UNIX ARM Assembler on Android

Brendan Sechter

2016-01-10

Months ago, someone on the FreeBSD forums wanted help getting an assebly language program running on a 64 bit intel machine. I read through the FreeBSD Developers' Handbook x86 Assembly Language Programming section, and sure enough the 32 bit examples did not work. x86 and x86-64 assembler are just plain different. Also, the ABI is completely different.

I managed to find an x86-64 hello world example for FreeBSD. The environment works. Great! Now what? The problem with hello world examples is that there is no input. Without knowing where to go next, a hello world example is not very useful. Between the Developer's Handbook, the System V AMD64 ABI Reference and an x86-64 tutorial (that has since disappeared) I managed to write a command line utility in x86-64 ASM that processes command line arguments.

Then I thought back to the days when I wrote ARM assembler for the Gameboy Advance and Nintendo DS and wanted to write a command line UNIX utility in ARM assembler. My Raspberry Pi was halfway around the world at the time, but my Android phone was handy. No FreeBSD on my phone, but a few people had written hello world examples for android (1) (2) (3). FreeBSD and Linux appear to use the same ARM EABI documented on the ARM site. Also, Android's bionic C libaray has a lot of BSD in it.

The Developer's Handbook notes that "Assembly language programming under UNIX® is highly undocumented". I am writing this post to document writing a command line UNIX application in assembler that conforms to the ARM EABI. Specifically, this application will run on Android. Remember, there has never been an easier time to learn assembler!

My GitHub repository for this project can be found here. Please note that manually linking object files is probably not standard Android NDK usage. The build instructions may break in the future. If that happens, good luck figuring the necessary flags to build the examples. =)

Software Versions

```
$ date
January  8, 2016 at 11:32:13 AM JST
$ uname -a
Darwin siderite.local 15.2.0 Darwin Kernel Version 15.2.0: Fri Nov 13 19:56:56 PST 2015; root:x
$ adb version
Android Debug Bridge version 1.0.32
$ $ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/clang --version
clang version 3.6
Target: armv5te-none-linux-androideabi
Thread model: posix
$ $ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-as --version
GNU assembler (GNU Binutils) 2.24.90
This assembler was configured for a target of `arm-linux-androideabi'.
$ $ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-ld --version
```

```
GNU gold (GNU Binutils 2.24.90) 1.11
$ $ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-gdb --version
GNU gdb (GDB) 7.7
This GDB was configured as "--host=x86_64-apple-darwin --target=arm-linux-android".
$ adb shell "uname -a" # Nexus 5; Cyanogenmod CM-13.0-20160108-NIGHTLY; Android 6.0.1
Linux localhost 3.4.0-cyanogenmod-g9e39333 #1 SMP PREEMPT Wed Jan 6 19:02:34 PST 2016 armv7l
$ adb shell "toybox --version" # for chmod; busybox will work
ac4365b3c292-android
```

Instructions

First, install the Android SDK and Android NDK. Make sure ADB has been installed and you connect to your test device.

```
$ adb devices
List of devices attached
071f7b2ef0e95581    device

$ adb shell "uname"
Linux
```

A rooted device is required to run ASM programs on Android with these instructions. Running ADB as root is handy. You will need chmod, so you will need to install busybox or some other box that provides the same functionality.

```
$ adb root
restarting adbd as root
$ adb shell "chmod +x /root/does_not_exist"
chmod: /root/does_not_exist: No such file or directory
```

Next, install the Android NDK standalone toolchain. The sysroot path probably needs to be defined. Also adding paths for the SDK, NDK and the to be generated standalone NDK is a good idea. I added these lines to my .profile. Adjust pathnames as necessary.

```
export ANDROID_SDK="$HOME/android-sdk"
export ANDROID_NDK="$HOME/android-ndk"
export ANDROID_NDK_STANDALONE_TOOLCHAIN="$HOME/android-ndk-standalone-toolchain"
export SYSROOT="$ANDROID_NDK_STANDALONE_TOOLCHAIN/sysroot"
export PATH="$ANDROID_SDK/tools:$ANDROID_SDK/platform-tools:$ANDROID_NDK_STANDALONE_TOOLCHAIN/b
```

Next, reload .profile and generate the standalone toolchain. The Android NDK standalone toolchain page has instructions for targetting different architectures and Android versions.

```
. .profile
$ANDROID_NDK/build/tools/make-standalone-toolchain.sh \
  --toolchain=arm-linux-androideabi-clang3.6 \
  --install-dir=$ANDROID_NDK_STANDALONE_TOOLCHAIN
```

64 bit ARM devices use aarch64 instead of arm. The following commands may be useful when trying to figure out the architecture of your device.

```
adb shell "getprop ro.product.cpu.abi"
adb shell "getprop ro.product.cpu.abi2"
```

The first program to build and run is a hello world example written in C. In general, it is generally a good idea to have a working C implementation before writing anything in ASM.

```
// main.c
#include <stdio.h>
```

```c
int main(int argc, char **argv)
{
    printf("Hello, World! [C]\n");
    return 0;
}
```

Build it with the following commands.

```
CRT="$SYSROOT/usr/lib/crtbegin_dynamic.o $SYSROOT/usr/lib/crtend_android.o"
$ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/clang --sysroot=$SYSROOT -fPIE -DANDROID -g -c main.c -o
$ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-ld --sysroot=$SYSROOT -pie --dynami
```

This is a funny way of building a C program. What is going on? The end goal is to build and run assembler programs. Assembly files need to run through the assebler and the linker. In order to link ASM object files with C object files all object files need to be manually linked. The entry point to a C program is main, but the program really takes control in the _start function. The CRT files contain this _start function. It has all of the setup code for the "C runtime". This code zeros memory and does other boilerplate tasks. Your compiler usually includes the C runtime automatically so you do not need to think about it. The -lc flag links the standard C library. This is another step your compiler usually handles automatically.

Running the program on the phone should print "Hello, World! [C]". This will remount the system partition of your phone in read write mode. If you do not know what that means stop reading and do not proceed.

```
adb root
adb remount
adb shell "mkdir /system/test"
adb push c-hello-world /system/test
adb shell "chmod +x /system/test/c-hello-world"
adb shell /system/test/c-hello-world
```

With the C version working, it is time to rewrite the program in ARM ASM. Let us start with a header that defines Android system calls.

```
@ system.inc
@ Android Syscall Reference https://code.google.com/p/android-source-browsing/source/browse/lib

.set stdin,  0
.set stdout, 1
.set stderr, 2

.set SYS_nosys, 0
.set SYS_exit,  1
.set SYS_fork,  2
.set SYS_read,  3
.set SYS_write, 4
.set SYS_open,  5
.set SYS_close, 6

.macro sys.syscall id
    mov r7, \id
    swi $0
.endm

.macro sys.exit
```

```
    sys.syscall $SYS_exit
.endm

.macro sys.fork
    sys.syscall $SYS_fork
.endm

.macro sys.read
    sys.syscall $SYS_read
.endm

.macro sys.write
    sys.syscall $SYS_write
.endm

.macro sys.open
    sys.syscall $SYS_open
.endm

.macro sys.close
    sys.syscall $SYS_close
.endm
```

Next, the main assembler file.

```
@ start.s
.include "system.inc"
    .syntax unified
    .set ALIGNMENT,8

.text
    .align ALIGNMENT
    .global _start
_start:
    nop @ for gbd breakpoint

    @ Hello World
    @ sys.write(stdout, message, length)
    mov r0,$stdout
    adr r1,message
    mov r2,$length
    sys.write

    @ sys.exit(0)
    mov r0,$0
    sys.exit

@ Data needs to be in .text for PIE
@.data
message:
    .asciz "Hello, World! [ASM]\n"
length = . - message
    .align ALIGNMENT
```

The next step is building the ASM version.

```
$ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-as --gdwarf2 start.s -o start.o
$ANDROID_NDK_STANDALONE_TOOLCHAIN/bin/arm-linux-androideabi-ld --sysroot=$SYSROOT -pie --dynami
```

The start.s file contains a start function, so the C runtime is not necessary. Functions from the C library can be called from assebler, but this function is using system calls, so -lc is not necessary.

Running it on the phone should print “Hello, World! [ASM]”.

```
adb push asm-hello-world /system/test
adb shell "chmod +x /system/test/asm-hello-world"
adb shell /system/test/asm-hello-world
```

A Makefile for the hello world project looks like this.

```
TOOLCHAIN=$(ANDROID_NDK_STANDALONE_TOOLCHAIN)
SYSROOT=$(TOOLCHAIN)/sysroot
CC=$(TOOLCHAIN)/bin/clang --sysroot=$(SYSROOT)
AS=$(TOOLCHAIN)/bin/arm-linux-androideabi-as
LD=$(TOOLCHAIN)/bin/arm-linux-androideabi-ld --sysroot=$(SYSROOT)
CRT=$(SYSROOT)/usr/lib/crtbegin_dynamic.o $(SYSROOT)/usr/lib/crtend_android.o
INSTALL=/system/test

CFLAGS=-fPIE -DANDROID -g
ASFLAGS=--gdwarf2
LDFLAGS=-pie --dynamic-linker=/system/bin/linker

ASM_TARGET=asm-hello-world
ASM_PARAM=
ASM_DEPS=system.inc
ASM_OBJ=start.o
ASM_LIBS=

C_TARGET=c-hello-world
C_PARAM=
C_DEPS=
C_OBJ=main.o $(CRT)
C_LIBS=-lc

all: $(ASM_TARGET) $(C_TARGET)

force: clean all

$(C_TARGET): $(C_OBJ)
    $(LD) $(LDFLAGS) $^ -o $@ $(C_LIBS)

$(ASM_TARGET): $(ASM_OBJ)
    $(LD) $(LDFLAGS) $^ -o $@ $(ASM_LIBS)

%.o: %.c $(C_DEPS)
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.s $(ASM_DEPS)
    $(AS) $(ASFLAGS) $< -o $@

install: all
```

```
    adb root
    adb remount
    adb shell "mkdir $(INSTALL)"
    adb push $(ASM_TARGET) $(INSTALL)
    adb push $(C_TARGET) $(INSTALL)
    adb shell "chmod +x $(INSTALL)/$(ASM_TARGET) $(INSTALL)/$(C_TARGET)"

uninstall:
    adb root
    adb remount
    adb shell "mkdir $(INSTALL)"
    adb shell "rm -rf $(INSTALL)/$(ASM_TARGET) $(INSTALL)/$(C_TARGET)"
    adb shell "rmdir $(INSTALL)"

test:
    adb root
    adb shell "$(INSTALL)/$(ASM_TARGET) $(ASM_PARAM) && $(INSTALL)/$(C_TARGET) $(C_PARAM)"

clean:
    rm -rf $(ASM_TARGET) $(C_TARGET) *.o
```

The next program will echo all of the command line arguments. This is the C source code:

```c
// main.c
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

// newer NDK removed SYS macros
#ifndef SYS_exit
  #define SYS_exit __NR_exit
#endif
#ifndef SYS_write
  #define SYS_write __NR_write
#endif

#define BUFFER_SIZE  2048
#define MESSAGE      "Args: [C]\n"

char buffer[BUFFER_SIZE];

const char message[] = MESSAGE;
const int  length    = sizeof MESSAGE - 1; // sizeof inclues \0

int main(int argc, char** argv)
{
  // write message
  syscall(SYS_write, STDOUT_FILENO, message, length);

  // loop over argv until argvn_ptr is null
  char *argvn_ptr = *(argv++);
  while (NULL != argvn_ptr) {
    char *buffer_ptr = buffer;
    // copy from argvn_ptr to buffer_ptr until \0 is encountered
    while ('\0' != *argvn_ptr) {
```

```
      *(buffer_ptr++) = *(argvn_ptr++);
    }
    // append \n and write buffer
    *buffer_ptr++ = '\n';
    syscall(SYS_write, STDOUT_FILENO, buffer, buffer_ptr - buffer);
    // next arg
    argvn_ptr = *(argv++);
  }
  // done
  syscall(SYS_exit, 0);
}
```

This is probably not what you expected to see. To be fair, the first version looped over argv and used printf. The C version is, however, supposed to be a C representation of the ASM. This version of C main uses the same algorithm as the following ASM. The following start.s uses the same system.inc.

```
@ start.s
.include "system.inc"
        .syntax unified
    .set ALIGNMENT,8
    .set BUFFER_SIZE,2048

.bss
    .comm buffer,BUFFER_SIZE,ALIGNMENT

.text
    .align ALIGNMENT
        .global _start
_start:
    nop @ for gbd breakpoint

    @ Intro Message
    @ sys.write(stdout, message, length)
    mov r0,$stdout
    adr r1,message
    mov r2,$length
    sys.write

    @ Load Buffer via Global Offset Table
    ldr r0,.Lgot    @ got_ptr = &GOT - X
    add r0,r0,pc    @ got_ptr += X
    ldr r4,.Lbuffer @ buffer_offset
.Lpie0: ldr r4,[r4,r0]  @ buffer = *(got_ptr+buffer_offset)

    @ Write Args
    pop {r0,r1} @ pop argc, argvn_ptr = argv[0]
proc_arg:
    teq r1,$0   @ if NULL != argvn_ptr
    beq done
    mov r2,r4   @ buffer_ptr = buffer
copy_char:
    ldrb    r0,[r1],$1  @ c = *argv_ptr++
    teq r0,$0
    beq output
```

```
    strb    r0,[r2],$1  @ *buffer_ptr++ = c
    b   copy_char
output:
    mov r0,$0x0A    @ c = '\n'
    strb    r0,[r2],$1  @ *buffer_ptr++ = '\n'
    mov r0,$stdout
    mov r1,r4       @ buffer
    sub r2,r2,r1    @ length = buffer - buffer_ptr
    sys.write
    pop {r1}    @ argv_ptr = argv[n]
    b   proc_arg
done:
    @ sys.exit(0)
    mov r0,$0
    sys.exit

@ Data needs to be in .text for PIE
@.data
message:
        .asciz "Args: [ASM]\n"
length = . - message
    .align ALIGNMENT

    @ Global Offset Table
.Lgot:
    .long   _GLOBAL_OFFSET_TABLE_-.Lpie0
.Lbuffer:
    .word   buffer(GOT)
    .align ALIGNMENT
```

The Makefile is more or less the same, but it has these changes.

```
ASM_TARGET=asm-arg-echo
ASM_PARAM=1 2
ASM_DEPS=system.inc
ASM_OBJ=start.o
ASM_LIBS=

C_TARGET=c-arg-echo
C_PARAM=1 2
C_DEPS=
C_OBJ=main.o $(CRT)
C_LIBS=-lc
```

Build and test both versions with the following commands.

```
make all
make install
make test
```

The output should be as follows:

```
$ make test
adb shell "/system/test/asm-arg-echo 1 2 && /system/test/c-arg-echo 1 2"
Args: [ASM]
/data/data/test/asm-arg-echo
1
```

```
2
Args: [C]
/data/data/test/c-arg-echo
1
2
```

The programs can be uninstalled as follows.

```
make uninstall
```

The GitHub repository has six projects. The hello world and arg_echo projects are listed above. There are a couple more versions of hello world. The puts_hello_world project links to libc and replaces the system call with puts(). The main_hello_world project goes a step furthur and uses an ASM main function and the CRT instead of a _start function. The interoperate project calls C, ASM and inline ASM from both C and ASM. The arg_sort project uses structs and malloc to sort command line arguments with a binary tree. The GitHub Makefiles have targets for working with GDB. NOTES.txt contains project notes and references.

Todo

- EABI command line arguments (kind of pushed on the stack; _start and main are different) (EABI post?)
- EABI function calls (first few parameters in registers, everything else on the stack) (EABI post?)
- Position independent code / Global offset table / Procedure linkage table (probably another post)
- GDB (probably another post)
- Work more references into post

References:

- Android ASM GitHub Main Project Page
- Android ASM GitHub Project Notes
- Android ASM GitHub hello_world Project
- Android ASM GitHub arg_echo Project
- Android ASM GitHub puts_hello_world Project
- Android ASM GitHub main_hello_world Project
- Android ASM GitHub interoperate Project
- Android ASM GitHub arg_sort Project
- Android SDK
- Android NDK
- Android NDK Standalone Toolchain
- Android Version to API Level
- Android Debug Bridge
- Android Syscall Reference
- Android, getting bionic C library back in sync with upstream
- ARM EABI Documentation
- ARM Predeclared Core Register Names
- ARM, Getting Started: ARM Assembly for Android
- ARM Assembly Part 3 - “Hello world” in ARM assembly
- ARM, ‘Hello World!’ in ARM assembly
- ARM AALP: 3. The Instruction Set
- ARM, Whirlwind Tour
- ARM, Learning Assembly Basics
- ARM position-independent code in Gas

- ARM, Shared Libraries
- GAS, Assembler Directives
- GAS, Working with C structures and GAS
- GDB, How C/C++ Debugging Works on Android
- GDB, Android debugging with remote GDB
- FreeBSD Forums: Assembly - simple Hello World
- FreeBSD Developers' Handbook: x86 Assembly Language Programming section
- FreeBSD x86-64 Hello World
- System V AMD64 ABI Reference
- Wikipedia: Application Binary Interface