruction (step over) or not (step in).
  - Breakpoints.
    - Disabling and enabling of breakpoints.
    - Only allow one breakpoint per address in the object code.

If time available:

- Silent breakpoints, i.e. breakpoints that cause a command to be executed when breakpoint is hit without halting the execution of the target program.
- Conditional breakpoints.
  - A counter associated to a breakpoint specifying how many times to halt execution when breakpoint is hit, before breakpoint is disabled.
  - A counter associated to a breakpoint specifying how many times to ignore breakpoint hits, before enabling the breakpoint.

- Watchpoints, including conditional watchpoints.
  - Halting of target execution upon access of by user specified memory location(s), specifying either read and/or write access.
  - Disabling and enabling of watchpoints.

If time available:

- Profiling of functions.
- A counter associated to a watchpoint specifying how many times to halt execution when watchpoint is hit, before watchpoint is disabled.
- A counter associated to a watchpoint specifying how many times to ignore watchpoint hits, before enabling the watchpoint.

- Examination and, optionally, modification of the target program object code or data residing in memory. Changes may not be saved.
- Examination of CPU registers and flags and, optionally, modification of register contents.
- Provide exception and signal handling for the target program.

If time available:

- More elaborate examination of the stack..
- Examination of target program memory contents using user-definable structure templates.
- Examination of target program memory contents using predefined data types such as floating-point etc.

2.1.2 External Interface Requirements

The debugger will have a graphical user interface (GUI).

Functions will be invoked by buttons in the window to which the function is relevant. This was chosen over in-window menus to give a larger freedom when choosing the language in which to implement the GUI, since in-window menus are not recommended under the Macintosh OS X, see [3]. Operations such as Cut, Copy, Paste and Find/Replace will be placed in the application's menus, as common in applications under the Macintosh OS X.

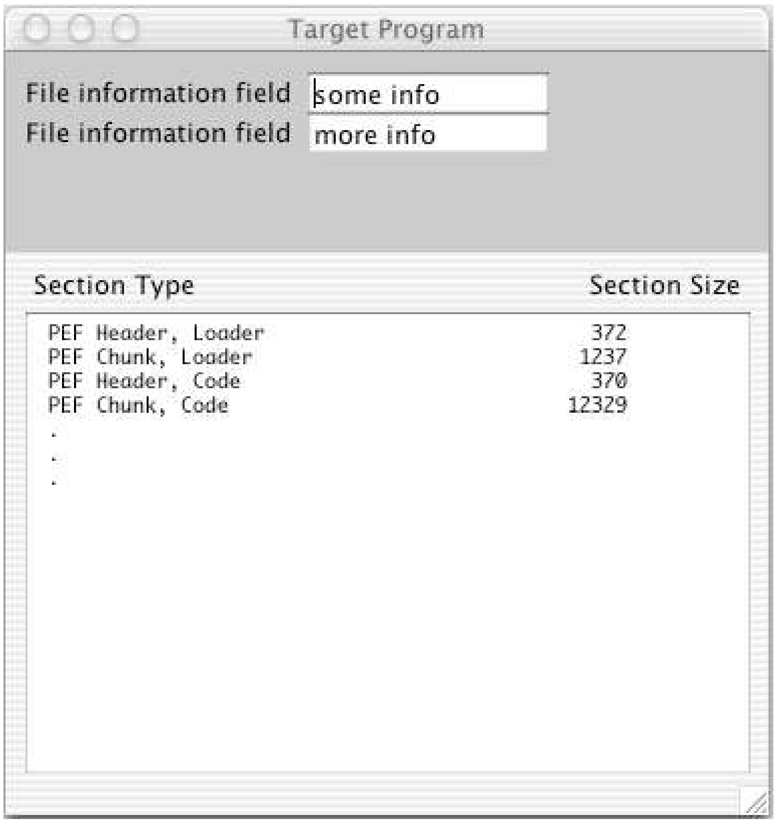
A separate window containing buttons will be used for controlling the execution of the target program.

Commonly used functions will also be accessible by using keyboard shortcuts.

Windows presenting large quantities of data will have functions for searching, and if applicable, replacing, in the data. The contents of windows displaying data may be exported to a file.

These are the main windows of the debugger:

File window
The file window will contain a list of the different parts of a program file such as code segment, data segment etc. This window has been postponed for a later development cycle.



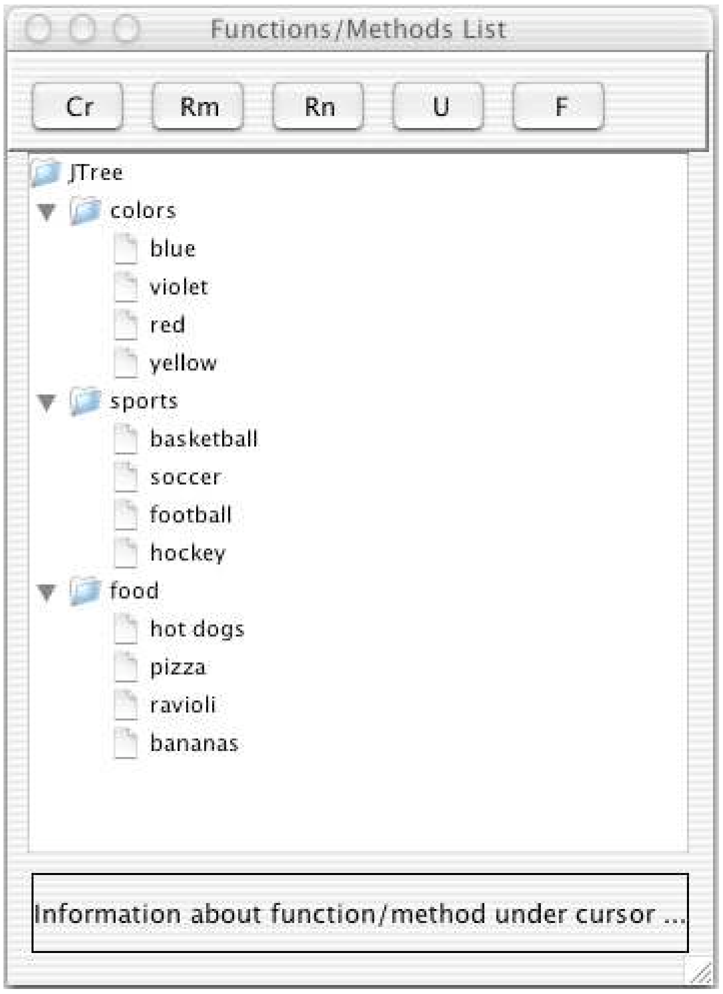
File information field some info
File information field more info

```
Section Type                                Section Size
PEF Header, Loader                                   372
PEF Chunk, Loader                                   1237
PEF Header, Code                                     370
PEF Chunk, Code                                    12329
.
.
.
```

File window prototype.

Functions window

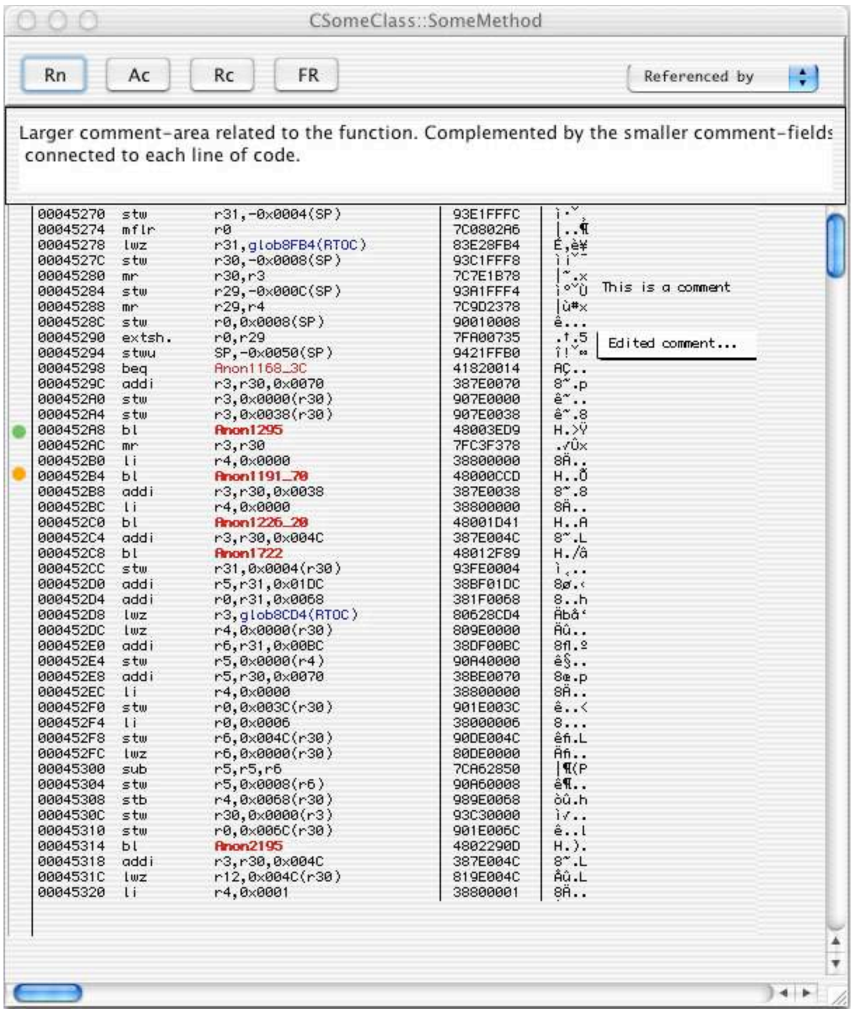
The functions window will contain the list of the functions of the target program and any function groups created by the user.



Functions window prototype.

Visited functions window
The visited functions floating window will contain a list of a user specifiable number of last visited functions for quick access. This window has been postponed for a later development cycle.

Disassembly window
The disassembly window will contain disassembly of object code, the hexadecimal code of the object code and the ASCII representation of the object code. Breakpoints related to specific addresses in the target program, along with their current status (enabled/disabled), will be displayed. Each disassembly window disassembles a single function. A popup-menu containing a list of the functions calling the disassembled function will facilitate fast and easy access of referring functions.
The user shall be able to decide whether to be able to view several disassembly windows at the same time or one single disassembly window. The latter facility has been postponed to a later development cycle.

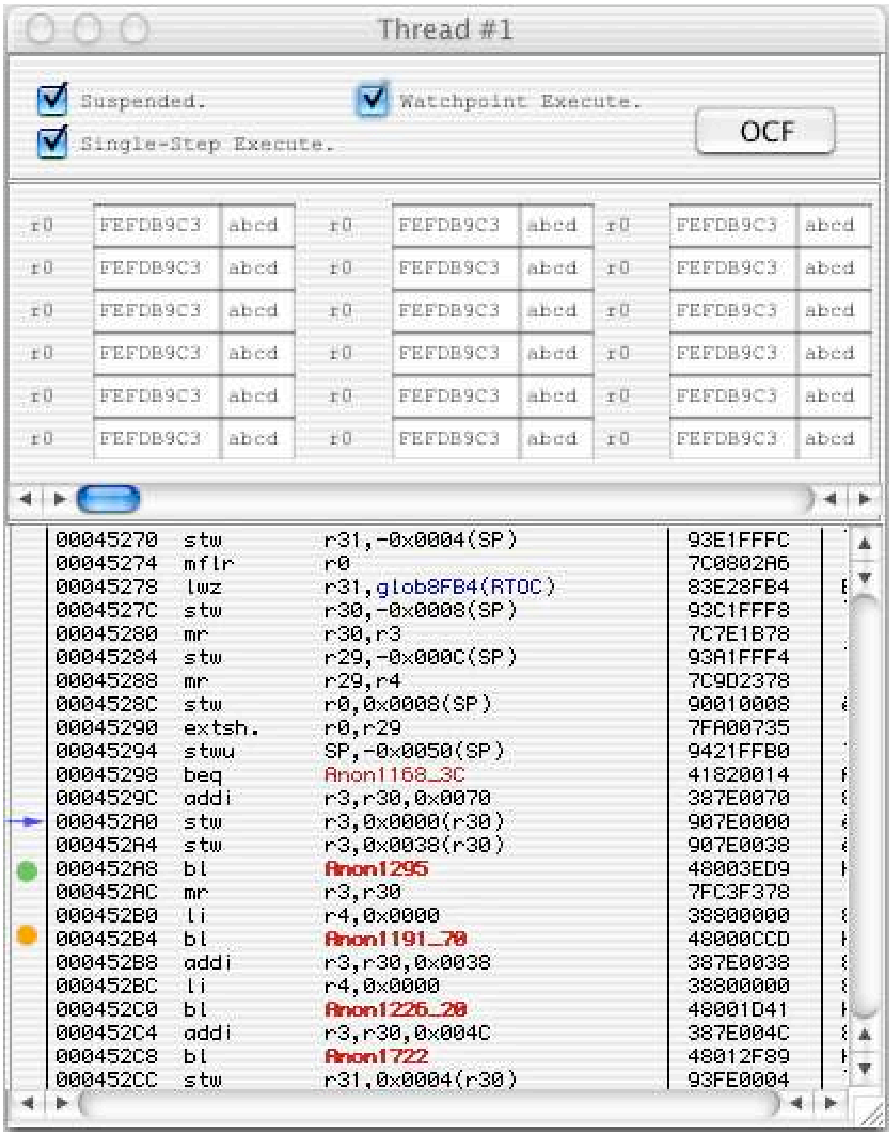


Larger comment-area related to the function. Complemented by the smaller comment-fields connected to each line of code.

```
00045270  stw     r31,-0x0004(SP)      93E1FFFC  ì·˘¸
00045274  mflr    r0                   7C0802A6  |..¶
00045278  lwz     r31,glob8FB4(RTOC)   83E28FB4  É,è¥
0004527C  stw     r30,-0x0008(SP)      93C1FFF8  ìi˘¯
00045280  mr      r30,r3               7C7E1B78  |~.x
00045284  stw     r29,-0x000C(SP)      93A1FFF4  ì°˘Ù   This is a comment
00045288  mr      r29,r4               7C9D2378  |ù#x
0004528C  stw     r0,0x0008(SP)        90010008  ê...
00045290  extsh.  r0,r29               7FA00735  .†.5   Edited comment...
00045294  stwu    SP,-0x0050(SP)       9421FFB0  î!˘∞
00045298  beq     Anon1168_3C          41820014  AÇ..
0004529C  addi    r3,r30,0x0070        387E0070  8~.p
000452A0  stw     r3,0x0000(r30)       907E0000  ê~..
000452A4  stw     r3,0x0038(r30)       907E0038  ê~.8
000452A8  bl      Anon1295             48003ED9  H.>Ÿ
000452AC  mr      r3,r30               7FC3F378  .√Ûx
000452B0  li      r4,0x0000            38800000  8Ä..
000452B4  bl      Anon1191_70          48000CCD  H..Õ
000452B8  addi    r3,r30,0x0038        387E0038  8~.8
000452BC  li      r4,0x0000            38800000  8Ä..
000452C0  bl      Anon1226_20          48001D41  H..A
000452C4  addi    r3,r30,0x004C        387E004C  8~.L
000452C8  bl      Anon1722             48012F89  H./â
000452CC  stw     r31,0x0004(r30)      93FE0004  ì˛..
000452D0  addi    r5,r31,0x01DC        38BF01DC  8ø.‹
000452D4  addi    r0,r31,0x0068        381F0068  8..h
000452D8  lwz     r3,glob8CD4(RTOC)    80628CD4  Äbå‘
000452DC  lwz     r4,0x0000(r30)       809E0000  Äû..
000452E0  addi    r6,r31,0x00BC        38DF00BC  8fl.º
000452E4  stw     r5,0x0000(r4)        90A40000  ê§..
000452E8  addi    r5,r30,0x0070        38BE0070  8æ.p
000452EC  li      r4,0x0000            38800000  8Ä..
000452F0  stw     r0,0x003C(r30)       901E003C  ê..<
000452F4  li      r0,0x0006            38000006  8...
000452F8  stw     r6,0x004C(r30)       90DE004C  êfi.L
000452FC  lwz     r6,0x0000(r30)       80DE0000  Äfi..
00045300  sub     r5,r5,r6             7CA62850  |¶(P
00045304  stw     r5,0x0008(r6)        90A60008  ê¶..
00045308  stb     r4,0x0068(r30)       989E0068  òû.h
0004530C  stw     r30,0x0000(r3)       93C30000  ì√..
00045310  stw     r0,0x006C(r30)       901E006C  ê..l
00045314  bl      Anon2195             4802290D  H.).
00045318  addi    r3,r30,0x004C        387E004C  8~.L
0004531C  lwz     r12,0x004C(r30)      819E004C  Åû.L
00045320  li      r4,0x0001            38800001  8Ä..
```

Disassembly window prototype.

Thread Window
The thread window will display the state of one thread of the target program; the contents of CPU registers and flags in hexadecimal and ASCII form and disassembly of a region surrounding the position in the program code at which the thread is currently executing. Controls for suspending and resuming the thread will be available, controlling whether the thread is to be executed or not when the application is single-stepped, and for controlling whether the thread is to be executed or not when one or more watchpoints are active. There will be one thread window for each thread in the target program.



✓ Suspended.
✓ Watchpoint Execute.
✓ Single-Step Execute.

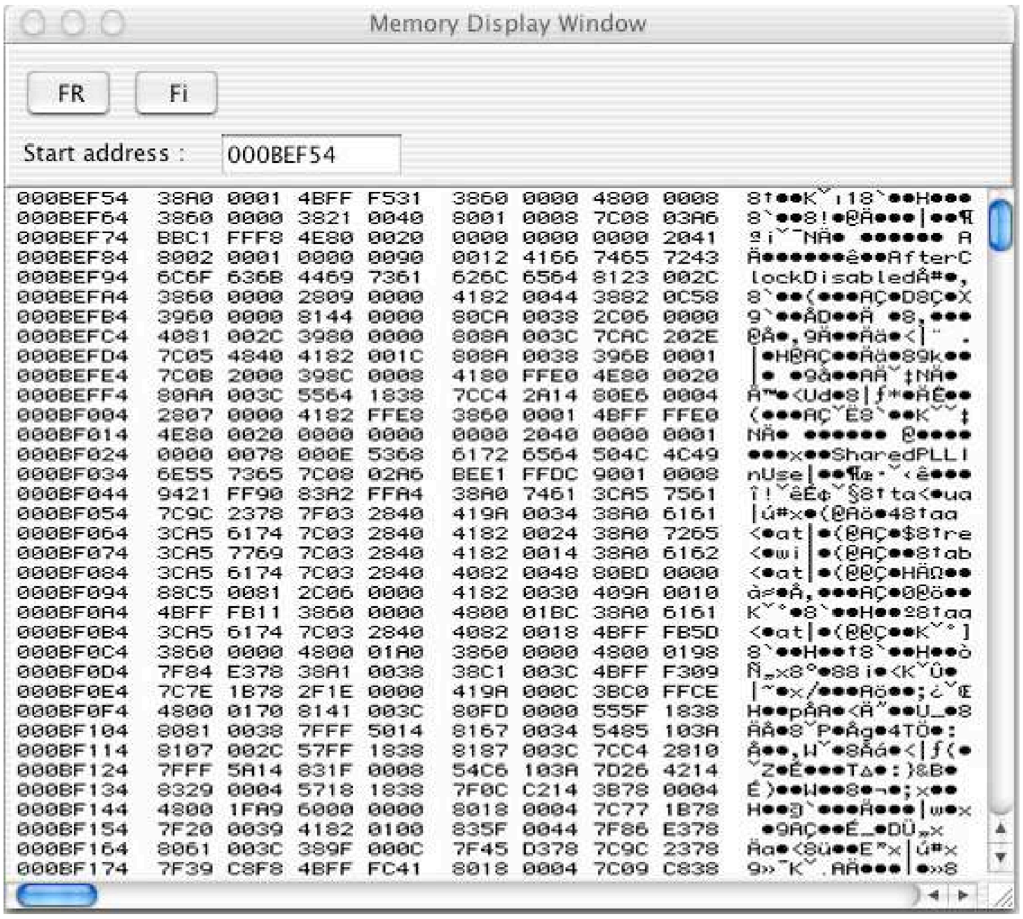
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd
r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd	r0	FEFDB9C3	abcd

```
  00045270  stw     r31,-0x0004(SP)     93E1FFFC
  00045274  mflr    r0                  7C0802A6
  00045278  lwz     r31,glob8FB4(RTOC)  83E28FB4
  0004527C  stw     r30,-0x0008(SP)     93C1FFF8
  00045280  mr      r30,r3              7C7E1B78
  00045284  stw     r29,-0x000C(SP)     93A1FFF4
  00045288  mr      r29,r4              7C9D2378
  0004528C  stw     r0,0x0008(SP)       90010008
  00045290  extsh.  r0,r29              7FA00735
  00045294  stwu    SP,-0x0050(SP)      9421FFB0
  00045298  beq     Anon1168_3C         41820014
  0004529C  addi    r3,r30,0x0070       387E0070
→ 000452A0  stw     r3,0x0000(r30)      907E0000
  000452A4  stw     r3,0x0038(r30)      907E0038
● 000452A8  bl      Anon1295            48003ED9
  000452AC  mr      r3,r30              7FC3F378
  000452B0  li      r4,0x0000           38800000
● 000452B4  bl      Anon1191_70         48000CCD
  000452B8  addi    r3,r30,0x0038       387E0038
  000452BC  li      r4,0x0000           38800000
  000452C0  bl      Anon1226_20         48001D41
  000452C4  addi    r3,r30,0x004C       387E004C
  000452C8  bl      Anon1722            48012F89
  000452CC  stw     r31,0x0004(r30)     93FE0004
```

Thread window prototype.

Memory window

The memory window will display memory contents of the target program in hexadecimal and ASCII form.



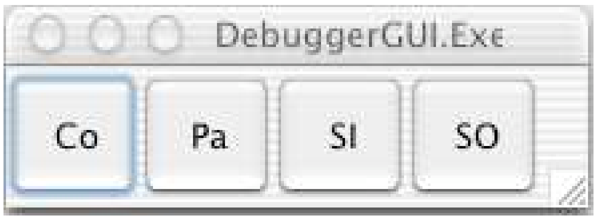
Start address : 000BEF54

```
000BEF54   38A0 0001 4BFF F531   3860 0000 4800 0008   8†●●Kˇ¡18`●●H●●●
000BEF64   3860 0000 3821 0040   8001 0008 7C08 03A6   8`●●8!●@Ä●●●|●●¶
000BEF74   BBC1 FFF8 4E80 0020   0000 0000 0000 2041   ªiˇ¯NÄ● ●●●●●● A
000BEF84   8002 0001 0000 0090   0012 4166 7465 7243   Ä●●●●●●●ê●●AfterC
000BEF94   6C6F 636B 4469 7361   626C 6564 8123 002C   lockDisabledÅ#●,
000BEFA4   3860 0000 2809 0000   4182 0044 3882 0C58   8`●●(●●●AÇ●D8Ç●X
000BEFB4   3960 0000 8144 0000   80CA 0038 2C06 0000   9`●●ÅD●●Ä ●8,●●●
000BEFC4   4081 002C 3980 0000   808A 003C 7CAC 202E   @Å●,9Ä●●Ää●<|¨ .
000BEFD4   7C05 4840 4182 001C   808A 0038 396B 0001   |●H@AÇ●●Ää●89k●●
000BEFE4   7C0B 2000 398C 0008   4180 FFE0 4E80 0020   |● ●9å●●AÄˇ‡NÄ●
000BEFF4   80AA 003C 5564 1838   7CC4 2A14 80E6 0004   Ä™●<Ud●8|ƒ*●ÄÊ●●
000BF004   2807 0000 4182 FFE8   3860 0001 4BFF FFE0   (●●●AÇˇË8`●●Kˇˇ‡
000BF014   4E80 0020 0000 0000   0000 2040 0000 0001   NÄ● ●●●●●● @●●●●
000BF024   0000 0078 000E 5368   6172 6564 504C 4C49   ●●●x●●SharedPLLI
000BF034   6E55 7365 7C08 02A6   BEE1 FFDC 9001 0008   nUse|●●¶æ·ˇ‹ê●●●
000BF044   9421 FF90 83A2 FFA4   38A0 7461 3CA5 7561   î!ˇêÉ¢ˇ§8†ta<●ua
000BF054   7C9C 2378 7F03 2840   419A 0034 38A0 6161   |ú#x●(@Aö●48†aa
000BF064   3CA5 6174 7C03 2840   4182 0024 38A0 7265   <●at|●(@AÇ●$8†re
000BF074   3CA5 7769 7C03 2840   4182 0014 38A0 6162   <●wi|●(@AÇ●●8†ab
000BF084   3CA5 6174 7C03 2840   4082 0048 80BD 0000   <●at|●(@@Ç●HÄΩ●●
000BF094   88C5 0081 2C06 0000   4182 0030 409A 0010   à≈●Å,●●●AÇ●0@ö●●
000BF0A4   4BFF FB11 3860 0000   4800 01BC 38A0 6161   Kˇ˚●8`●●H●●º8†aa
000BF0B4   3CA5 6174 7C03 2840   4082 0018 4BFF FB5D   <●at|●(@@Ç●●Kˇ˚]
000BF0C4   3860 0000 4800 01A0   3860 0000 4800 0198   8`●●H●●†8`●●H●●ò
000BF0D4   7F84 E378 38A1 0038   38C1 003C 4BFF F309   Ñ„x8°●88¡●<KˇÛ●
000BF0E4   7C7E 1B78 2F1E 0000   419A 000C 3BC0 FFCE   |~●x/●●●Aö●●;¿ˇŒ
000BF0F4   4800 0170 8141 003C   80FD 0000 555F 1838   H●●pÅA●<Ä˝●●U_●8
000BF104   8081 0038 7FFF 5014   8167 0034 5485 103A   ÄÅ●8ˇP●Åg●4TÖ●:
000BF114   8107 002C 57FF 1838   8187 003C 7CC4 2810   Å●●,Wˇ●8Åá●<|ƒ(●
000BF124   7FFF 5A14 831F 0008   54C6 103A 7D26 4214   ˇZ●É●●●T∆●:}&B●
000BF134   8329 0004 5718 1838   7F0C C214 3B78 0004   É)●●W●●8●¬●;x●●
000BF144   4800 1FA9 6000 0000   8018 0004 7C77 1B78   H●●©`●●●Ä●●●|w●x
000BF154   7F20 0039 4182 0100   835F 0044 7F86 E378   ●9AÇ●●É_●DÜ„x
000BF164   8061 003C 389F 000C   7F45 D378 7C9C 2378   Äa●<8ü●●E"x|ú#x
000BF174   7F39 C8F8 4BFF FC41   8018 0004 7C09 C838   9»˘K˘.AÄ●●●|●»8
```

Memory window prototype.

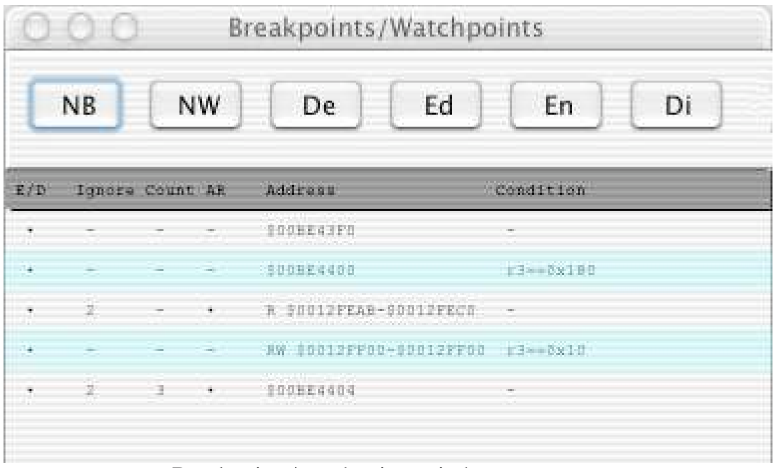
Execution control window

The execution control window will contain buttons for controlling execution of target program: continue, stop, step over and step in. This window will also be a floating window.



Execution control window prototype.

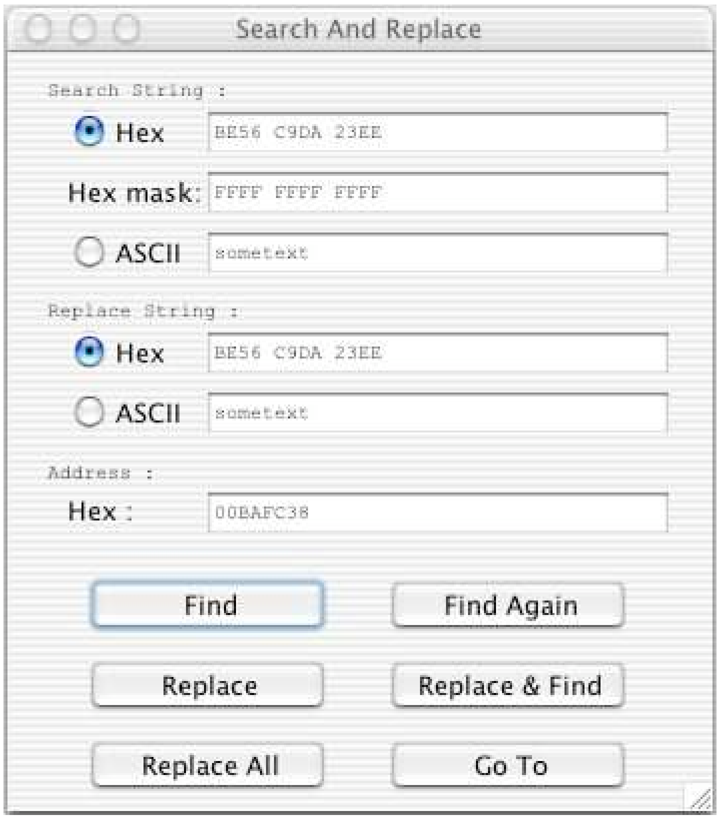
Breakpoints/Watchpoints window
The breakpoints/watchpoints window will display breakpoints and watchpoints created by the user, any conditions associated with each breakpoint/watchpoint and the status of the breakpoint/watchpoint.



E/D	Ignore	Count	AR	Address	Condition
•	–	–	–	$00BE43F0	–
•	–	–	–	$00BE4400	r3==0x180
•	2	–	•	R $0012FEA8-$0012FEC0	–
•	–	–	–	RW $0012FF00-$0012FF00	r3==0x10
•	2	3	•	$00BE4404	–

Breakpoints/watchpoints window prototype.

Find And Replace Window
The find and replace window is invoked when the user wants to perform a search or a search-and-replace operation either in the Disassembly Window or in the Memory Window. A Find menu in the application menus serves as an alternative way of accessing the same functionality. Searching and replacing data has been postponed to a later development cycle.


Search And Replace

Search String :
[✓ Hex] BE56 C9DA 23EE
Hex mask: FFFF FFFF FFFF
ASCII sometext

Replace String :
[✓ Hex] BE56 C9DA 23EE
ASCII sometext

Address :
Hex : 00BAFC38

Find and replace window prototype.

2.1.3 External Interface Implementation

There are three choices regarding the implementation of the GUI.

- CodeWarrior PowerPlant, a framework written in C++.
- Cocoa, a set of frameworks written in Objective-C.
- Java.

CodeWarrior PowerPlant
- Commercial product.
- Slightly complex, according to my previous experiences.
- Platform dependent.

Cocoa
- Free, including a complete development environment, see [4].
- Having worked through a tutorial, the impression is that it is easy to work with; major parts of the work of creating a GUI is done using a graphical interface building program.
- Platform dependent.
- Typical behavior of a Macintosh application's GUI is implemented in the framework.
- Tools give aid in constructing a GUI adhering to Aqua interface guidelines, as described in [3].

Java
- Free.
- Previous extensive experience of working with Java and its ease of usage.
- Platform independent.

Java might seem like the obvious choice given its platform independence and ease of usage, but I am actually inclined to write the GUI using Cocoa due to the greater possibility of adhering to interface guidelines, application GUI default behavior without coding and the desire to learn something new.

During the course of the development, using Java in connection with Objective-C and C++ proved difficult, so Cocoa more or less became the only alternative available to me. The availability of GNUStep, an open source, multi platform, object oriented application framework very similar to Cocoa, also became known to me, which I assume, without further investigation, will make porting of the debugger GUI easier.

Section 3 – Analysis

Having established the basics of the debugger, it is clear that there are a number , to me, non-trivial issues that need to be investigated.

An example is the memory protection of UNIX.
It is designed to prevent a program from accessing parts of the memory owned by other programs. Memory protection does however also prevent the debugger from accessing the target program's memory resident code and data.

3.1 Areas To Investigate

An attempt have been made to divide the areas requiring investigation in categories, however some overlapping does occur. The investigation has taken place under the operating system OS X and all solutions apply only to this environment.
However, there are solutions relying, for instance, on the Mach-O kernel API, see [5], which are likely to be more easily adaptable on other platforms supporting the Mach-O kernel API.
The areas that are to be investigated are:

Target Program Control

- Launching a target program under the control of the debugger.
- Attaching the debugger to a program currently executing.
- Suspending and resuming the target program from the debugger.
- Handling of signals and exceptions from the target program.
- Single stepping the target program.
- Breakpoints.
- Watchpoints.
- Threads

Target Program Access Prior To Execution

- Reading target program object file
- Disassembling code
- Editing of hexadecimal and ASCII data.

Target Program Access During Execution

- Accessing the CPU registers of the target program.
- Accessing target program memory.

Other Areas

- Interpreting conditions and commands in breakpoints.
- Interpreting machine instructions.

Facade Design
Finally, design choices have to be made in order to strive towards a platform independent facade for the parts of the software solving the above questions.

3.2 Target Program Control

3.2.1 Launching A Program

As described in section 4.9 in [6], launching a target program to be debugged is accomplished by first having the debugger create a child process. The child process then calls the UNIX function ptrace () once to declare that it expects to be traced by its parent. This also enables the target program launch without actually executing any instructions. After having called ptrace (), the child process launches the target program using the common UNIX function execv (). The target program is now ready to be debugged.

No alternative was found to ptrace () in the case of controlling the launching of a program to be debugged.

3.2.2 Attaching To A Program Currently Executing.

In order to be able to present a list of programs (i.e. processes) executing on the computer, a, by Apple Computer, undocumented Cocoa class named NSProcInfo have been used. Partial documentation of this class can be found at [7].

The process can then be attached to using either ptrace () [8], or by suspending the process.

Important note !
As mentioned in [8], if the user of the debugger does not have root privileges, it will only be possible to attach to programs with the same UID as the debugger, i.e. programs launched by the user of the debugger. Attaching to zombie processes, i.e. processes that do not exist but as an entry in the process table, is not possible either.

Since I do not intend to rely on ptrace () calls when interacting with the target program after it has been attached to, mainly due to lack of versatility compared to the Mach-O API but also due to lack of efficiency as mentioned in section 4.9 in [6] (there is no /proc filesystem in OS X), I decided not to use it when attaching to the target program.
The debugger will rely completely on the Mach-O API as far as controlling the target process and its threads.
Portability benefits from using the Mach-O API since the Mach-O kernel is available on different hardware platforms.
Having attached to a running program (i.e. suspending its process), it is possible to suspend each of its threads, in order to prevent the program from executing when its process is later resumed and to control the threads individually.

3.2.3 Suspending And Resuming The Target Program

In order to control the target programs execution, ptrace () can be used in conjunction with sending signals to the target program [8]. Due to not supporting control over more than a single thread within a process, ptrace () is useless in the debugger.
Multithreaded programs are very common today and the debugger should be able to control such programs.
Alternatively, Mach-O API calls [9] may be used to control the execution of a process. These calls also provide control over individual threads within a process.
Using the Mach-O API calls is an obvious choice.

3.2.4 Target Program Signals And Exceptions

Process and thread termination notification

Being able to determine when the target program or one of its threads has terminated without polling, as described in [10], seems preferable, since experience tells me it is easier to write programs to receive notifications rather than constantly have to poll for data.

In the Mach-O kernel, which is part of the kernel in OS X, tasks and threads are able to send different kinds of messages[9, 11]. Among these there is a message sent when the task or thread has died. To be able to receive these messages, a Mach-O port needs to be created. A Mach-O port is a channel of communication similar to a pipe in UNIX. Registering on a task, or thread, using this port will enable receiving a notification when the task, or thread, dies.

I choose to have one common listener handling messages from both the process and its thread due to being the simplest solution and due to not finding any advantages of having multiple handlers. The listener need to be started in a thread of its own, since the Mach-O API call that receives the messages, mach_msg (), blocks if there are no messages to receive. However, mach_msg () cannot be allowed to block indefinitely, since the listener needs to be uninstalled when the debugger resigns control over the target program. Setting a time-out time for mach_msg (), it is made to regularly time-out, which enables regular checking of whether an attempt to uninstall the listener has been made.

In order to be able to identify the thread or process from which the notification originated, one also needs to maintain a connection between the notification ports, supplied when notifications received, and the object representing the process or thread. Message(s) may then be sent to the appropriate object notifying it of any changes in the state of the process or thread they represent.

Exception handling
Exception handling is done in a manner similar to process and thread termination notification, see [12] for a discussion and sample code. In fact, by creating a Mach-O port set, it is possible to wait for messages from multiple sources using just one listener, meaning a single handler can handle process and thread termination notification as well as exceptions from the process and its threads.

There are a number of exceptions the debugger chooses not handle, such as memory page fault exceptions etc. In order for these exceptions to be handled correctly, one needs to find the previous exception handler port and save it, to be able to forward these exceptions to the appropriate handler.

Upon receiving an exception, it is also necessary to determine if the task causing the exception really is the task for which exceptions are supposed to be handled.
The target program may have spawned a child-process which have inherited the exception port settings, thus causing the exception handler to receive exceptions from a task not of interest. This issue is solved in the debugger by forwarding any exceptions from other tasks to the original, saved, exception port(s).

System call notification
Older UNIX/Mach-O kernel versions relied on traps and special kinds of exceptions generated by system calls in order to invoke system functions.
For code residing in user space (i.e. not in the kernel), OS X relies on dynamic binding using function stubs in the calling program that resolves the address of the appropriate library routine at runtime. From these libraries, services residing in the kernel are invoked by sending a Mach-O message to a port a kernel server is listening on [9] but also using the CPU instruction sc (system call).

Jonathan Rentzsch is discussing a technique for "dynamically overriding" system calls on his web page [19], which is a good general technique.

Using the existing exception-handling of the debugger, breakpoints could be used to provide system call notification. Such breakpoints would be invisible to the user and placed at the entry point of the code sending the message to a server port, or the stub calling the library containing the system call code. The debugger will, when the breakpoint is hit, recognize this as a non-user breakpoint and handle it accordingly. Such technique is simpler than the above mentioned one for overriding system calls and uses existing technology, which makes me prefer it.

Regretfully, I have not had time to further investigate where to place such breakpoints so this technique is not used in the first incarnation of the debugger.

Signals
By using the above described measures to have the debugger notified about process and thread termination and exceptions, the debugger will be provided with information concerning the execution of the target program. Handling signals seems unnecessary, considering they in most cases are manifestations of exceptions on a higher level [6].

3.2.5 Single Stepping

The target program can be single-stepped using ptrace () but, as before, ptrace () does not take multiple threads into account.

Single-stepping a thread under OS X is accomplished by getting the thread state using the Mach-O API call thread_get_state () and setting bit 21 in the machine state register (bit 22 for branch-trace), see [14, 15, 16, 9, 12]. The thread state is then set using the Mach-O API call thread_set_state (). As the thread is resumed, the processor will take a trace exception after the execution of every machine instruction. With branch-tracing, the exception will be taken after the execution of branch instructions only. In the case of conditional branches, regardless of whether the branch is taken or not.
Please also refer to section 3.4.1 concerning steps required to change a thread's state with a deterministic result.

Having caught a trace-exception, the thread that caused the exception shall be suspended.

Step over is similar to step in except for when stepping over subroutine calls.
Having stepped into a subroutine, the program counter will be located at the first instruction of the subroutine. Stepping over will cause the subroutine to be executed at normal speed and the program will stop at the instruction after the subroutine call.
This is accomplished by setting a temporary breakpoint at the instruction after the subroutine call and resuming the execution of the thread of interest in the target program. The breakpoint is deleted as it is hit by the thread stepping over the subroutine call.

Control of whether active watchpoints are hit or not will be done when single-stepping the target program, see section 3.2.7 and section 3.6.2.

3.2.6 Breakpoints

Is there hardware support for breakpoints ?
In the MPC7410 and MPC750 processors, there is hardware support for a single breakpoint using the Instruction Address Breakpoint register, see [14, 15, 16].
Support for hardware breakpoint is optional for the PowerPC processor family.
Having just a single breakpoint does not suffice, so an alternative solution being able to supply multiple breakpoints will be preferred.

How to implement breakpoints without hardware support ?
Breakpoints without hardware support are implemented by writing a special "breakpoint instruction" to the address in the target program at which the breakpoint is to be located.
From the kernel's exception handler source-code [17] I was able to determine the breakpoint instruction to be 0x7FE00008.
When a thread tries to execute the breakpoint instruction, an exception will be generated.
In order to resume execution of a thread that has encountered a breakpoint, the following procedure, as proposed in [18], is used:

- Suspend all other threads.
- Overwrite the breakpoint instruction with the original instruction.
- Put the thread into single-stepping mode.
- Let the thread execute one step.
- Write back the breakpoint instruction.
- Cancel single-stepping mode for the thread.
- Resume all threads.

It is important to suspend all other threads in the target program when resuming execution of the thread that hit the breakpoint. This since the debugger actually removes the breakpoint during a period of time and another thread in the target program could, during that time, execute the original instruction and thus fail to hit the breakpoint.
It is also important to remember to save the original instruction that were overwritten by the breakpoint instruction, since it is needed for several additional purposes:

- When displaying memory or disassembling code, it is preferable not to let the user see the breakpoint instruction.
- When removing the breakpoint, the debugger needs to restore the original instruction.

Even though the debugger will not allow the user to set multiple breakpoints at an address, the debugger have to be able to handle multiple breakpoints being present at one address since, as described in section 3.2.5, the debugger sets breakpoints that will be invisible to the user. Not only is it the case that the user might set a breakpoint at an address where an invisible breakpoint already is present, but the debugger might also need to set multiple invisible breakpoints at the same address when more than one thread is executing in the same code.

3.2.7 Watchpoints

Is there hardware support for watchpoints ?
Again, in the MPC7410 and MPC750 processors, see [14, 15, 16], there is hardware support for a single watchpoint monitoring a single byte using the Data Address Breakpoint register. Support for hardware watchpoint is optional for the PowerPC processor family.
A single hardware-supported watchpoint is very interesting, since it would not slow down the target program at all, compare with watchpoints without hardware support. However, manipulating the DABR (data access breakpoint register) have to be done when the CPU is in supervisor mode, i.e. from the kernel.
As reference, I have examined the features of a common Intel CPU and found that it does supply similar features, see [19].

How to implement watchpoints without hardware support ?
Implementing watchpoints without hardware support would involve single-stepping the target program and interpreting every machine instruction in order to determine if access of memory being watched will occur or not. Such interpretion will result in a substantial overhead for every machine instruction in the target program. Additionally, the stepwise execution will cause an exception to be thrown and handled after every machine instruction being executed. As a result, the target program will execute very slowly with one or more watchpoints active.

Implementing silent breakpoints which, when hit could cause the debugger to toggle the watchpoint monitoring feature on or off, could limit the impact of the target program slowdown, provided that the user of the debugger is able to limit the scope in which watchpoint monitoring is needed.
An alternative to single-stepping the target program is to branch-step it, a feature implemented in the PowerPC processors.
Branch-stepping is similar to single-stepping, but instead of generating an exception after each instruction, the CPU will generate an exception after a branch instruction has been executed (whether the branch was taken or not). Again comparing with the Intel CPU[19], I find that this feature seems to be specific to PowerPC processors and using it wouldn't promote platform independence.

To avoid generating an exception after every instruction, one could partially emulate the CPU. This is left as an idea for future enhancements of the debugger.

The limited time available forces me to choose the simplest option:
Single-stepping the thread(s), interpreting each machine instruction.

3.2.8 Threads

The following considerations has been made concerning the monitoring and control of multi-threaded programs:

Give the user control over thread suspension and resumption.
The user can suspend individual threads in the target program. The target program is single-stepped by single-stepping all threads the user has chosen to be single-stepped by setting or clearing a flag related to each thread, one at a time. In a similar manner, with one or more watchpoints active, only the threads flagged by the user will execute.

Provide thread-specific breakpoints.
The ability for the user to set thread-specific breakpoints have been postponed to a later development iteration, but breakpoints set by the debugger when stepping over subroutine calls are, as above, thread-specific.

The target program may deadlock if the user suspends a thread A that need to complete some actions before another thread B is allowed to continue. It is left to the user to recognize and resolve such situations.

The debugger should be aware of the creation and destruction of threads within the target program. Notification of the death of a thread and system call notification has been discussed in section 3.2.4 above. Due to the limited time available, I have not had time to research this further, but had to opt for a simpler solution:
When an exception that is not a trace exception occurs in the target program, the debugger will check for new and terminated threads in the target program. Further details in section 3.2.9 below.

3.2.9 Summing It Up

Below follows a list of situations in which the debugger will have to exercise control over the target program. For each situation, comments and pseudo code for handling the situation is listed. In the pseudo code below, if not explicitly stated otherwise, it is assumed that threads unknown to the debugger will never cause exceptions.

On other exceptions

Comments:
At all times, it is possible that some other kind of exception occurs in the target program, such as memory access exceptions, divide-by-zero exceptions etc.

Pseudo code:
```
if (the target program is running)
{
        Suspend target program process.
        Check for new and terminated threads.
        Suspend all threads in the target program.
        Update thread states for all threads in target program.
        Resume target program process.
        Clear target stepping flag.
        Clear target resuming flag.
}
Exit exception handler.
```

On breakpoint exceptions

Comments:
A breakpoint exception will be taken due to one of the following reasons:
1. The debugger was about to step over a subroutine call.
A system breakpoint was placed at the instruction immediately following the subroutine call and the thread about to step over the subroutine call was resumed.
The thread tried to execute the system breakpoint.
System breakpoints not associated to the thread causing the interrupt will be ignored.
2. The debugger was about to step a branch to an address outside the target program code.
A system breakpoint was placed at the address pointed to by the LR (link register) and the thread about to step the branch was resumed.
The thread tried to execute the system breakpoint.
3. A thread tried to execute a breakpoint set by the user.
4. There is a breakpoint in the code that is not known by the debugger.
The user might have written it "by hand".

It is assumed that suspending a process suspends all threads in it.
This is the case with Mach-O processes, see [9].

Pseudo code:
```
if (the target program is running)
{
        Suspend target program process.
        Check for new and terminated threads.

        // Non-target program thread.
        if (thread causing exception not among target program threads)
        {
                Forward exception to original exception-handlers.
                Exit exception handler.
        }

        Get all enabled breakpoints with address at which breakpoint exception occurred -> breakset.

        // Non-debugger breakpoint, case 4 above.
        if (breakset is empty)
        {
                Clear target stepping flag.
                Clear target resuming flag.
                Suspend all threads in the target program.
                Update thread states for all threads in target program.
                Resume target program process.
                Exit exception handler.
        }

        // Handle system breakpoints, case 1 & 2 above.
        if (breakset contains no user breakpoints)
        {
                Suspend all threads in the target program.
                Resume target program process.

                // Ignore system breakpoints for other threads.
                if (no breakpoint in breakset associated to thread (sys bpt))
                {
```

```
                if (target stepping flag not set)
                {
                    Set target resuming flag.
                    Make threads stepping list empty.
                }
                Associate breakpoints to thread (atBreakpoints).
                DoStep for thread.
                Exit exception handler.
            }

            // Case 1 and 2 above.
            The association between the thread and the breakpoint is broken (sys bpt).
            Delete breakpoint.

            if (breakset contains more breakpoints)
            {
                if (target stepping flag not set)
                {
                    Set target resuming flag.
                    Make threads stepping list empty.
                }
                Associate the breakpoints to thread (atBreakpoints).
                DoStep for thread.
                Exit exception handler.
            }

            if (target stepping flag)
            {
                // Common block DoTargetStepping
                if (threads stepping list not empty)
                {
                    Remove first thread from list.
                    DoStep for first thread from list.
                    Exit exception handler.
                } else if (active watchpoints)
                {
                    // Got watchpoint(s), continue stepping until user cancels.
                    Set target stepping flag.
                    Create list of threads to execute with watchpoints (threads stepping list).
                    Remove first thread from list.
                    DoStep first thread from list.
                    Exit exception handler.
                }
                Clear target stepping flag.
                // End common block DoTargetStepping.
            }
        } else
        {
            // Case 3 above.
            if (breakpoint associated to thread (sys bpt))
            {
                if (system breakpoint in breakset)
                {
                    The association between the thread and the breakpoint is broken (sys bpt).
                    Delete breakpoint.
                }
            }

            Associate the breakpoints to the thread causing the exception (atBreakpoints).
            Clear target stepping flag.
            Clear target resuming flag.
        }

        // Arrives here if a stepping-round is finished and not executing with
        // watchpoints or in case 3.
        Suspend all threads in the target program.
        Update thread states for all threads in target program.
        Resume target program process.
}
Exit exception handler.
```

On trace exceptions

Comments:
A trace exception will be taken under the following circumstances:

- The debugger has had a thread at a breakpoint step over it.
- The debugger is single-stepping all threads tagged for execution when single-stepping the target program.
- The debugger is executing the target program with threads tagged for
- watchpoint-execution and there are one or more enabled watchpoints.

Two of these circumstances may apply at the same time, for instance when the debugger is starting to single-step the target program after having hit a breakpoint.

Since the debugger will try to prevent stepping into code outside of the target program code, I find it unnecessary to check for new and terminated threads upon receiving a trace exception. It would also further slow down the execution of the target program when watchpoints are active.

Pseudo code:

```
if (the target program is running)
{
    Suspend thread that caused the exception.

    Break any association between the thread and any watchpoint associated to it.

    // Have we just stepped over a breakpoint ?
    if (thread that caused the exception has breakpoint(s)
    associated to it (atBreakpoints))
    {
        Enable breakpoints (atBreakpoints) associated to the
        thread causing the exception.
        Break the association between the thread causing the exception and the
        breakpoint(s) (atBreakpoints).

        if (target resuming flag is set)
        {
            if (threads stepping list not empty)
            {
                Remove first thread from list.
                DoStep for first thread from list.
                Exit exception handler.
            } else
            {
                Clear target resuming flag.
                Resume all (selected by user to be) active threads.
                Exit exception handler.
            }
        }
    }

    if (target stepping flag)
    {
        // Common block DoTargetStepping
        if (threads stepping list not empty)
        {
            Remove first thread from list.
            DoStep for first thread from list.
```

```
                Exit exception handler.
            } else if (active watchpoints)
            {
                // Got watchpoint(s), continue stepping until user cancels.
                Set target stepping flag.
                Create list of threads to execute with watchpoints (threads stepping list).
                Remove first thread from list.
                DoStep for first thread from list.
                Exit exception handler.
            }
            Clear target stepping flag.
            // End common block DoTargetStepping.

            // Finished one step with all active threads.
            Suspend all threads in the target program.
            Update thread states for all threads in target program.
            Resume target program process.
            Exit exception handler.
        }
}
Exit exception handler.
```

Stepping a thread (DoStep)

Comments:
The debugger will view all calls outside of the target program code as atomic units, i.e. it will try to prevent single-stepping into such code.
There are two ways such a call can be taken:

- By a subroutine call.
  This is handled by detecting subroutine-calls and setting a system breakpoint at the instruction immediately following the system call.
- By a subroutine call to a stub.
  The stub moves the address of the routine to call into the CTR register and performs a branch to this address using the "bctr" instruction. This is a little more difficult to detect, since the same technique is also used to call virtual methods in C++ classes which resides in the target program code.
  The solution I have chosen is for the debugger to detect branch instructions with a target address outside of the target program code. When such an instruction is detected as it is being about to be single-stepped, a breakpoint is placed at the address pointed to by the LR (Link Register) if the address is a valid address within the target program code. If not, the thread will be single-stepped, stepping into the branch.

IMPORTANT !
The target program may deadlock, as described in section 3.2.8 above, when calling a subroutine outside of the target program code which waits for another thread in the target program to complete some action before exiting the subroutine.

Memory that is to be monitored by active watchpoint may be read from or written to during execution of code outside of the target program. All such memory-access will be ignored.

It is not possible to step over subroutine calls outside of target program code, since the debugger will not allow setting (temporary) breakpoints outside of the target program code.

Pseudo code:

```
if (the target program is running)
{
        if (thread about to be single-stepped has breakpoint(s) associated to it (atBreakpoints))
        {
                Disable the breakpoint(s).
        }

        if (instruction to be executed is a subroutine call and
        (step-over flag is set or call is to an address outside of target program code))
        {
                Delete any previous system breakpoint associated to thread.
                Set system breakpoint at instruction after subroutine call.
                Associate thread to system breakpoint.
                Set thread not stepping.
                Resume thread.
                Exit to await user action.
        }

        if (instruction to be executed is a branch instruction and
        call is to an address outside of target program code)
        {
                if (address in LR is a valid target program code address)
                {
                        Delete any previous system breakpoint associated to thread.
                        Set system breakpoint at instruction at address in LR (Link Register).
                        Associate thread to system breakpoint.
                }
                Set thread not stepping.
                Resume thread.
                Exit to await user action.
        }

        if ((target stepping and active watchpoint) and
        no watchpoint is associated to the thread to be stepped)
        {
                Get address and length of memory read or written by instruction to be executed.
                if (a matching active watchpoint exists)
                {
                        Associate the watchpoint hit with the thread.
                        Notify user.
                        Exit to await user action.
                }
        }

        Break any association between the thread and any watchpoint associated to it.
        Set thread stepping.
        Resume thread.
}

Exit exception handler.
```

Resuming threads in the target program

Comments:
When resuming multiple threads in the target program after a breakpoint has been hit with no active watchpoints, all threads at a breakpoint need to step over the breakpoint before all threads can be resumed, as described in section 3.2.6 above.
If there are active watchpoint(s), any breakpoints having been hit will automatically be disabled before taking a step with the concerned thread.

Pseudo code:

```
Clear target resuming flag.
Clear target stepping flag.
Clear step-over flag.
Clear target paused flag.

// If there are watchpoints, start stepping active threads.
if (active watchpoint and thread to execute with watchpoints, that is not suspended by the user)
{
        Set target stepping flag.
        Clear step-over flag.
        Create list of threads to execute with watchpoint(s) (threads stepping list).
        Remove first thread from list.
        DoStep for first thread from list.
        Exit to await user action.
}

Create list of threads at breakpoints (threads stepping list) that are not
suspended by the user.
if (list of threads at breakpoints not empty)
{
        Set target resuming flag.
        Remove first thread from list.
        DoStep for first thread from list.
        Exit to await user action.
}

// Arrive here only if no threads at breakpoints.
Set all (selected by user to be) active threads to not stepping.
Resume all (selected by user to be) active threads.
```

Start step-in round

Comments:
All steps tagged for execution when single-stepping will take one step, one at a time.
Subroutine calls residing within the target program code address space will
be stepped into. Other subroutines will be executed as atomic units, at normal speed.
A step-round may be interrupted by breakpoints, other exceptions or the user,
in which case the next step-round (if any) will start over.
Watchpoints may be hit when single-stepping the target program.

IMPORTANT !
The target program may deadlock, as described in section 3.2.8 above, when calling a subroutine outside of the target program code which waits for another thread in the target program to complete some action before exiting the subroutine.

Pseudo code:
```
Set target stepping flag.
Clear step-over flag.
Create list of threads to execute when single-stepping (threads stepping list).
Update list of breakpoints at the current program counter of each thread (atBreakpoints).
if (threads stepping list not empty)
{
        Remove first thread from list.
        DoStep for first thread from list.
}
Exit to await user action.
```

Start step-over round

Comments:
All steps tagged for execution when single-stepping will take one step.
Any subroutine calls will execute as atomic units, at normal speed.
A step-round may be interrupted by breakpoints, other exceptions and the user, in which case the next step-round (if any) will start over.
Watchpoints may be hit when single-stepping the target program.

IMPORTANT !
The target program may deadlock, as described in section 3.2.8 above, when calling a subroutine which waits for another thread in the target program to complete some action before exiting the subroutine.

Pseudo code:
```
Set target stepping flag.
Set step-over flag.
Create list of threads to execute when single-stepping (threads stepping list).
Update list of breakpoints at the current program counter of each thread (atBreakpoints).
if (threads stepping list not empty)
{
        Remove first thread from list.
        DoStep for first thread from list.
}
Exit to await user action.
```

Pausing the target program

Pseudo code:

```
if (the target program is running)
{
        Suspend target program process.
        Check for new and terminated threads.
        Suspend all threads in target program.
        Resume target program process.
}

Exit to await user action.
```

3.3 Target Program Access Prior To Execution

3.3.1 Reading Target Program Object File

The target program object file will be read prior to being launched, in order to extract information from it such as:

- Class names.
- Functions names.
- Variable names.
- Name of library references.

In the first development iteration, only function names, including system call stubs, will be extracted from the symbol table in the target program object file.

An important question connected to this area is how to store the extracted information in memory for fast access. Both symbols and comments are a type of labels associated to a specific, unique, address. Such information is preferably stored in a hash-table, or a similar data-structure, providing constant time access to it. Cocoa provides a class called NSMutableDictionary, see [4], which uses hash-tables to provide rapid access to objects in the dictionary.

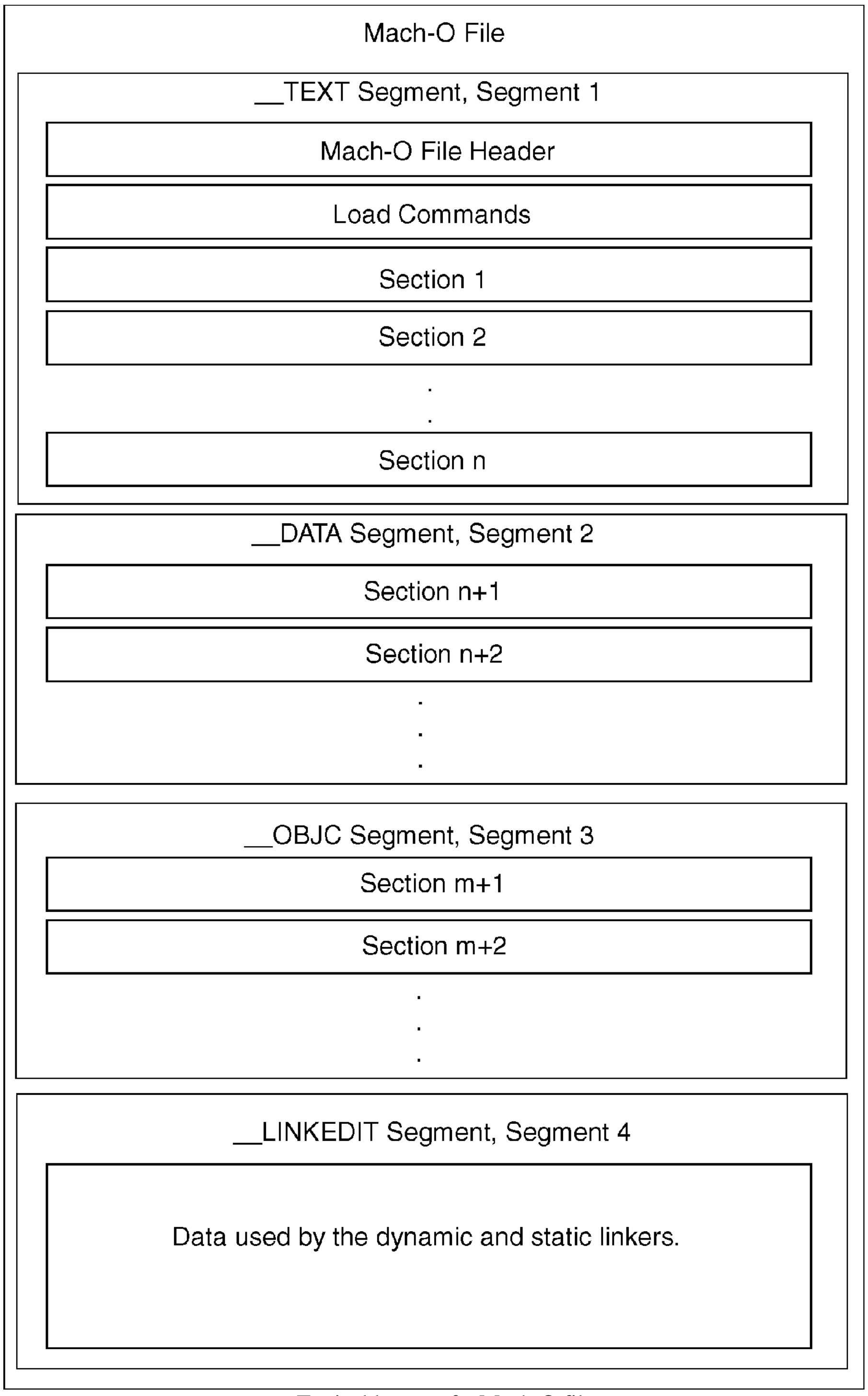
There are two kinds of symbols of interest to the debugger:

- Symbols having an absolute address.
  These are symbols which address reside inside the code of the target program.
  Such symbols are easily extracted from the symbol table found in the __LINKEDIT segment as the address of the symbol is found in the same entry as the reference to the symbol name.

- Indirect symbols.
  These symbols are used to refer to symbols outside of the target program code, for instance, in dynamically linked libraries. Such external functions are called through stubs. These symbols are extracted in the following manner:
  Iterate over the stub sections in the __TEXT segment. Using the index of the current stub, obtain and index into the symbol table from the indirect symbol table. The index of the current stub is an index ranging over all stubs contained in the stub sections in the __TEXT segment. The symbol obtained from the symbol table may then be associated with the address of the current stub.

The Mach-O file format

The information presented in this section is intended as a complement to [1, 20], further clarifying some issues, and providing a very brief overview of the structure of a Mach-O file.


Mach-O File
__TEXT Segment, Segment 1
Mach-O File Header
Load Commands
Section 1
Section 2
Section n
__DATA Segment, Segment 2
Section n+1
Section n+2
__OBJC Segment, Segment 3
Section m+1
Section m+2
__LINKEDIT Segment, Segment 4
Data used by the dynamic and static linkers.

Typical layout of a Mach-O file.

Segments

A segment is a part of the file that is mapped in to the address space of the process that loads it. No data in a Mach-O file is located outside of a segment. Being mapped into the address space, the size of a segment is to be aligned to virtual memory page boundaries. When needed, the segment is padded with zero-bytes at the end. The last segment in the file need not be padded.

A segment may have a larger size in memory than in the file, an example is the page-zero segment that does not occupy any space in the file but occupies the first virtual memory page of the process, once loaded.

Some different types of segments and their names are:

__PAGEZERO	Causes first virtual memory-page to be zero-filled.
__TEXT	Contains executable code and other read-only data.
__DATA	Contains writable data.
__OBJC	Contains runtime data for the Objective-C language.
__LINKEDIT	Contains data used by the static and dynamic linkers.

A segment consists of zero or more sections.

Segment Sections

The segment sections enables further specification of the contents to be mapped into memory space. For instance, specifying the starting address of the section, alignment and relocation information.
When being written to disk, the address of the section have to be aligned with respect to the section alignment, i.e. if the section alignment is 8 and the current address is not a multiple of 8, zero-bytes have to be inserted until reaching an aligned address.
There is also a type of section which only specifies a section of memory to be filled with zero bytes and thus have no data in the file.
I have been unable to find information on the relocation entries specified by the fields reloff and nreloc in the segment data-structure that is part of the segment_command data-structure. In all the files I have encountered, they have always been zeroed.

The __TEXT segment

The __TEXT segment, i.e. the segment that contains the code etc. also contains the Mach-O file header and the load commands. The executable code is contained in a section named __text in this segment.
The __TEXT segment is padded immediately preceding the __text section in order to position the entry-point of the program at a desired location.
It may also require padding at the end of the segment, as above.

The Mach-O Header

First in every Mach-O file, there is a header that identifies the file as a Mach-O file, which architecture the file is intended to be used on and the number of load commands in the file.

Load Commands

Load commands are chunks of the Mach-O file that contains a different set of data depending on the type of the particular load command.

Common to all load commands are:
- An integer indicating the type of load command.
- Total size in bytes of the load command.

A load command may refer to other part(s) of the file that contains additional data, such as symbol table, thread state, executable code etc.

The __LINKEDIT segment

As before, the __LINKEDIT segment contains data used by the dynamic and static linkers. It is divided into the following tables:

Two-Level Hints Table
Contains two-level name space hints table.

Symbol Table
The symbol table contains information created by the compiler for link editing and debugging. The symbol table consists of the following parts:

- Symbol table
  The symbol table is an array in which each entry contains the following information: Index to symbol name, symbol type, number of the segment-section the symbol is found in, additional info about symbol and the value of the symbol.
- String table
  The string table consists of symbol names stored as zero-terminated strings.

Dynamic Linker Symbol Table
Describes sizes and locations of the parts of the symbol table used by the dynamic linker.

The dynamic linking symbol table may contain the following parts:
- Table of contents for a dynamic shared library.
- Module table for a dynamic shared library.
- External reference table data
  Each entry consists of an index into the symbol table (24 bits) and flags to indicate the type of reference (8 bits).
- Indirect symbol table
  A 32-bit index into the symbol table to the symbol referred to by pointer or stub
- External relocation table.
- Local relocation table.

3.3.2 Disassembling Code

For documentation on the instructions, instruction formats and addressing modes of the PowerPC CPUs, please refer to [14, 15, 16, 21, 22].
When writing the disassembler, the emphasis has been put on speed of execution.
This not only due to desiring fast disassembly, but also due to reusing the code in the interpreter that interprets machine instructions – see section 3.5.2 below !
For a detailed technical discussion about disassembling PowerPC assembly language, please refer to the development report !

The disassembler have been extensively tested in the following ways:

- Disassembly of "hand-coded" instructions
  Instructions have been created by setting and clearing the appropriate bits in a machine instruction. The output from the disassembler when disassembling such an instruction have been compared to the expected result. This approach have not been used with machine instructions with primary opcodes 31 and 63, since the amount of such instructions is quite big and generating test code for all of would be too time consuming.
- Comparing the output from another disassembler
  Programs have been disassembled using my own disassembler and a disassembler enclosed with the system development tools. Selecting a group of instructions to examine have been done using the UNIX utility grep. The output have then been compared using the UNIX utility diff.

3.3.3 Editing Hexadecimal and ASCII Data

This is purely a question of programming the graphical user interface.
Also see section 3.5 below for a discussion on how to generalize memory access.
Data is presented in both hexadecimal and ASCII form in the basic implementation of the debugger, however editing of the data has been postponed to a later iteration.

3.4 Target Program Access During Execution

3.4.1 Accessing CPU Registers of the Target Program

The debugger will keep track of the state of each thread in the target program.
A thread-state consists of the contents of a number of CPU registers at a certain point in time.
Under OS X, getting and setting of CPU registers is not supported using ptrace () due to this functionality not being implemented in the kernel, see [17].
The other option setting and getting CPU registers for an individual thread is using the Mach-O API functions thread_get_state () and thread_set_state (), which are supported in OS X, see [9]. These calls facilitate getting and setting of CPU registers, floating point registers, exception state (upon exception passed out of the kernel) and vector state.

Suspending a Mach-O thread may result in it being suspended either in kernel code or in user code. If the thread is suspended in kernel code, the thread also has a kernel-state, besides the user-state, which cannot be accessed from user code, such as the debugger. Upon resumption of a thread having been suspended in kernel code and having had the user state modified, the kernel-state will be restored and execution resumed. Thus any user-state modification will be lost.

In [11] the following procedure is recommended when changing the state of a thread:
- Suspend the thread using thread_suspend ().
- Call thread_abort () (or preferably thread_abort_safely ()).
  System calls are aborted, causing the thread's state to be relocated to a position after the system call, returning a result-code indicating system call having been interrupted. Aborting an exception leaves the thread's state at a point where the exception is to be taken, thus the exception will be retaken upon resuming the thread.
- Now the thread state may be changed using thread_set_state ().
- Resume the thread.

3.4.2 Accessing Target Program Memory

Reading And Writing

Determining whether an address, or an address range, within the target program memory space is a legal address or not, as well as finding the start address and length of a memory region containing a specific address can be accomplished using the Mach-O API function vm_region (), see [9].

After thorough investigation of the kernel (Darwin 6.8) source code [17], I am forced to conclude that reading and writing of target program memory using ptrace () is not implemented, despite the man page suggesting otherwise.

The second alternative is to use the Mach-O API functions vm_read () and vm_write () as described in [9], which facilitates reading and writing of memory of an arbitrary task for which one has the Mach-O port. There is a version of the vm_read () function named vm_read_overwrite () which allows reading to a buffer supplied by the caller, as opposed to allocating memory for the buffer. Using these functions, a section of memory of arbitrary size can be read with one single function call. Empiric testing result in finding no constraints regarding the address and length of the memory to be read, i.e. it is possible to read from any address within the target program's memory space, even odd addresses. It is also possible to read memory of length ranging from 1 byte and up.

Memory Protection

The memory containing the executable code of the target program is write-protected per default. In order to be able to modify this memory the protection-level needs to be changed. This can be accomplished using the Mach-O API function vm_protect (). The write-protection need to be restored immediately after the data has been written since otherwise any attempts of the target program, or some other program, trying to write to the target program memory will not cause an access exception.

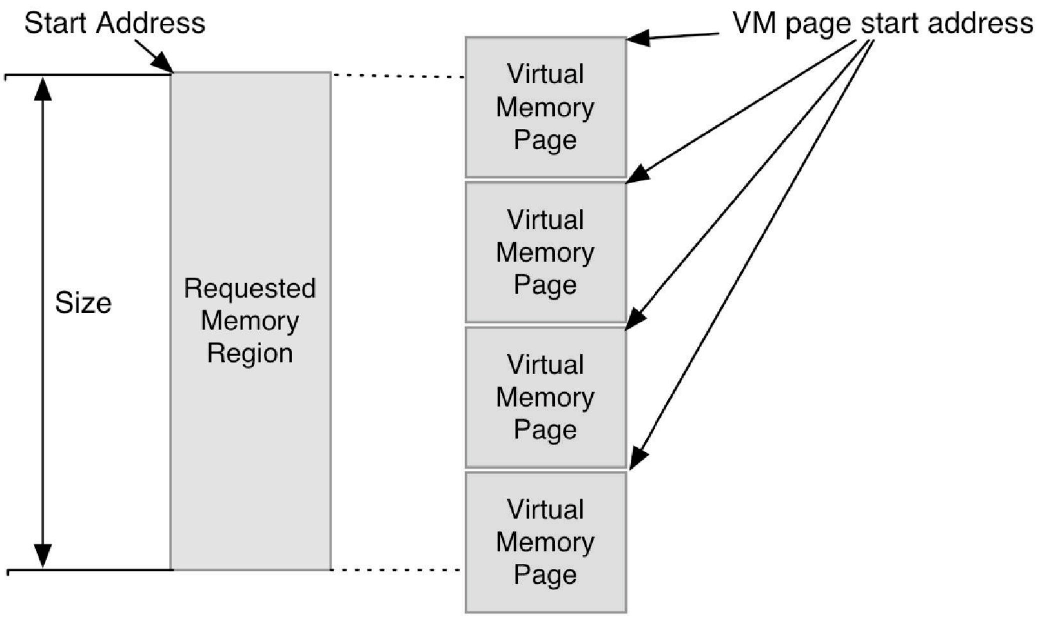
Synchronizing And Caches

After having written modified data to memory, it is necessary to synchronize memory with its backing store image (virtual memory) using the Mach-O API function vm_msync ().

Flushing any caches, such as the CPU instruction and data cache is also very important in order to, for instance, make newly created breakpoints visible to the CPU despite already having pre-fetched the previous instruction at the address of the breakpoint. Data and/or instruction cache(s) is flushed using the Mach-O API function vm_machine_attribute (), also described in [9].

Memory Access Method
Having investigated the Mach-O runtime architecture [1], I have decided to limit memory-access from the debugger to the memory in the __TEXT and __DATA segments of the target program, i.e. the segment containing executable code and other read-only data and the segment containing writable data. The thread windows are an exception, since will also disassemble code outside of the target program executable code.

Prior to accessing a region of memory of the target program, the debugger need to determine whether it is possible to access memory at the first address of the region or not. As above, this is accomplished using the Mach-O API function vm_region ().
If the size of the memory region is within the size of a virtual memory region, a single call to vm_region () suffices, if not, one or more additional calls to vm_region () are required, each with the first address of the requestion region set to the start address plus the size of the virtual memory region.
An object will be created that represents a the memory region in the target program and handles all access to it.



Verifying availability of requested memory region requires , in this case, four subsequent calls to vm_region().

3.5 Memory Access Design Solutions

As above, there are two states in which the target program can be in, concerning memory access:

- Loaded into memory, but not yet launched.
  The debugger is responsible for reading the file and allocating memory for the different parts of the file.
- Launched.
  The operating system is responsible for reading the file, allocating memory for the different parts of the file and preparing it for execution. The debugger will have unloaded the target program file prior to having the operating system launch the target program.

The access to the target program code and data should be independent of these two states.

From the above follows:
Opening a window displaying a section of memory of the target program is not possible until the program has been launched.
Opening a window displaying a part of the target program file is not possible when the program has been launched.
If a window has been opened that display a part of a file, it must be closed when the target program is launched or when the target program is closed.
If a window has been opened that displays a section of memory in the target program, it must be closed if the program is closed.

The solution is as follows:

Closing windows that display memory that is to become invalid can be accomplished using the design pattern Observer [23]. The window observes the object representing the part of memory being displayed, which sends an appropriate message when about to become invalid.

An abstract superclass called ObservableMemRegion is created which specifies the common interface for memory being a part of a file or memory residing in RAM.
It also provides a uniform interface for the window that is to display the memory.
The window is to observe suitable subclass of this class, without having to know anything about the subclass(es).

For memory residing in RAM, a class called MemoryRegion was created.
Requests for instances of MemoryRegion have to be done to a MemoryManager class which keeps track of all MemoryRegions and deallocates them when the target program closes. The MemoryManager can also request all MemoryRegions to send 'updated' messages to their observers for instance when the target program has been relaunched.
The MemoryRegion class is responsible for reading from and writing to memory.
All instances of MemoryRegion also need to be associated to the list that contains the breakpoints, in order to be able to determine the addresses at which breakpoints have been written and make these transparent to the user.

For memory being a part of a file having been read into memory, a class called MemoryBlockPart was created. Instances of this class need to be notified when the object containing the memory-data they are representing is deleted. This can also be accomplished using the design pattern Observer [23]; the instance of MemoryBlockPart observes the object containing the memory-data.

According to the GRASP pattern Expert [2] the responsibility for deciding what subclass of Observable MemRegion to use in a situation will be given to the TargetProgram since it knows about both the instance of the ProgramFile, when the target program has been loaded into memory but not yet launched, the instance of the MemoryManger, when the target program file has been unloaded and the target program launched, and the current state of the target program.

3.6 Other Areas

3.6.1 Interpreting Conditions and Commands in Breakpoints

Parsing the strings describing conditions in breakpoints and commands in silent breakpoints. Postponed due to lack of time.

3.6.2 Interpreting Machine Instructions

For the basic functionality of the debugger, interpretation of machine instructions are needed in the following areas:

- Watchpoints, see section 3.2.7.
- Stepping over subroutine calls, see section 3.2.5.
- Dividing the target program code in functions.
- Finding references to a function.

The information needed to determine whether a watchpoint is hit or not is the address range of the memory accessed by the machine instruction and some value indicating whether the access is a read or write access.

I have decided to implement only the ability to decide the following things about a single machine instruction:

- If the instruction accesses memory and, if so:
  - Which address in memory it accesses.
  - The length of the memory accessed.
  - Whether the memory access is a read or a write access.
- If the instruction is a subroutine call and, if so:
  - The address of the subroutine called.
- If the instruction is a branch instruction and, if so:
  - The address branched to.
- If the instruction is a return-from-subroutine.

Theory of operation
In order for the disassembler and interpreter to operate as fast as possible, it is desirable to completely avoid scanning tables.

The switch-case construct in C++ with, for instance, primary opcodes (the first 6 bits of a PowerPC machine instruction) as selectors, was examined comparing two different kinds of switch-case constructs:
One in which the numbers in the case-clauses were consecutive, i.e. for instance 1, 2, 3, 4, 5 etc.
One in which the numbers in the case-clauses were not consecutive, i.e. for instance 1, 5, 12, 57, 65, etc.

Examining the object code generated by the GCC 3.3 compiler for the switch-case construct, I notice the following:
In the former case, the compiler is able to generate code that calculates an offset to the code that follows the case-clause. This is always done using a fixed number of instructions, regardless of the number supplied to the switch-clause.
In the latter case, the compiler generates a series of compare and conditional branches, one for each case-clause. The result is that the lower down in the list of case-clauses a match is made, the more instructions need to be executed.

Having analyzed the task of disassembling PowerPC opcodes, I have on several occasions encountered the need to make selection based on non-consecutive data.
The solution to this is to have tables translating the non-consecutive data to consecutive data. For instance, say I need to make a selection from the following data: 2, 5, 8, 9
The table would then look like this:
{0, 0, 1, 0, 0, 2, 0, 0, 3, 4}

This isn't very important for the disassembler, since its operation is based on user-interaction and only smaller sections of code will be disassembled at a time. It is more significant for the interpreter, since it will need to execute once for each active thread after every instruction in the target program when there are watchpoints active.

The main drawback with the above approach is, in my opinion, code redundancy.
Many instructions have identical parameter-configurations and could use the same code for extracting the bits in question from the instruction and building the string representing the parameters of the instruction. I have attempted to minimize this by creating inline functions performing bit-extraction and provide tables to look up output strings in.

3.7 Facade Design

3.7.1 Facade Classes

The class representing the target program, as described in the development report, does in many cases act as a facade, hiding platform-specific code.
The classes used to handle the use-cases, see the collaboration diagrams in the development report, also act as facades.

3.7.2 Encapsulation of Standard Classes

In choosing the different classes that the model layer of the debugger will consist of, I have not relied on an existing class library, except perhaps for the Standard Template Library. The Standard Template Library, also known as the GNU C++ Standard Library, classes are totally platform independent with the source code available [24].

Not relying on classes from a specific class library one still can use such classes avoiding depending on them by encapsulating them by classes of the debugger.
For instance, take the list of comments of a target program:
A comment consists of a string and an address in the target program to which the comment is associated. Such a list may simply be implemented as an array.
It will probably be more efficient to implement it as some kind of hash table, using the address as a key. These choices will be invisible to the rest of the debugger, since the comment list class only need to support the operations as described in the development report collaboration diagrams while the class containing the actual data structure is an aggregate to the comment list class.
This not only promote portability, but also makes changing the debugger program's model layer easier.

3.7.3 Model-View-Controller

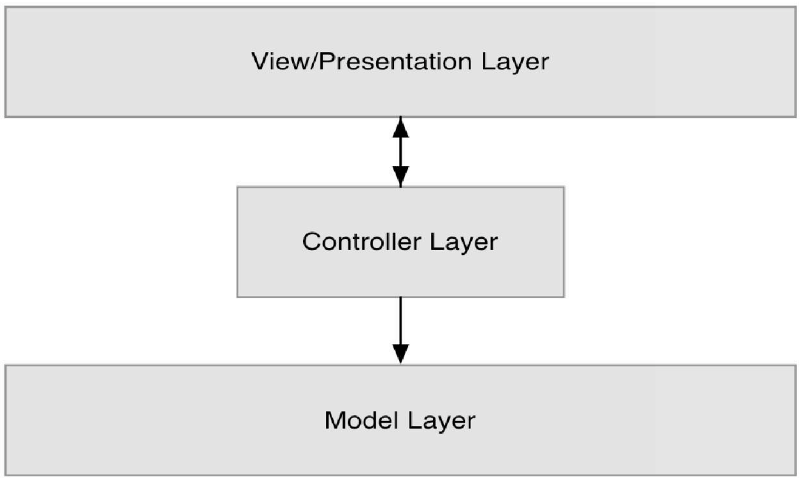
In order to completely isolate the presentation layer from the model layer, controller classes for each window and one controller class handling the application menus have been used.

Section 4 – Implementation

4.1 Implementation

4.1.1 Structure

The code of the debugger has been split in three different layers.



The three layers of the debugger application code.

The view/presentation layer is responsible for presenting information to the user.
The controller layer is responsible for mediating between the two other layers.
The model layer contains the logic and data of the debugger.

The arrows in the figure indicates visibility between layers. For instance, the control layer sends requests to the model layer.

4.1.2 Tools

The entire project have been developed using the Developer's Tools available freely from Apple Computer. Among the tools are:

Project Builder/Xcode	An integrated development environment.
Interface Builder	Tool aiding in constructing graphical user interfaces.
otool	UNIX command displaying object files.
nm	UNIX symbol table display tool.

Tools from other sources:

HexEdit	OpenSource file editor, available at SourceForge[2].

[2]http://www.sourceforge.net

4.1.3 Method

A naming convention have been used for all the variable names:

Prefix	Variable type	Example
s	Static data member	sInstance
m	Data member of class	mItsWindow
in	Input parameter	inIndex
out	Output parameter	outAddress
io	Input/output parameter	ioCounter
the	Local variable	theObject

Classes written in Objective-C have names that start with “OC”, for example OCProcess. Classes written in C++ have names that start with “C”, for instance CTargetProgram. The use case handling classes does not follow this convention.

A uniform layout have been used for header files of classes and also for the files containing the implementation of the classes. Related functions have been grouped together.

Part of the implementation were done during the analysis phase, where small sample programs were created and tested. Basic classes were created, for instance representing files, chunks of a file, threads and processes etc.

Next, the graphical user interface was imlpemented.
The different windows and their layout were created in the InterfaceBuilder tool.
Any custom views, such as the view in the function window containing the disassembled code, were manufactured. One controller class were created for each window and one for the application menus, see section 3.7.3 above.

The different classes of the model layer were then implemented.
The base for the implementation work of the model layer is the collaboration diagrams found in the development report.

For each use case, the classes in its collaboration diagram were created, if they did not already existed. The methods specified by the collaboration diagram were then implemented, if not already present. Finally a use case handler was implemented, connecting the parts needed to perform the use case operation.

4.1.4 Testing

The classes were tested as they were implemented. For instance, having implemented the class representing the preferences of the debugger application, a small piece of sample code were created that set preferences, wrote the preferences to disk, read the data back and displayed the values.

Operations, such as extracting the symbols from a target program were tested and the result output and compared to known results, obtained from manual inspection or existing tools.

The use case operation was tested with any available results presented in textual form, before being connected to the view/presentation layer.

The different entities of the graphical user interface and associated controller classes were tested separately by having the methods in the controller classes provide prefabricated data without invoking the model layer.

Finally, the debugger has been tested on small sample programs written exclusively for this purpose.

Section 5 – Conclusions

5.1 Conclusions

5.1.1 Object Code Level Debugging – To Be Or Not To Be

Is object code level debugging really interesting when many good source level debuggers are available ?
Using a source level debugger is much more efficient and less demanding for the person performing the debugging, since source code is easier to read and understand than object code. This is especially the case with object code that is to execute on a CPU with a reduced instruction set (RISC).

I have however, encountered bugs that I wouldn't have been able to solve without the aid of an object level debugger. An object level debugger is also useful when debugging code for which the source is not available, for instance a library containing support functions. I have also found that understanding the operation of a computer at this level has provided me with, what I consider, essential insights.

My conclusion is that, even though I do prefer source code level debugging, when possible, I do want the possibility to do object code level debugging. When I do need to debug object code, I want tools that are good, reliable and easy to use, in order to be able to focus completely on the debugging.

5.1.2 Experiences

I must admit that I were not entirely clear about the complexity of the project when I started out, some 5 months ago. The project also contains a significant amount of technology that I have had no prior experience with. Nevertheless, it has been a very interesting project which have taught me a lot.

I feel that I should have scheduled time for learning new technologies that were not dealt with during the investigation phase, namely Cocoa programming. I also should have scheduled more time for the design – instead of the planned two weeks, it took me nearly four weeks and the design was only partially completed. Ideally, I would also have preferred to do another iteration over the design, in order to further facilitate portability to other platforms. A positive thing that came out of not having finished the entire design at once is the realization that it is to be preferred to design for a subset of the use cases of the program being developed, implementing these then going back to the design phase with acquired experiences.

I also think cultivating good habits concerning testing and documentation of testing is very important.

Among the most important things learned from this project is the importance of good documentation.

5.2 Future Work

The main areas in which work remains to be done are use case handling and user interface.

Most of the model layer, especially the most central and complex parts involving exception handling and target program memory access, have been implemented. For a more detailed specification of the use cases remaining to be implemented, please refer to the development report.

A substantial amount of work remaining to be done as far as the user interface is concerned. For instance, when editing and searching in memory. Given my lack of previous experience concerning programming user-interfaces in Cocoa, there are also a number of improvements I would like to make. For instance, having proper tool-bars at the top of windows which now have buttons.

Error messages and their presentation to the user need to be reviewed and improved.

Further research in the following areas:
- Thread creation notification
- Extraction of Objective-C runtime information
- Extraction of information from libraries used by the target program
- How to handle system calls when stepwise executing the target program
- Other object file formats, namely PEF (preferred executable format) files.

References

[0] “System Overview”
(PDF document) Apple Computer, 2003.

[1] “Inside Mac OS X: Mach-O Runtime Architecture”
(PDF document) Apple Computer, 2003.

[2] Craig Larman, “Applying UML and patterns.”, Prentice Hall, 1998

[3] Introduction to the Aqua Human Interface Guidelines
http://developer.apple.com/documentation/UserExperience/UserExperience.html
Apple Computer, 040104

[4] Cocoa Documentation
http://developer.apple.com/documentation/Cocoa/Cocoa.html
Apple Computer, 040104

[5] The Mach Project Home Page
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html
Carnegie Mellon University, 040104.

[6] “The design and implementation of the 4.4 BSD operating system.”
McKusick, Bostic, Karels and Quarterman, Addison-Wesley, 1996.

[7] YOSHIMURA hideaki, NSProcInfo class documentation.
http://www.tech-arts.co.jp/macosx/macosx-dev-jp/htdocs/1100/1183.html
040104

[8] ptrace man page, author unknown.

[9] Keith Loepere (editor) “Mach 3 Kernel Interfaces”
Open Software Foundation and Carnegie Mellon University , 1992.

[10] “Technical Note TN2050 – Observing Process Lifetime Without Polling”
http://developer.apple.com/technotes/tn/tn2050.html, Apple Computer, 040104.

[11] Keith Loepere (editor) “Mach 3 Kernel Principles”
Open Software Foundation and Carnegie Mellon University , 1992.

[12] Timothy J Wood “Mach Exception Handlers 101 (Was Re: ptrace, gdb)”
http://www.omnigroup.com/mailman/archive/macosx-dev/2000-June/002030.html
040104.

[13] Jonathan Rentsch, “Dynamically Overriding Mac OS X”
http://rentzsch.com/papers/overridingMacOSX
040104.

[14] “MPC7410/MPC7400 RISC Microprocessor User's Manual”
Motorola, 2002.
Note: This is the manual covering CPU commonly referred to as G4.

[15] “MPC750 RISC Microprocessor Family User's Manual “
Motorola, 2001.
Note: This is the manual covering the CPU commonly referred to as G3.

[16] “Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture” Motorola, 2001.

[17] Darwin sourcecode (Mac OS X kernel).
http://developer.apple.com/darwin/
Apple Computer, 040104.

[18] Daniel Schulz, Frank Mueller, “A ThreadAware Debugger with an Open Interface”
from “International Symposium on Software Testing and Analysis”, 2000.

[19] “IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide” Intel, 2003.

[20] “NeXT step 3.3 Developer's Documentation”
http://www.channelu.com/NeXT/NeXTStep/3.3/nd/index.html
NeXT Software Inc. 1994. Webpage 040104.

[21] Jerry Young, “Simplified Mnemonics for PowerPC Instructions “
Motorola, 2003.

[22] “AltiVec Technology Programming Environments Manual”
Motorola, 2002.

[23] Gamma, Helm, Johnson, Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software” , Addison-Wesley, 1994.

[24] “Standard C++ Library v3”, Free Software Foundation
http://gcc.gnu.org/libstdc++/
040104.

Glossary

Debugger
Program controlling the execution of, and enabling analyze of, another program.

Target Program
The object program that is being debugged and all of its associated data.

Function
A section of code that has one entrypoint and ends with a "return-to-caller".
In this document, the term function has also been used to denominate methods of a class, since very little differs methods from functions when looking at the object code.

Disassembling
The process of converting machine code to a, for humans, more readable form.

Machine Code
A sequence of machine instructions consisting of numbers.

Machine Instruction
A single instruction to the CPU to perform some action.

Breakpoint
An address in the target at which execution of the target is halted and control is to be transfered to the debugger.
May have a condition associated with it, in which case the execution only is to be halted if the condition is true.

System Breakpoint
Breakpoint set by the debugger, not visible to user.
Only intercepts thread to which it is associated.

Watchpoint
An address or address-range in the target program which, when read and/or written, causes the execution of the target to halt and control transfered to the debugger.
May have a condition associated with it, in which case the execution only is to be halted if the condition is true.

Profiling
The gathering of runtime statistics about functions/methods such as:
- time spent in function (function timing)
- the number of times a function was called (function counting)
- present a list of executed or not executed functions (function coverage)

Object Code
See “Machine Code”.

Call Chain
Given a function/method, display the function/method that calls it, then the function/method that calls it etc. etc.

Exception
Interruption of target execution usually caused by error, for instance a division by zero

Signal
Method of making a running process aware of certain software or hardware conditions

Thread
Basic unit of execution which is part of a process.
Each thread has its own execution stack.
Threads within a process share the virtual memory address space.

Process
An instance of a running program, basic unit of resource ownership.

Task
Same as "Process".

Trap
Special machine instructions that cause a certain kind of exception to be taken.
In certain systems, traps are used to access system functions.

AltiVec
A technology used in parallell with the PowerPC architecture that features parallell processing.

Appendix A: User's Manual