Porting Minix C/assembler source code to iPAQ (StrongARM CPU)

Konrad Rzeszutek, Paul Gonin, Venkata Mahadevan, Suchada Phalachaipiromsil

A HOWTO-guide explaining the neccesary steps in porting an Operating System (in this case Minix) to another platform, how to fix/setup a cross-compiler, how to deal with compiling assembler code along with C code, and how to deal with weird errors, problems.

Contents

1 Introduction

This document entails the steps required to build a cross-platform compiler, the usage of it, info about iPAQ StrongARM assembler, how to make a libc library and kernel libraries, how to compile assembler code against C code, and how to deal with weird errors.

The authors of this guide assume prior experience with C and assembler language. The C language knowledge is a must. We also assume you know UNIX and has basic knowledge of GNU suite programs.

2 How this guide is organized.

Steps:

- Getting/installing a cross-compiler suite.
- Getting/using a bootloader on the target platform.

- Compile assembler code.
- Compile C code along with assembler code and link 'em together.
- Drinking lots of coffee.

3 Explanation of terms.

This guide uses a lot of weird names, phrases and such. To faciliate a quicker assimilation of information, this little section is a must read.

Big-endian vs little-endian

Imagine that you are trying to assemble a simple command:

```
mov     r1,r0
```

When the compiler assembles it, it comes to something like this:

```
e1a01000
```

and that's how it gets writen into the output binary file. But if you are using a little-endian, this will be written into the binary file as:

```
0010a0e1
```

As you can see, it's just a difference in bytes ordering.

Side note:, endianess can be defined as the way to see a byte in which case we would be talking about bit ordering endianess but the StrongArm use big endian bit ordering endianess, in our case we are concerned with byte-ordering endianess which deals with the ordering of a sequence of bytes of 4 bytes). The StrongArm is a 32bits processor which uses Little Endian words (4 bytes).

Minix

A simple microkernel operating system. Check out the source code at *Minix* <http://www.minix.org> repository.

text location

Location (in absolute 32bits) for the linker to assume that the kernel will be running from. You see, when the linker links the code, it uses absolute address to reference to string variables and such. So if you don't specify what location in memory the linker should assume the program(kernel) will be loaded, your program(kernel) will display garbage. Here is an example, lets assume we want to load into a register the pointer to a string variable:

```
      puts("boo!");
800c:       e59f0004        ldr     r0, [pc, #4]    ; 8018 <L4>
8010:       eb000019        bl      807c <_puts>
8014:       ea000000        b       801c <L2>
00008018 <L4>:

8018:       0000809c        muleq   r0, r12, r0

0000809c <LC0>:
809c:       216f6f62        cmncs   pc, r2, ror #30 ; "boo!"
```

The *ldr* is a load word into a register, *bl* is a branching with return (something equivalant to i386 *call*). As you can see, the register r0 contains now the contents of location 8018, which is 0000809c. 0000809c is the string "boo!" in hexidecimal. The problem is that when you deploy this program (kernel) on to a system, you might not put the code at 00008000, but somewhere else. So when your program (kernel) runs, it will fetch the data from the wrong location! That is why you need to change the text location to the address where you will put the program (kernel). The argument for the linker is -Ttext=0x[some-value-here]

Cross-toolchain

A toolchain actually consists of a number of components. The main one is the compiler itself gcc, which can be native to the host or a cross-compiler. This is supported by binutils, a set of tools for manipulating binaries. These components are all you need for compiling the kernel, but almost anything else you compile also needs the C-library glibc. As you will realize if you think about it for a moment, compiling the compiler poses a bootstrapping problem, which is the main reason why generating a toolset is not a simple exercise.

4 Building a cross-toolchain

If you have access to a build cross-toolchain, it might be easier to use/install that one. However, you might run into problems with it - the headers might have wrong information, you can't change the linkers text location , its built for big-endian instead of little-endian. But on the other hand, most (if not all) work without problems. Here is a list of websites with cross-toolchains:

- *Arm Linux v4l Cross Tool Chain* <ftp://ftp.handhelds.org/pub/linux/arm/toolchain/>
- *ARM Software Development Toolkit release 2.02u (non-commercial license for education instututions)* <http://www.cse.Buffalo.EDU/~victord/arm/>

But if on the other hand you want to be a geek and have a desire to build a cross-toolchain from scratch, go ahead and read this section.

```
   This is naturally not the only cross-tool chain build
 guide in the world.  If you have trouble comprehending
 this section, you might consider visiting these great sites:
```

- *Building the Toolchain* <http://www.armlinux.org/docs/toolchain/toolchHOWTO/x183.html>
- Keep in mind that most of the documentation regarding building the cross-compiler came from the *HOWTO Build a Cross Toolchain in Brief*.

4.1 Getting the right sources

We are going to build an ARM cross tool chain (cross-platform compiler, cross-platform binutils) for a arm-coff file format (you could pick arm-elf, or arm-aout format as well). This COFF format produces *flat*, or standalone binaries, not tied in to any operating system.

The sources will reside in */ipaq/src*, the */ipaq/build* as the build directory, and */ipaq/local* as the installation prefix.

Neccessary steps:

- Get the source code for the bin utils - *GNU bin utils* <ftp://ftp.gnu.org/gnu/binutils/> or *Chain Tools sources* <http://darnok.dhs.org/~konrad/projects/ipaq/chain-toolchain>
- Get the source code for the GCC compiler - *GNU GCC* <http://gcc.gnu.org> or *Chain Tools sources* <http://darnok.dhs.org/~konrad/projects/ipaq/chain-toolchain>
- Patches for the gcc 2.95.2 compiler - gcc-2.95.2-diff.991022 and gcc-fold-const.patch. Both are available in *Toolchain sources* <ftp://ftp.handhelds.org/pub/linux/arm/toolchain/source/> or *Chain Tools sources* <http://darnok.dhs.org/~konrad/projects/ipaq/chain-toolchain>
- Get the linux kernel compiled for StrongARM. Get the sources from: <ftp://ftp.handhelds.org/pub/linux/compaq/ipaq/development/> or *Chain Tools sources* <http://darnok.dhs.org/~konrad/projects/ipaq/chain-toolchain>

Uncompress all the files in the */ipaq/src* source directory. You should have three directories *binutils-2.9.5.0.22*,*gcc-2.95.2*, and *linux*. Apply the two patches:

```
cd /ipaq/src/gcc-2.95.2
patch -p0 < ../gcc-2.95.2-diff-991022
cd gcc
patch -p0 < ../../gcc-fold-const.patch
```

Create the build directories. These are the directories where the programs will be build. In this document, */ipaq/build* is the build directory.

```
mkdir /ipaq/build
mkdir /ipaq/build/binutils-2.9.5.0.22
mkdir /ipaq/build/gcc-2.95.2
```

4.2 Installing binutils

BinUtils are your friends. They are essentialy the basic tools needed by the cross-compiler to function. They include utilties such as *objcopy, objdump, as, ar, strip, ld* and bunch other.

- Choose the prefix for the new tool chain. This is a fixed directory where the tool chain will forever reside (unless you re-build the tool chain). As I mentioned before, we will use */ipaq/local* in this document.
- Choose the target for the new tool chain. In our case we are using *arm-coff* since its one of the formats that provides flat-format (standalone, not tied in with any operating system).
- In the */ipaq/build/binutils-2.9.5.0.22* build directory (*/ipaq/build/binutils-2.9.5.0.22*) directory:

```
/ipaq/src/binutils-2.9.5.0.22/configure --target=arm-coff --prefix=/ipaq/local --host=i386
make
make install
```

And now you will find a bunch of new applications in */ipaq/local/bin*. Make sure you add this directory to your PATH before procedding with compiling the cross-compiler!.

4.3 Setting up headers for the cross-compiler

For the cross-compiler to compile correctly, its neccesary to have the include files from the Linux. Even though you are going to compile programs using your own include headers, this step is still neccesary. If you don't follow this step, you wont be able to compile the GNU gcc cross-compiler. There only reason why you want to this is that GCC will compile. You can remove the include files later on.

Dirty secret: It uses the include files to make its libgcc.a file. If you are going to link source files using gcc, then you need this file. If you aren't and you are going to use your own libraries, you won't use this file.

- Get the ARM Linux kernel. Get it from *Handhelds.org website* <http://www.handhelds.org> or *Chain Tools sources* <http://darnok.dhs.org/~konrad/projects/ipaq/chain-toolchain>.
- Uncompress and its source directory (*kernel/linux*), type

```
make dep
```

- Copy all the *include/asm-arm* and *include/linux* directories.

  cd /ipaq/local/arm-linux mkdir include cd include cp -dR /ipaq/src/kernel/linux/include/asm-arm ./ cp -dR /ipaq/src/kernel/linux/include/linux ./

4.4 Building GCC.

– Get into the gcc source directory.

```
cd /ipaq/src/gcc-2.95.2/gcc/config/arm
```

and with your favorite editor (hint: *vi*) edit the file *t-linux*. Append *-Dinhibit_libc -D__gthr_posix_h* to the line that says:

```
TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC
```

which should result in:

```
TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc
-D__gthr_posix_h
```

– Get into your build directory and run the *configure* program:

```
cd /ipaq/build/gcc-2.95.2 /ipaq/src/gcc-2.95.2/configure -target=arm-coff
-host=i386-pc-linux-gnu -prefix=/ipaq/local -disable-threads
-with-cpu=strongarm110 -enable-languages=c
```

– Compile the program.

```
make make install
```

If there are problems, such as:

* *Couldn't find stdlib.h and unistd.h.* That means your *-Dinhibit_libc* flag wasn't passed to the Makefiles during auto-configuration. Edit the Makefile in the */ipaq/build/gcc-2.95.2/gcc* directory. Find where it says:

  ```
  GCC_CFLAGS=$(INTERNAL_CFLAGS) $(X_CFLAGS) $(T_CFLAGS) $(CFLAGS) -I./include
  $(TCFLAGS)
  ```

  Add to it: *-Dinhibit_libc*, so that it will look like:

  ```
  GCC_CFLAGS=$(INTERNAL_CFLAGS) $(X_CFLAGS) $(T_CFLAGS) $(CFLAGS) -I./include
  $(TCFLAGS) -Dinhibit_libc
  ```

* *Error: no such 386 instruction:.* You are using the wrong linker. Make sure you have in your PATH variable the directory */ipaq/local/bin* where the arm-coff-ld resides.

– That's it. Just do

```
make install
```

4.5 Thoughts

Instead of *arm-coff* you can use *arm-aout* if you want too.

5 Porting Minix

Now that the process of building a cross-toolchain for the ARM family of microprocessors has been described, we turn our attention to the issues involved in porting the Minix kernel to the Ipaq.

Minix kernel (as any other operating system) has its own set of libraries. Usually all of those are called *kernel libraries* and are only used in the kernel. The user-land libraries are *libc*. Since the user-land libraries use the kernel heavily (*syscalls*), naturally the kernel must implement the syscall facility. That is what the diffrent parts of the kernel do. When a *syscall* gets invoked a function inside the kernel gets called and handles the dispatch. Simple, eh?

Unfortuntly, some operating systems don't keep all the libaries functions completly seperated. Thus you end up using some of the functions from the standard libc libraries such as: *printf, strcpy, etc* Granted if these were implemented in the kernel libraries a certain unneccessary duplication would arise.

And this is where your dilemma is. You can port the kernel libraries and the standard C libraries (libc). Or you can just port the kernel libraries and only port the neccesary functions from the standard C libraries. That latter part is the easiest to do (but later on you have to clean up the code -> its a mess).

We choose to port the kernel libraries and only the neccesary user-land functions from the standard C libraries (libc).

5.1 The problems with Cross-compiling Minix

The whole point of going through the complex process of building a cross-toolchain for ARM processors under Linux was the fact that there is no version of Minix available for the ARM processor family that includes the Amsterdam C Compiler (ACK) that Minix uses. Therefore the following steps had to be undertaken to compile the Minix kernel under the cross-toolchain environment that we set up under Linux i386:

– Makefiles: must be modified to work under Linux Make, including setting explicit paths to find the include files and libraries.

– Compiler: GCC 2.95.2 built as a cross-compiler for ARM targets (arm-coff-gcc)

– Assembler: GNU assembler

– Linker: GNU ld (arm-coff-ld)

– Assembler code: must be translated from Amsterdam assembler format to GNU assembler format. A time-consuming process because the respective syntax of these assemblers is quite different. Minix i386 assembler code must also be converted to ARM assembler code in order to compile and execute on the Ipaq.

– Build tools: binutils with tools such as objcopy, objdump, as, ar, strip, and ld

5.2 Obtaining a Bootloader for the Ipaq In order to boot the Minix kernel (or any other kernel for that matter), a bootloader for the target platform must be procured. The bootloader initializes the "raw" hardware of the computer and allows an operating system (or stand-alone) application to be started by jumping to a particular memory address that contains the binary code of the program. A bootloader for the Ipaq developed by Compaq's Cambridge Research Laboratory is available for download at *Handhelds.org* <http://www.handhelds.org/downloads.html>

5.3 Makefile

Its always easier to write code when you have a *Makefile*. It will save you hours of retyping/code/etc. For more information regarding how to use/write stuff for Makefile, check out *Makefile* <http://www.opussoftware.com/tutorial/TutMakefile.htm> or any website that can find out on the web that teaches about *Makefile*s.

For testing programs, this following Makefile is pretty good:

```
###############################################################################
#
###############################################################################

ROOT = ..
ARCH = /ipaq/local/bin/arm-coff
AR = ${ARCH}-ar
CC = ${ARCH}-gcc
LD = ${ARCH}-ld -N
OC = ${ARCH}-objcopy
OD = ${ARCH}-objdump
CFLAGS = -g  -Wall  -D_MINIX -D_WORD_SIZE=4 -D_EM_WSIZE=4
BASIC = Makefile

all: uart.o hello.o

        $(LD) hello.o uart.o -o A.o
        $(OD) -DSs A.o > debug
        $(OC) --output-format=binary A.o A

###############################################################################



uart.o:
        $(CC) $(CFLAGS) -c uart.S -o uart.o

hello.o:
        $(CC) $(CFLAGS) -c hello.c -o hello.o
```

5.3.1 arm-coff-gcc

The most important flag in this case is the *-c* which instructs gcc compiler to not run the linker. The *-o* is the output flag.

5.3.2 arm-coff-ld

The arm-coff-ld is run with the input object files (*hello.o uart.o*) and with the output *A.o*. Only linking off these two files is done. So if you are using printf or some methods from libc, then the linker won't be able to find the the libc library and complain about it. You will have to add the *-L* flag to provide the libgcc functionality, or implement (port) those functions by yourself.

5.3.3 arm-coff-objdump

This little handy utility disassembles all the binary opcodes from the file along with combining with source code. This program requires the input file to have a header (which we get after linking the files). Look below for the example:

hello.c

```
void gccmain() {

        someweirdfunction("boo!");
}
```

uart.s

```
.text
.global _someweirdfunction

.text

_someweirdfunction:
        mov pc,lr
```

debug

```
A.o:     file format coff-arm-little

Contents of section .text:
 8000 0dc0a0e1 00d82de9 04b04ce2 04009fe5  ......-...L.....
 ... bla bla ..
ioDisassembly of section .text:

00008000 <_gccmain>:

void gccmain() {
    8000:       e1a0c00d        mov     r12, sp
    8004:       e92dd800        stmdb   sp!, {r11, r12, lr, pc}
    8008:       e24cb004        sub     r11, r12, #4    ; 0x4

0000800c <LBB2>:

        someweirdfunction("boo!");
    800c:       e59f0004        ldr     r0, [pc, #4]    ; 8018 <L4>
    8010:       eb000019        bl      807c <_puts>
    8014:       ea000000        b       801c <L2>

00008018 <L4>:
    8018:       0000809c        muleq   r0, r12, r0

0000801c <L2>:
}
    801c:       e91ba800        ldmdb   r11, {r11, sp, pc}

00008020 <_someweirdfunction>:
    8020:       e1a0f00e        mov     pc, lr

.. bla bla ..
```

5.3.4 arm-coff-copy

This program enables one to strip the header, debug information out the executable file and make it completly binary - something equivalant to MS-DOS COM files. This means that the file starts executing at the first byte off the code. Has no stack reserved, and assumes nothing about the operating system. All is left to you, the reader :p

5.4 C/Assembler linkage

How does the compiler link your assembler code against the C libraries (so that you can use the functions from your C code)? Very easy, it just *links* your C code against the assembler code. It basicly inserts branching conditions or inline assembler statments. Look above in the *debug* file.

5.5 Text address

This is problem you are going to run into sonner or later. The problem is that when you compile programs using GCC, the files have a header. This header specifies at what is the starting address, how stack to allocate, etc. For a example off a header, check out *Startup state of Linux/i386 ELF binary* <http://linuxassembly.org/startup.html>. It gives good overview of what the headers does.

But in our case, after we link the program, we strip it off the header so that only binary opcode is left. That is of course what we want, but then our address in the code are screwed up. You see, when the linker links the files it assumes a relocatable address, and the operating system subtracts/adds the program's starting memory location to the memory address in the program. This of course works only when you have somebody taking care of relocation. When the header is striped, the program has the wrong address.

Its quite easy to fix this. You specify manually at what address the program should run from (in memory). Thus, if your bootloader allows you to load the program code at location *0x0010000* then that is what address you should tell the linker to link the programs with. The flag you add is *-Ttext=0x0010000*. Your new entry in the Makefile would look like this:

```
LD = ${ARCH}-ld -Ttext=0x100000
```

5.6 What's the deal with the "_function_name"?

Good question. The GCC cross-compiler (and only the crosscompiler!, not the user-land compiler) assumes that all invocations of functions (regardless its defined in assembler or C code), have to be preceded with the _ character. This problem (or feature) is to allow to diffrentiate between user-land functions and kernel functions. Both of them might have the same name, and this appendix of _ would eliviate a lot of headaches that could have been stumbled upon. When the operating system is ported, the new compiler (which would be nativly compiled) would not assume of such thing.

Actually, the answers lies in the configuration files for the target platform. If you compile the cross-compiler using *arm-elf*, this _ issue will not be present, b/c the ELF files can only be run in a user-land process. While the *arm-coff, arm-aout* don't have this restriction and can be run from memory without headers.

5.7 Since I'm not using headers, how do I know where the program will start from?

Since headers are not being used, a logical questions to ask is: how do I know where the program will start from? In most C books that you may have read, it is assumed that *every* meaningful program starts in a function called main. This is not necessarily true. In reality, it can start anywhere - its the linker which decides who gets called when the program is invoked. Look at the parameter -*e* in your *ld -help*. If you have a program with the function *wazzup()*, just link the program with *ld -e wazzup* and the program will start executing at the function *wazzup()*. This behavior only works if the program has a header.

We won't be using a header. So what we have to do is to stack the programs together.

```
arm-coff-ld entry.o klib.o kernel.o end.o -o kernel.o arm-coff-objcopy
-output-format=binary kernel.o kernel
```

This will produce a kernel binary file which will have the instructions starting in *entry.o*. So do make sure you don't have some data variables in the beginning of the *entry.o* file, otherwise you will be pulling your hair.

6 The porting process.

This section will explain how we attacked the problem of porting Minix v2.0.2 (i386) to an iPAQ (StrongARM CPU). Its quite technical and many hours of sweat have been poured over our stupid mistakes, so don't dare to e-mail us any corrections/ideas. We will silence you :p

6.1 Get to know thyself

Actually, its should be *Get to know the compiler*. Understand from the previos section how the compiler works, how to cross-link assembler and C source code. How to deal with text-location and working around no support from libc.

If you think you got a grasp on that, then you are good to go.

6.2 Some thoughts

Modularize modularize and modularize everything you can. Do little test/suites thing to make sure that your code works.

Do daily backups of your data.

Don't drink too much coffee. You will end up spilling it on the keyboard.

6.3 Providing Input to the Ipaq

As mentioned in an earlier section, input to the Ipaq can be achieved in 2 ways: via its touch screen or serial port. It is plainly obvious that the latter is the only feasible method of providing input to the Ipaq in this case because the touch screen requires some sort of windowing system and touch screen driver to be useable. Therefore, the Ipaq had to be connected to the host PC (running Linux) via a serial cable. The terminal program, Minicom, was used to transfer data (such as files and keystrokes from the host PC) to the Ipaq. Output from the Ipaq was returned to Minicom via the same serial cable. Any terminal program can be used, provided the following settings are maintained: 115,200bps, 8 data bits, no parity bits, and 1 stop bit. Flow control must also be disabled in order for this to work.

For screenshots of our accomplishment visit this *iPAQ Linux success!* <http://www.darnok.org/~konrad/projects/ipaq/images> url.

6.4 Bootloader installation

Instructions for installing the bootloader can be found at the *Handhelds.org website* <http://www.handhelds.org>.

The bootloader was exactly what we wanted. After we looked at the source code, it became evident that we have multiple ways of loading our kernel. We could either load in the memory (*load ram*), which would mean that our starting position would become *0xc0000000*. That is of course the flag that must be submitted to the linker. Otherwise you are screwed.

6.5 Printk and Standard Output

It is obviously essential to have the ability to print to standard output in order to be able to print messages and other information when testing and debugging the kernel. On the PC, output to the screen is achieved by simply writing data to the frame buffer of the video card. Due to the Ipaq's radically different architecture, it is not possible to do this in a manner similar to the PC. Therefore, the simplest means of outputting information from the Ipaq was to write data to its serial port. The information will then show up in the window of the terminal program (Minicom) running on the host PC. The following snippet of ARM assembler code does this:

6.5.1 boo.s

```
 .text

                .global putc            @ void putc (char x)
                .global uart_init       @ void uart_init (void)

#define UTCR0   0x00
#define UTCR1   0x04
#define UTCR2   0x08
#define UTCR3   0x0c
#define UTDR    0x14
#define UTSR0   0x1c
#define UTSR1   0x20

#define BAUDRATE        115200
#define BAUD_DIV        ((230400/BAUDRATE)-1)

                .align

test_putc:
                mov     r0,#'M'
                bl      putc
                mov     pc,lr

putc:
                ldr     r3, UART_BASE
                mov     r1, r0
                mov     r0, #0
                b       1f
                mov     pc,lr

uart_init:
                ldr     r3, UART_BASE
                mov     r1, #0
                str     r1, [r3, #UTCR3]
                mov     r1, #0x08               @ 8N1
                str     r1, [r3, #UTCR0]
                mov     r1, #BAUD_DIV
                str     r1, [r3, #UTCR2]
                mov     r1, r1, lsr #8
                str     r1, [r3, #UTCR1]
                mov     r1, #0x03               @ RXE + TXE
                str     r1, [r3, #UTCR3]
                mov     r1, #0xff               @ flush status reg
                str     r1, [r3, #UTSR0]
                mov     pc, lr

UART_BASE:      .long   0x80050000              @ UART3
```

This code outputs the character 'M' to standard output, which in this case, is the serial port of the Ipaq. At the host PC, the character 'M' will appear in the terminal program's window. The function uart_init is responsible for initializing the serial port on the Ipaq and putc writes a single character to the serial port. Modifying this code to print a sequence of characters i.e. a string is relatively simple. Since the putc function has been declared global, it can be accessed from a C program. Then, within the C program, a loop that repeatedly calls the putc function by passing it pointers to characters can execute in order to print out a character sequence. Alternately, a puts function that does the same thing can be coded in assembler. The latter method is probably more efficient.

However, this code is not entirely sufficient for our purposes. For example, suppose we want to print formatted output such as decimals and hexadecimals. Since we are not using any libc, we cannot use functions such as printf. However, the Minix kernel has a function called printk, which is short for kernel print. Kernel print uses a put character function (which we already have - see above) to print formatted output. This function is written in C, so it can be linked with the assembler code above. Please refer to the Minix v2.0.2 source for a full listing of this function.

6.6 Assembler Libraries (klib and mpx)

here are two main libraries used in the Minix i386 kernel. These are klib386.s and mpx.s. Hence, the first task in porting the Minix kernel to the Ipaq is to convert these libraries to run on the StrongARM CPU. Due to the vast differences in hardware between an Ipaq and an IBM PC, some functions required on the PC side may not be required on the Ipaq and vice-versa.

The klib contains a number of assembly code utility routines required by the kernel. In the klib386.s for the IBM PC, there are many routines that are PC-specific such as functions to copy and move data to the frame buffer of the video card. To get a minimalist Minix kernel running on the Ipaq, the following functions from klib386.s were ported to the StrongARM CPU to form a new library called klibsa1110.s:

```
void phys_copy(phys_bytes source,phys_bytes destination,
               phys_byes bytecount);

void lock();

void unlock();

void exit();
```

The *phys_copy* function simply copies a block of physical memory. The source, destination, and bytecount are represented as unsigned longs. The implementation for the StrongARM CPU loads 4 words at a time and stores them on a memory stack. The *lock()* and *unlock()* functions disable and enable CPU interrupts respectively. The *exit()* function is provided merely as a convenience for library routines that use exit. Since no calls to exit can actually occur within the kernel, a dummy version that simply returns to the caller is implemented.

The mpx library handles process switching and message handling. All transitions to the kernel go through this library. Interrupts (software and hardware) and sending/receiving message calls can cause transitions to the kernel. The most important function in this library is the system call (*s_call*) function. The system call function handles interrupts caused by system calls (software interrupts / SWI's). This was the only function ported to the StrongARM from the mpx.s library.

6.7 Setting up a stack for the C Environment

The kernel must be allocated a stack in main memory for it to operate in. The following snippet of assembler code does this:

6.7.1 head.s

```
 .text
                .align 2
                .global boo

                .text
boo:
                ldr sp,STACK
                bl _start

STACK:          .long 0xc0080000
```

7 Conclusions and Future work

The following goals were achieved in our attempt to port the Minix 2.0.2 OS kernel to the Ipaq:

– A usable cross-compiler suite that generates code executable on the Ipaq was built

– Several assembler language functions from the kernel libraries were ported to run on the StrongARM. Testing confirmed their validity.

– Standard output, including formatted standard output, functions were successfully demonstrated on the Ipaq.

– C and ARM assembly language functions were successfully linked.

– A good deal was learned about The Minix Operating System and operating systems in general.

This project could be continued and completed if given additional time. A lot of the groundwork has already been done. What remains to be done is finishing the remaining assembler functions in the kernel, modifying some of the include files, and attempting a full kernel compile.

Another possibility for the continuation of this project would be switching from Minix to *ucLinux* <http://www.uclinux.org>. Although uCLinux is still a monolithic operating system like Linux, the small size of its kernel and embedded capabilities make it more akin to Minix. The main advantage of using uCLinux is that GCC is used as its C compiler, so there is no need to spend time modifying C code and include files when porting. The uCLinux developer community is also much more active than the Minix one, so obtaining assistance when undertaking a project of this magnitude is easier.

A Corel Netwinder would also be useful in continuing this project. A Netwinder is a StrongARM based computer system intended to serve as a web server. NetBSD and Linux have already been ported to this platform, so using this system as a development environment for the Ipaq is feasible.

8 Links

8.1 Websites

– *GNU Toolchain for ARM targets info* <http://www.armlinux.org/docs/toolchain/>
– *Arm Linux v4l Cross Tool Chain* <ftp://ftp.handhelds.org/pub/linux/arm/toolchain/>
– *ARM Software Development Toolkit release 2.02u* <http://www.cse.buffalo.edu/~victord/arm/>
– *Building the Toolchain* <http://www.armlinux.org/docs/toolchain/toolchHOWTO/x183.html>

– 8.2 Newsgroups

  * comp.sys.arm
  * comp.os.minix

8.3 IRC channels

  * #ipaq on irc.openprojects.net