dwelch67 / raspberrypi

Code Issues 22 Pull requests 4 Actions Projects Wiki Secu

master raspberrypi / baremetal /

dwelch more typos fixed in baremetal ... on Jan 19, 2016 History

..

README	5 years ago
TOOLCHAIN	5 years ago

README

```
this is a rough draft, if/when I complete this draft I will at some point
go back through and rework it to improve it.
Update: draft 2.  I went through almost all of this and cleaned it up.
Update: draft 3.  Lots of typos and misspellings that I had missed before

THIS IS NOT AN ASSEMBLY LANGUAGE TUTORIAL, IT DOES HAVE A LOT OF
ASSEMBLY LANGUAGE IT IT.  IF YOU ARE STUCK FOCUSING ON THE ASSEMBLY
LANGUAGE YOU ARE MISSING OUT, THE FOCUS IS CONTROLLING THE TOOLS SO
THAT THINGS ARE PLACED WHERE WE WANT THEM TO BE PLACED SO THE PROCESSOR
BOOTS RIGHT AND LAUNCHES OUR C PROGRAM, AND SO OUR C FUNCTIONS CAN
CALL OTHER C FUNCTIONS.  ASSEMBLY LANGUAGE KNOWLEDGE IS NOT REQUIRED
FOR THIS TUTORIAL.  ASSEMBLY LANGUAGE KNOWLEDGE IS NOT REQUIRED FOR
THIS TUTORIAL.  ASSEMBLY LANGUAGE KNOWLEDGE IS NOT REQUIRED FOR THIS
TUTORIAL.

See the top level README for information on where to find the
schematic and programmers reference manual for the ARM processor
on the Raspberry Pi.  Also find information on how to load and run
these programs.

This was originally written for the ARM11 based Raspberry Pi since
then a Cortex-A7 based (Raspberry Pi 2) has come out.  When you get
to this point the ARM11 based uses a file named kernel.img the
Cortex-A7 uses one named kernel7.img.  I will use kernel.img in the
text, but if you are on a Raspberry Pi 2 use kernel7.img instead.

The purpose of this tutorial is to give you a foundation for bare
metal programming.  The actual touching of registers and making the
chip do things is not addressed here, that is the purpose of the
individual blinker and uart examples.  This tutorial is about mastering
```

the toolchain to understand the foundation of those programs and also to allow you to create your own and hopefully avoid common traps.

First and foremost, what is bare metal programming? You are going to get different answers to that question from people who say they are bare metal programmers. I would say most of them are right despite the difference of opinion on specific details.

To try to generalize my opinion of this I would start by saying that bare metal programming means you are talking to the hardware directly, bypassing an operating system, or certainly if you have no real/formal operating system running. Processors/computers do not require operating systems to run. Operating systems are just programs anyway themselves perhaps being considered bare metal programming.

To begin bare metal programming you start by understanding how the processor boots, how and where it loads and executes its first instruciton, and then making programs that fit that model, placing the first instruction of your program such that the processor executes it when it boots.

The second generalization I will make is that with bare metal programming you are often programming registers and memory for peripherals directly. For example printf() is not bare metal, there are way too many layers of stuff often landing in system calls which are often tied to an operating system. That doesnt mean you cant rig up a printf that works in a bare metal environment, but it does contradict the concept of bare metal. This of course is a gray area for the definition. For example if you wanted to read items off of or write things to the sd card, using a filesystem most programmers even if they create all the code from scratch are going to end up with some sort of layered approach, at one end is low level bare metal talking to registers that wiggle things on a bus somewhere on the other end some sort of open file or create file, read file, close file, etc. Being your own creation it doesnt have to conform to any other file function call standard fopen(), fclose(), etc. So what happens when one person writes some bare metal code, no operating system involved, that can open, read, write, close files on the sd card on the Raspberry Pi, then shares that code? Is that bare metal? Tough question.

I have seen some folks argue that you are not bare metal if you are not writing in assembly. I would argue back maybe you are not bare metal if you are not writing machine code. I keep my bare metal definition to no operating system (unless the operating system IS the bare metal program you are writing) and programming peripherals, etc, directly from your program, or through libraries but not through an operating system.

To continue this tutorial you are going to be exposed to my personal preferences which are not a bare metal thing in general but my personal bare metal things. These will be explained as we go. I have been around the block many times, I have been burned by compilers and manuals and other things and am trying to share some of those experiences

at the same time when I had been around the block fewer times I was that person that refused to take someone elses code as is. I always had to rewrite it myself before even trying it. What I have learned since is that unless the other persons programming environment or tools or whatever are not so painful to get up and running, you should make an attempt to use their environment with their code the way they do it. In particular for these kinds of things that you have not learned and dont know how to do but the author appears to know how to do, THEN, start to make that code your own. Eventually if you are like me, completely replacing all of it including the environment. Other than the potential pain of trying to get their environment up and running, this path of just trying it their way then re-inventing the wheel to make it your own, will have greater success sooner and less frustration.

I assume you are running Linux. The things I am doing here for the most part can be done easily in Windows or on a MAC, but I am not going to get into explaining certain things three times or N times to cover all the possible operating system variations. I tend to run a 64 bit Linux, I switched from Ubuntu to Linux Mint when the post gnome 2 disaster happened. Linux Mint has worked to salvage the Linux desktop for everyone else and I am using Mint now. I do have a number of computers or laptops that I develop on and not all run the same distro or version. For the most part the focus will be on using the gnu tools (binutils and gcc) and other than forward slashes vs backslashes in path names there should be nothing operating system specific about this discussion.

So as soon as we say no operating system, we open a big can of worms. That is as big a problem as the fear of programming peripherals directly, perhaps the biggest problem of bare metal programming. Why is it a problem? Well lets think about the classic hello world C program and maybe what you do or dont realize is going on. In some way, shape, or form you have installed a C compiler on your computer, and they tell you how to compile your first hello world program and it works. One or a few includes, the main() function and a single printf() call. Well there is a HUGE amount of stuff behind that program, it is not one trivial line of code. A myriad of C libraries required, math libraries, etc all to support the uber generic printf function and whatever format string you might send to it. That is just scratching the surface the C libraries that are linked in, a number of them have an intimate relationship with the operating system. The C libraries nor printf code itself handles the console directly, it makes calls to the operating system and its myriad of drivers that ultimately illuminate pixels on the screen. When you go bare metal YOU have to do all of this, a hello world printf() program should NOT be your first bare metal program. Generally your first bare metal program is turning an led on and off assuming the hardware folks have provided an led you can turn on and off with software (usually a good idea for them to do that). Later a uart with individual characters then later a string, but a formatted string, perhaps never.

Note this discussion is limited to assembly language and C. This is one of those personal preference things. In my opinion if you want to be a bare metal programmer you need to know C, no exceptions. And at least

some assembly, dont have to be an assembly guru, just enough to get into your C program and perhaps support interrupts or other exceptions. You should work to make your C programming strong though.

Another one of my simplifications in life is I try to avoid C library calls in my bare metal C programs and further I try to avoid compiler specific library calls, we will see what that means in a bit.

A C compiler is just a program that takes an input and produces an output. That program is compiled to run on a particular computer, my computer. That compiler's job is to create other programs that will also run natively on my computer. The Raspberry Pi uses an ARM processor, most computers out there (servers, desktops and laptops) are running some flavor of the x86 instruction set, generally Intel or AMD chips. ARM is a completely separate company from intel and AMD and their processors use a completely different and incompatible in any way instruction set. On a side note Intel and AMD make chips, ARM does not make chips it just sells its processor designs to people who make chips.

It is quite possible to use a compiler on my computer to generate a program that runs on an ARM processor. A general term for a compiler that runs on one computer but produces output (instructions) that are for another computer/instruction set is called a cross compiler.

Just because a compiler is open source does not mean that that compiler can be made to be a cross compiler. Some/many compilers in history are targetted to their native platform and not cross compiler capable. GCC is designed to generate code for many different instruction sets on the backend. And itself can be built as a cross compiler, but the way GCC works for each architecture you want to target you need to compile gcc for that architecture. LLVM/Clang for example is designed from the ground up to be both a traditional compiler and a Just In Time tool, so its output remains mostly target independent until Just In Time. I suspect it is mostly used as a static compiler though. It has a backend that turns the generic into target specific. A big difference from the gnu tools is that the default build of this backend can output for any of the supported targets with the one tool. No need to re-build for each desired target.

Just because a compiler CAN be built as a cross compiler does not mean it is a good compiler, the more generic you get the more you take away from tuning for a particular instruction set. Both GNU tools and LLVM do a pretty good job in general for each target. Understanding that each target is maintained to some extent by individuals and different individuals produce different quality code so either of these toolchains might have a bad apple or two due to the maturity of the target or the individual or team working on it but other targets may be mature.

This tutorial is going to focus primarily on the gnu toolchain, which is one of those that can be used as a cross compiler but is not trivial to make it a cross compiler.

Fairly soon you will need some tools. At first we only need binutils

which is GNU's collection of assembler and linker tools. There are other tools in there, the assembler and linker are the first we care about. This is NOT a tutorial on teaching assembly language, you will see some, but just enough to get a C programming running. That means we will need a C compiler as well fairly soon. Now I say that this is a non-trivial task. Since this is more of a moving target than this README (hopefully), see the file TOOLCHAIN in this directory for info on finding a gnu toolchain for your platform.

As with C libraries, I also try to not use gcc libraries (I will let you figure out what that means). This is one of those personal things not a general bare metal thing, and the benefit here is that I am only relying on the compiler to do the job of compiling, turn C into ASM. Dont try to do more than that. I become less dependent on the specific compiler and the code is more portable.

So you will need a GNU ARM cross compiler toolchain. binutils and gcc at a minimum, more than that is beyond the scope of this tutorial, have fun. If you cant get that toolchain up you may be stuck at this point. Now the one get out of jail free card you have here is that your Raspberry Pi can run Linux, and you can get a native, non-cross-compiler ARM gnu toolchain on your Raspberry Pi when running Linux fairly easy. Simply prepare a Linux sd card for your Raspberry Pi and use it as a normal computer. At the price point of a Raspberry Pi, if you want to do it this way you might want to have a second Raspberry Pi. One as a Linux development machine where you create the programs and the other as the bare metal machine where you try to run those programs. Where you see arm-none-eabi-gcc for example, on an ARM based Linux system just type gcc instead. If you are using the Linux cross compiler you may have something like arm-Linux-gnueabi-gcc. If I have done my work right then any one of these will work. If you are on an x86 computer though the gcc command by itself WILL NOT WORK. Let me say that again WILL NOT WORK (it builds x86 programs not ARM).

Well beyond the scope of this document but you can also run Linux in a virtual machine like qemu, and within that virtual machine like running on a Raspberry Pi, you can then use a native ARM compiler. And there are other ARM based boards as well the BeagleBones and such that can run Linux and have a native gnu toolchain.

For bare metal the first thing we have to learn is how does our processor/computer boot. We have to know this so we can make our program work, we have to build our program so that the first instruction in our program is placed in the computer such that it is the first instruction run by the computer.

The Raspberry Pi is very much NON STANDARD with respect to how the ARM is brought up. ARM processors boot in one of two ways normally. The normal way an ARM boots is the first instruction executed its at address 0x00000000. The Cortex-M processors specifically (the Raspberry Pi does NOT use a Cortex-M) the ADDRESS of the first instruction executed is at address 0x00000004, the processor reads 0x00000004 then uses the value read as an address, and then starts executing there. The Raspberry Pi contains two primary processors one is a GPU, a processor dedicated to

graphics processing. It is a fully capable general purpose processor with floating point and other features that allow it to be used for graphics as well. The GPU and the ARM share the rest of the chip resources for the most part, they share the same RAM, they share the peripherals, etc. The GPU boots first, how exactly, I dont know, it eventually reads and things from the sd card, then it reads the file kernel.img which it loads into ram for us. Then the GPU controls the ARM boot. So where does the GPU place the ARM code? What address? Well that is part of the problem. From our (users) perspective, the firmware available at the time that the Raspberry Pi first hit the streets was placing kernel.img in memory such that the first instruction it executed that we had control over was at address 0x00000000. Understand that the purpose for the Raspberry Pi is to run Linux (for educational purposes) and at least on ARM, the Linux kernel (also known as a kernel image) is typically loaded at ARM address 0x00008000. So those early (to us) kernel.img files had 0x8000 bytes of padding. Later this was changed to a typical kernel.img that instead of being loaded at address 0x00000000 was loaded at 0x00008000.

So the typical setup is the GPU copies the kernel.img contents to address 0x00008000 in the ARM address space, then it places code at address 0x00000000 which does a little bit of prep then branches to the kernel.img code at offset 0x00008000. Since kernel.img is our entry point, it is the ARM boot code that we can control, we have to build our program based on where the bytes in this file are placed and how it is used. The presence of a file named config.txt and its contents can change the way the GPU boots the ARM, including moving where this file is placed and/or what address the ARM boots. All of these things combined can put the contents of the file in memory where you didnt expect and your program may not run properly.

Here is another one of my personal preferences to deal with. I prefer to use the most current GPU firmware files from the Raspberry Pi repository: bootcode.bin and start.elf. I prefer to not use config.txt, not have a file named that on the sd card, and the only other file being kernel.img that I create instead of the one from the Raspberry Pi folks. This means that I prefer to deal with how the kernel.img file is used for the Linux folks. From the time that I received my first Raspberry Pi to the present, the up to date bootcode.bin and start.elf have placed kernel.img at 0x00008000 in ARM address space, and that is my ARM entry point. 0x00008000 is the location for the first ARM instruction that we choose to control.

So now we are ready to approach our first program. We know that our program is a file named kernel.img which is just a binary file that is copied to ARM memory space at address 0x00008000. We have built and/or installed a gnu cross compiler for ARM, at a minimum binutils and gcc.

Now now for another preference of mine. If you think about your C programming experience, although you may have been taught to avoid global variables at all costs you know they exist and you have or should have been taught at least something about them. Even if you have not you have no doubt initialized static local variables:

```
unsigned int apple;
unsigned int orange = 5;
int main ( void )
{
    static unsigned int pear = 7;
    unsigned int peach;
    ...
}
```

With the code above as a C programmer your are taught that apple will have the value zero, orange and pear will have the values indicated in the code when the body of your main program runs. Now you should also know that peach will be undefined, you have to assign it a value before you can safely use it.
-How does all of that happen?
-Is there C code that runs before main() is called that prepares memory so that your program has those memory locations filled with values?

If that were the case and it was C code, and that C code made the same assumptions about variables being pre-initialized, would there be C code that preceeds that code? This feels like a "Which came first, the chicken or the egg" problem. But it is not. The answer is there is some code written in assembly language the is executed before main() is called and that assembly language code prepares these memory locations so that when your C code starts apple, orange and pear have the proper values loaded. This assembly language code is often called the bootstrap code. A very appropriate term for us as that small bit of assembly language code will both be the boot code for the ARM, the first instructions, that we control, that the ARM runs and it is also the code that we are using to prepare memory, etc so that the C programs work as desired.

And this is my preference on this with respect to bare metal. For the code that follows and much of the code in my repos, I DO NOT support the initializing of variables in the way described above. If you were to take one of my examples and add the apple orange and pear variables above you should not expect to get 0, 5, and 7. Further what you do find you should not expect to find every time, simply make no assumptions about the starting contents of variables. This is my preference not a generic bare metal thing. It is a problem that you have to solve for generic bare metal programming and this is how I solved it. When you finish this tutorial go over to the bssdata directory, and read about why I do it the way I do it and what other work you have to do to insure those variables are pre-initialized before main() is called. The short answer is it involves toolchain specific things you have to do, and I prefer to lean toward more portable including portable across toolchains (minimizing effort to port) solutions. So one thing is I try to make my C code so that it does not use "implementation defined" features of the language (that do not port from one compiler to another, inline assembly for example). Second I try to keep the boot code and linker scripts, etc as simple as possible with a little sacrifice on adding some more code. Linker scripts in particular are toolchain specific and the the entry label and perhaps other boostrap items are also

toolchain specific. You will see what all of that means in the bssdata directory.

Also note that I do not use main() as the entry point funciton in my code. The first time I learned all of this stuff the compiler tools I was using at the time would add extra junk to your binary when it saw the word main(). If you used some other name then it would not add that junk, and not bloat the binary. The Raspberry Pi has relatively lots of memory at 128KB+ for the ARM. In the embedded bare metal programming world you very often face 8KB or 16Kb or 32KB etc and you cannot afford the toolchain sucking up chunks of that memory with stuff you are not using. Part of bare metal programming is you being in control of everything, the code, the peripherals, and the binary.

Good, bad, or otherwise the GNU tools dominate, binutils which includes an assembler, linker and library tools and gcc which includes a C compiler and can include other things. One of the pro's is that when you learn the GNU tools for one platform most of that knowledge translates to other platforms (learn embedded ARM with gnu tools and the learning curve for MIPS is much smaller). What are the tools we are going to be using? We should at this point already know that gcc is the C compiler and we can compile our programs into something called an object or your experience may be limited to creating binaries from your C program without seeing any of the intermediate files. There is actually a bit of hidden magic that goes on. When you compile your hello world program on your Linux machine, the first one or few files generated is your C code in different forms they make another file which is your C code plus all of the includes expanded into that file. Eventually the actual C compiler is called and that turns the C code into assembly language in a text file. Yes, assembly language. Then the assembler is called by the compiler and the assembler assembles the assembly language into an object file, which in this case is a flavor of binary file that has most of the instructions in machine code but is not a complete binary because there may be some functions or variables in other objects that wont be resolved until link time. For our hello world printf to output something it needs to link with a C library which makes system calls and may or may not have to link with other stuff. So the linker takes the object that came from our code and links that with these other items and creates a binary that is compatible with the operating system we are running.

The next thing we have to know is there can be a difference between the entry point into our program and the first instruction in the program. If you think about it most programs we use a compiler for run on operating systems. The operating system loads the program from the filesystem into memory and then performs a jump into that memory, it can jump to any address. It may or may not do that but it is at least possible on a system that is already running. But for booting a processor we cannot change the processor to boot anywhere we want and on the Raspberry Pi we cant or at least shouldnt try to change its habit of executing the first instruction in the kernel.img file. So we have to make sure we control the whole linking process to insure that happens.

I think we have enough ammo to stop chatting and start writing some

programs. I hope you dont hate me at this point but this tutorial is not actually going to run any programs on the Raspberry Pi, in order to build a brick wall someone has to show you how to mix the mortar and how to build that wall one layer at a time, the right amount of mortar per layer, how to keep the rows straight and keep the wall from leaning one way or the other. I mentioned at the beginning that bare metal programming is as much about knowing and manipulating the compiler tools as it is about manipulating peripheral registers. Before we can even begin to talk about peripherals we have to have code that actually runs on the hardware. We will touch on perhiperals in the sense that I will borrow from my other programs in this repository that already talk about the peripheral side of bare metal. This directory is about the compiler side of bare metal. Your takeaway here is being able to understand why my bare metal examples work.

The GNU linker is looking for a label named _start to know where the entry point of the program is. It is possible to override or replace this with something on the linker command line, it is easy enough to just use that label, so we will do that.

The bare minimum bootstrap code for this processor would be to set the stack pointer and to branch to our C program. Now I use notmain() as the name of my entry point into C. But you ask: What is a stack pointer? You should have learned about stacks in general in your prior programming training or experience. The stack is nothing more than a chunk of memory. How it differs from memory is not that it is special because it is not, it is how it is accessed. Our apple and orange variables above are global, they are at a fixed place in memory, lets say they end up after compiling and linking these variables end up at addresses 0x1234 and 0x1238 respectively. Any code in any function that wants to access them will after compiling and linking be accessing those addresses. But what about our peach variable above, that is a local variable and you may have been told that that "lives on the stack". Instead of being at a fixed address in memory, the peach variable will, after compiling and linking be at a fixed OFFSET in memory, offset relative to what? Relative to the stack pointer at some point in time in the function. The stack pointer is simply a register that holds a number which is an address in memory. Not special memory just memory on this platform the same memory we use for our program and our variables. When the compiler converts our C code into assembly code one of the things it has to do is manage these local variables and other things. Any C function that has local variables will cause the compiler to create code that moves the stack pointer as a way to allocate memory for that variable. We will cover this topic more as we go, for now understand that the minimum bootstrap code for this platform is to set the stack pointer and then to branch to our top level C function. Here is some code that does that:

```
.globl _start
_start:
    mov sp,#0x00010000
    b notmain
```

Now I told you this is not a lesson in assembly language programming,

but we will be looking at assembly language even if we dont know exactly what all the code means or does. Many may disagree with me but disassembling your program is one of the fastest and easiest ways to debug your bare metal code. I will keep saying this, a big part of bare metal programming is knowing your compiler tools, very often, esp with bootstrap code your bug may not be in the code itself but in the way you used the tools, the command lines or linker scripts that you used to compile and link that code. Get it wrong and no matter how bug free your C code is it will not run and you will have a hard time figuring it out without looking at what the compiler and linker generated. So the above code starts with a directive .globl, .global also works, both do the same thing, declare the label _start as global meaning it is visible to the linker. In C everything (functions and non-local variables) is global unless you put the word static in front of it then it becomes local:

```
static unsigned int apple;
unsigned int orange:
```

The apple variable which becomes a label or an address in assembler would not be global, where orange would be marked as global.

We read above that _start is a special name the linker is looking for. The linker interprets this as our entry point. Since we are not running this program on an operating system for example it doesnt actually matter if _start is our entry point, but for places where it is used it is a good habit to place it at our entry point for sake of habit. And that is what we are doing here.

The mov sp line basically says put the number 0x00010000 in the register named sp, which is an alias for r13. R13 in the ARM is a register that has special use as the stack pointer. Registers in a processor are very much like variables in a C program in how they are used.

And the last line b notmain means branch to notmain. Branch is also known as a jump in other assembly languages and is exactly like a goto in C.

We are going to start using the tools that you installed, this step may be a major research project for you or it might just work. You might only need to set the path to your tools to make this all work ("baremetal >" being the command prompt):

```
baremetal > arm-none-eabi-as --version
arm-none-eabi-as: command not found
baremetal > PATH=/opt/gnuarm/bin/:$PATH
baremetal > arm-none-eabi-as --version
GNU assembler (GNU Binutils) 2.22
Copyright 2011 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `arm-none-eabi'.
```

Your path may be and probably is different than mine. If you dont
get the command not found, then you wont need to mess with the PATH
it is ready to go. Again this may be a research project for you or it
may just work or somewhere in the middle.

The gnu assembler is a program named "as". When we make it a cross
assembler to not confuse it with the as assembler that we need for the
operating system we are running on, we add a prefix to the name. A
common one you will find in this day and age for gnu tools is
arm-none-eabi-. That will be tacked on the front of everything in the
GNU tools that we care about and that is the one I will be using. You
may have arm-linux-gnueabi- or you may have arm-elf- or arm-thumb-elf-
or many other prefixes. Although they can vary in theory, the way I
write my code, they should mostly come close to working.

Lets say I called that small bit of assembly bootstrap.s

```
baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-objdump -D bootstrap.o

bootstrap.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   e3a0d801    mov sp, #65536  ; 0x10000
   4:   eafffffe    b   0 <notmain>
```

So I have assembled the code into an object file. The default object
file format is elf. Then objdump -D disassembles that object file
so that we can see the machine code and other things the assembler
did.

So what do I mean by elf format? Well you may or may not know that
the term binary when you are talking about a program running the
binary loading the binary, compiling to binary. Is a loaded term
sometimes it is all binary bits and bytes that make up your program.
Most of the time, esp when running on an operating system, that file
is a mixture of the bits and bytes of your program that are wrapped by
a file format that contains things like debugging information and
other things.

If the file only contained the machine code and data that makes up the
program it would only need these 8 bytes (this is not a real, functioning
program remember).

```
e3 a0 d8 01
ea ff ff fe
```

How would the disassembler then know from that the names of things like
_start and notmain? The answer is the file is not 8 bytes it is

```
larger

baremetal > ls -al bootstrap.o
-rw-r--r-- 1 root root 664 Sep 23 13:47 bootstrap.o

baremetal > hexdump -C bootstrap.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 28 00 01 00 00 00  00 00 00 00 00 00 00 00  |..(.............|
00000020  94 00 00 00 00 00 00 05  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  09 00 06 00 01 d8 a0 e3  fe ff ff ea 41 15 00 00  |............A...|
00000040  00 61 65 61 62 69 00 01  0b 00 00 00 06 01 08 01  |.aeabi..........|
00000050  2c 01 00 2e 73 79 6d 74  61 62 00 2e 73 74 72 74  |,...symtab..strt|
00000060  61 62 00 2e 73 68 73 74  72 74 61 62 00 2e 72 65  |ab..shstrtab..re|
00000070  6c 2e 74 65 78 74 00 2e  64 61 74 61 00 2e 62 73  |l.text..data..bs|
00000080  73 00 2e 41 52 4d 2e 61  74 74 72 69 62 75 74 65  |s..ARM.attribute|
00000090  73 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |s...............|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 1f 00 00 00  |................|
....

You can see at offset 0x34 in the file we see the 8 bytes of our program.

There are many file formats supported by the GNU tools.  Elf is the
default format for ARM based programs and many others as well.  But we
can convert those into other formats using another of the binutils tools
and we will have to use that tool for the Raspberry Pi.  First off
notice that the .elf file format is binary itself most of the information
is not directly human readable you need to use other programs (like
objdump) to extract information from that file.  Another format that you
will see "binaries" in is the intel hex file format.  This is an ASCII
format file making it easier for us to read and manipulate as programmers
and hack at if so desired...You will still find this format used in various
corners of the embedded world.  Many rom/flash programmers support this
file format, many bootloaders (like my bootloader07) support this
format.

baremetal > arm-none-eabi-objcopy bootstrap.o -O ihex bootstrap.hex
baremetal > cat bootstrap.hex
:0800000001D8A0E3FEFFFFEAB6
:00000001FF

The objcopy command line takes a command line option -O with some
predefined name like binary, ihex, srec, and others. If possible it
determines the file format of the input file (bootstrap.o in this case)
and then converts what it can to the output file format.

baremetal > arm-none-eabi-objcopy bootstrap.o -O binary a.bin
baremetal > arm-none-eabi-objcopy bootstrap.hex -O binary b.bin
arm-none-eabi-objcopy: Unable to recognise the format of the input file `bootstrap.hex'
baremetal > arm-none-eabi-objcopy -I ihex bootstrap.hex -O binary b.bin
baremetal > ls -al *.bin
-rw-r--r-- 1 root root 8 Sep 23 14:04 a.bin
-rw-r--r-- 1 root root 8 Sep 23 14:04 b.bin
baremetal > diff a.bin b.bin
```

```
baremetal > hexdump -C a.bin
00000000  01 d8 a0 e3 fe ff ff ea                           |........|
00000008
```

That little exercise shows how to take just the bytes of our program
and put them in what we would most accurately call a binary file, just
the 8 bytes of our program nothing more nothing less. We will need
to do this for the Raspberry Pi. Notice how objcopy was not able
to recognize the file format for the intel hex file and we had to
specify it using the -I.

To see the file formats supported by objcopy try this:

```
baremetal > arm-none-eabi-objcopy --info
BFD header file version (GNU Binutils) 2.22
elf32-littlearm
 (header little endian, data little endian)
  arm
elf32-bigarm
 (header big endian, data big endian)
  arm
elf32-little
 (header little endian, data little endian)
  arm
elf32-big
 (header big endian, data big endian)
  arm
srec
 (header endianness unknown, data endianness unknown)
  arm
symbolsrec
 (header endianness unknown, data endianness unknown)
  arm
verilog
 (header endianness unknown, data endianness unknown)
  arm
tekhex
 (header endianness unknown, data endianness unknown)
  arm
binary
 (header endianness unknown, data endianness unknown)
  arm
ihex
 (header endianness unknown, data endianness unknown)
  arm
```

We have tried intel hex or ihex and I want to show you another ASCII
based one called srec or s record

```
baremetal > arm-none-eabi-objcopy bootstrap.o -O srec bootstrap.srec
baremetal > cat bootstrap.srec
S0110000626F6F7473747261702E7372656335
S10B000001D8A0E3FEFFFFEAB2
S9030000FC
```

You can use wikipedia to get the definitions for the intel hex and
s-record file formats and very easily write a program that parses those
files and extracts things, maybe write your own disassembler for
educational purposes or write a bootloader or an instruction set
simulator or any place where you need to take a compiler/assembler/linker
generated program and read it for any reason. Let me point out that
the elf specification is as readily available and although there are
libraries out there to parse those files, it is as easy to make an elf
parser as it is to make an ihex or srec parser. If you make it yourself
then you dont rely on some third party library that is going to change
over time causing your code to no longer work or have to change to
conform to some new standard for that library.

So now lets make our first C program, this is not hello world, even
simpler it does nothing, so we think:

```
void notmain ( void )
{
}
```

```
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-objdump -D notmain.o

notmain.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <notmain>:
   0:   e12fff1e    bx  lr
```

So what does bx lr mean? Bx is an ARM instruction that means branch
exchange, and lr is the link register. When you call a function in
your C code your expectation is that the processor will jump somewhere
and execute the code in the function then it will come back and
keep running your program/code after that function call.

```
...
  a = b + 7;
  c = fun(a);
  d = c * 5;
...
```

After calling the function fun() we expect the code to come back and run
d = c * 5. Well the way the ARM does it is the call to a function uses
an instruction called branch link, which saves the address of the code
after the function call in a register called the link register. Then
at some point we encounter one of a couple instructions in ARM that
will allow the program to jump to the address in the link register
returning to where we were executing just after the function call. One
is the branch exchange and the other is a mov pc = lr

```
bx lr
```

```
or

mov pc,lr

Depending on the tools and how you use them you should mostly see the
bx lr in assembly and in the code generated by the compiler if you dont
then there may be a reason which you may or may not be concerned about
at this time.  I will keep saying this, this is not a tutorial on
assembly language, but you may already see that assembly language is
required in order to start up C code, and I argue required in order
to debug bare metal code.  I am only touching on a little bit of
asm readability which is a long way away from teaching how to program in
assembly language.  I have to cover some basics so that we can get to
our C code and also so we can see what the compiler and tools are doing.

So now we have two objects bootstrap.o and notmain.o that we need to
link together.  Way above we talked about having our program start at
address 0x8000, so lets try linking for the first time.

baremetal > arm-none-eabi-ld -Ttext 0x00008000 bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eafffffe    b   8008 <notmain>

00008008 <notmain>:
    8008:   e12fff1e    bx  lr

Cool, our first Raspberry Pi bare metal program.  Problem is we cannot
run this, for a number of reasons.  First off I intentionally used the
wrong instruction in the bootstrap code, second this is an elf file
not a bin file.  How do we fix these things?

So now that I have mentioned the link register and how it is used to get
back from one function after calling it.  If you think about the
compilers job, at one level it doesnt really know or care what the name
of your function is or its purpose.  When compiling the code in the
main() function it for the most part doesnt care if it is called main()
or notmain() or pickle() it does a job, it assumes that function is
called from another function and it uses the proper return instruction.
Since we called notmain() from assembly we should be prepared for the
notmain() function to return, so we should have used a branch link
instruction and put some code after the call to the notmain function.
If notmain() returns then we are pretty much done so we can put the
processor into an infinite loop, waiting for the user to turn the power
off to try another program.
```

```
.globl _start
_start:
    mov sp,#0x00010000
    bl notmain
hang: b hang
```

So bl notmain performs a branch and link, branch like the b instruction is exactly like a goto in C, a branch and link is like calling a function in C. So we have to remember to put something after the branch link in case the function returns. In this case we send it into an infinite loop.

So here we go we have patched up bootstrap.s and need to assemble it and link it with notmain.o

```
baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-ld -Ttext 0x00008000 bootstrap.o notmain.o -o hello.elf

baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e12fff1e    bx  lr

...

baremetal > arm-none-eabi-objcopy hello.elf -O binary kernel.img
baremetal > hexdump -C kernel.img
00000000  01 d8 a0 e3 00 00 00 eb  fe ff ff ea 1e ff 2f e1  |............../.|
00000010
```

Now we have a file that we can put on our sd card and run. It does nothing that we can see, so it isnt much use to us, but it will work.

We can see that the linker has prepared the program such that our first instruciton is at address 0x8000. We load the stack pointer and call notmain(). Notmain does what it does (nothing) and returns from the function call which takes us back to the hang line which is an infinite loop, hang branches to hang forever or until the power is turned off.

A few things you should have noticed. When we disassembled the object files the address was zero not 0x8000. Well the object files are by

definition incomplete programs, even if everything we are going to
run is there we should use the linker to polish that file.

This is a disassembly of the object file bootstrap.o

Disassembly of section .text:

```
00000000 <_start>:
   0:   e3a0d801    mov sp, #65536  ; 0x10000
   4:   eafffffe    b   0 <notmain>
```

Also notice that when we disassembled that object the instruction was
a branch to address zero but it had a note of notmain, well there wasnt
a notmain in that code, something linker has to fix later. Once
we linked we saw:

Disassembly of section .text:

```
00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eaffffff    b   8008 <notmain>

00008008 <notmain>:
    8008:   e12fff1e    bx  lr
```

that the instruction changed from eafffffe to eaffffff, this is
something the linker did when it figured out where notmain was going
to be in memory it had to go back and fix all the references to notmain.
Which includes instructions.

The other thing you might have noticed is Disassembly of section .text
what is a section and what is .text and what does text have to do with
my programs machine code?

Well, and this is not limited to GNU tools, for the sanity of the
compiler and assembler and linker folks, portions of our programs
are broken into categories. There is the program itself, the machine
code and some other items that are needed for the machine code to run
these are for some historical reason that I have not researched called
.text. Or the .text segment. The .data segment like the apple and
orange global variables way above. Data actually is broken up into
different segments sometimes, and in particular with the GNU tools.
Most of the code out there that has global variables the globals are
not defined, not initialized in the code, but the language declares
those as assumed to be zero when you start using them (if you have
not changed them before you read them). So there is a special data
segment called .bss which holds all of our .data that when we start
running C code should be zero. These are lumped together so that some
code can easily go through that chunk of memory and zero that
area before branching to the C entry point. Another segment we may
encounter is the .rodata segment. Sometimes even with GNU tools you
may find the read only data in the .text segment.

For fun lets make one of each:

```
unsigned int apple;
unsigned int orange=5;
const unsigned int pickle=9;

void notmain ( void )
{
    static unsigned int pear=7;
    unsigned int peach;
}

arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-objdump -D notmain.o

notmain.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <notmain>:
   0:   e12fff1e    bx  lr

Disassembly of section .data:

00000000 <orange>:
   0:   00000005    andeq   r0, r0, r5

Disassembly of section .rodata:

00000000 <pickle>:
   0:   00000009    andeq   r0, r0, r9


So we see that the code is in .text.  The pre-initialized variable
orange is in .data.  And the read only variable pickle is in .rodata.
What happened to apple and pear and peach and where is the .bss segment?
Well notice that I used -O2 on the gcc command line this means
optimization level 2.  -O0 or optimizaiton level 0 means no optimization
-O1 means some and -O2 is the maximum safe level of optimization using
the gcc compiler.  There is a -O3 but we are not supposed to trust that
to be as well tested as -O2.  I am not going to get into that but
recommend you use -O2 often, esp with embedded bare metal where size and
speed are important.  I use it here because it produces much less code
than no optimization, you can play with compiling and disassembling these
things on your own with less or without optimization to see what
happens.

So our program didnt actually use use apple, or pear or peach so the
compiler optimized those away.  We didnt use orange or pickle either
but because those were defined as something and were also both global
variables the compiler when making an object doesnt know if other code
is using those variables so it has to generate something for them for
linking with other code.
```

```
Lets try to resolve this:

unsigned int apple;
unsigned int orange=5;
const unsigned int pickle=9;

void notmain ( void )
{
    static unsigned int pear=7;
    unsigned int peach;
    apple+=pear;
}


baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-objdump -D notmain.o

notmain.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <notmain>:
   0:   e59f300c    ldr r3, [pc, #12]   ; 14 <notmain+0x14>
   4:   e5932000    ldr r2, [r3]
   8:   e2822007    add r2, r2, #7
   c:   e5832000    str r2, [r3]
  10:   e12fff1e    bx  lr
  14:   00000000    andeq   r0, r0, r0

Disassembly of section .data:

00000000 <orange>:
   0:   00000005    andeq   r0, r0, r5

Disassembly of section .rodata:

00000000 <pickle>:
   0:   00000009    andeq   r0, r0, r9

So we still see a .data segment and a .rodata and .text, but no .bss
dont worry about that just yet.  I will just tell you that since the
pear and peach variables are limited in scope to being within the notmain
function and the notmain function is so simple that the optimizer has
optimized out the peach variable completely and simply taken the
number 7 and added it to the global variable apple as a constant
basically the optimizer has replaced our code with:

void notmain ( void )
{
    apple+=7;
}
```

```
We are just disassembling the object though, which is only part of the
picture, to see the whole picture we need to link

baremetal > arm-none-eabi-ld -Ttext 0x8000 -Tdata 0x9000 -Tbss 0xA000 bootstrap.o notmain.o -o
hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   0000a000    andeq   sl, r0, r0

Disassembly of section .data:

00009000 <__data_start>:
    9000:   00000005    andeq   r0, r0, r5

Disassembly of section .bss:

0000a000 <apple>:
    a000:   00000000    andeq   r0, r0, r0

Disassembly of section .rodata:

00008024 <pickle>:
    8024:   00000009    andeq   r0, r0, r9


So our apple variable has appeared is in the .bss section.  Notice
on the linker command line I specified a few things the text segment
address and data and bss but not the rodata.  The linker again has
put the .text where we said and where we need it at 0x8000 we said
to put .data at 0x9000 and it is there and notice it has the value
5 from our orange variable.  .bss is where we said at 0xA000.  Since
we didnt specify a home for .rodata notice how the linker has just
tacked it onto the end of .text  the last thing in .text was a four
byte address at address 0x8020, so the next address after that is 0x8024
and that is where the .rodata variable pickle is placed and has
the value 9 that we pre-initialized.
```

```
I want to point something out here that is very important for general
bare metal programming.  What do we have above, something like 12 32
bit numbers which is 12*4 = 48 bytes.  So if I make this a true
binary (memory image) we should see 48 bytes right?  Well you would be
wrong:

baremetal > ls -al hello.elf
-rwxr-xr-x 1 root root 38002 Sep 23 15:06 hello.elf
baremetal > arm-none-eabi-objcopy hello.elf -O binary kernel.img
baremetal > ls -al kernel.img
-rwxr-xr-x 1 root root 4100 Sep 23 15:17 kernel.img
baremetal > hexdump -C kernel.img
00000000  01 d8 a0 e3 00 00 00 eb  fe ff ff ea 0c 30 9f e5  |.............0..|
00000010  00 20 93 e5 07 20 82 e2  00 20 83 e5 1e ff 2f e1  |. ... ... ..../.|
00000020  00 a0 00 00 09 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000  05 00 00 00                                       |....|
00001004

We can see that the first thing in the file is our code that lives
at address 0x8000, understand that the file offset and the memory offset
are not the same.  What is important is that first thing in the file
ends up at 0x8000 and since it is our entry code we are good from that
perspective.  Now why isnt the file 48 bytes?  Because a binary file
when we define it as a memory image means that if we have a few things
at 0x8000 a few things at 0x9000 and a few things at 0xA000 in order
for those things to be in the right place in the file they need to be
spaced apart, the file has to have some filler to put the important
things at the right place.

If this is at 0x8000

    8000:   e3a0d801    mov sp, #65536  ; 0x10000

And this is at 0x9000

    9000:   00000005    andeq   r0, r0, r5

Then they are 0x1000 bytes apart.  The * in the hexdump output means
I am skipping a bunch of zeros, there is nothing you are missing. The
hexdump output verifies that these two items are 0x1000 byte apart.

00000000  01 d8 a0 e3

00001000  05 00 00 00

If you keep up with bare metal embedded programming you will no doubt
at some point come across a system that has the program memory space
in a flash at some high address say 0x80000000 and the memory
where you can put your .data is at some lower address  say 0x20000000.

You can very easily try this with the code we have written simply try
```

```
a different linker command line.

baremetal > arm-none-eabi-ld -Ttext 0x8000 -Tdata 0x9000 -Tbss 0xA000 bootstrap.o notmain.o -o
hello.elf
baremetal > ls -al hello.elf
-rwxr-xr-x 1 root root 38002 Sep 23 15:26 hello.elf
baremetal > arm-none-eabi-ld -Ttext 0x80000000 -Tdata 0x20000000 -Tbss 0xA000 bootstrap.o notmain.o -o
hello.elf
baremetal > ls -al hello.elf
-rwxr-xr-x 1 root root 66710 Sep 23 15:27 hello.elf

Our file grew but if you were to try to objcopy to a -O binary format
(I recommend you DO NOT do this).  What is going to happen?


80000000:   e3a0d801    mov sp, #65536  ; 0x10000

20000000:   00000005    andeq   r0, r0, r5

There are 0x60000000 bytes between these two items, that means the
binary file created would at least be 0x60000000 bytes which is
1.6 GigaBytes.  If you are like me you probably dont always have
1.6Gig of disk space handy.  Much less wanting it to be filled with a
single file which is mostly zeros.  You can start to see the appeal for
these not really a binary binary file formats like elf and ihex and
srec.  They only define the real data and dont have to hold the zero
filler.

The stuff I wrote in the bssdata directory continues with understanding
how to control the GNU tools and segments.  For the Raspberry Pi we
dont need to deal with all of this, you are actually missing out on
some of the experience (pain).

Here is something else I hope you caught:

baremetal > arm-none-eabi-ld -Ttext 0x8000 -Tdata 0x9000 -Tbss 0xA000 bootstrap.o notmain.o -o
hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
```

```
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   0000a000    andeq   sl, r0, r0

Disassembly of section .data:

00009000 <__data_start>:
    9000:   00000005    andeq   r0, r0, r5

Disassembly of section .bss:

0000a000 <apple>:
    a000:   00000000    andeq   r0, r0, r0

Disassembly of section .rodata:

00008024 <pickle>:
    8024:   00000009    andeq   r0, r0, r9

I dont expect you to know that the notmain assembly code is reading the
thing at 0x8020

    8020:   0000a000    andeq   sl, r0, r0

Which the linker has filled in with the address to the apple variable
which is in .bss.

baremetal > arm-none-eabi-objcopy hello.elf -O binary kernel.img
baremetal > ls -al kernel.img
-rwxr-xr-x 1 root root 4100 Sep 23 15:36 kernel.img

4100 bytes.  0x8000 + 4100 = 0x8000 + 0x1004 = 0x9004  the binary
only includes an image of memory from 0x8000 to 0x9003 the objcopy
to -O binary did not include bss it was chopped off.  Why? because
in part where we specified it and because in part the toolchain
expects that the .bss segment will be zeroed by the bootstrap code
and not waste space in the binary image for that data.

But what if we were to do this:

baremetal > arm-none-eabi-ld -Ttext 0x8000 -Tdata 0xA000 -Tbss 0x9000 bootstrap.o notmain.o -o
hello.elf
baremetal > arm-none-eabi-objcopy hello.elf -O binary kernel.img

baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
```

```
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   00009000    andeq   r9, r0, r0

Disassembly of section .data:

0000a000 <__data_start>:
    a000:   00000005    andeq   r0, r0, r5

Disassembly of section .bss:

00009000 <apple>:
    9000:   00000000    andeq   r0, r0, r0

Disassembly of section .rodata:

00008024 <pickle>:
    8024:   00000009    andeq   r0, r0, r9


baremetal > ls -al kernel.img
-rwxr-xr-x 1 root root 8196 Sep 23 15:40 kernel.img
baremetal > hexdump -C kernel.img
00000000  01 d8 a0 e3 00 00 00 eb  fe ff ff ea 0c 30 9f e5  |.............0..|
00000010  00 20 93 e5 07 20 82 e2  00 20 83 e5 1e ff 2f e1  |. ... ... ..../.|
00000020  00 90 00 00 09 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00002000  05 00 00 00                                       |....|
00002004

Know your tools, know your tools, know your tools.  Now we have important
stuff at 0x8000 and 0xA000

    8000:   e3a0d801

    a000:   00000005

The file is now 8196 bytes

0x8000 + 8196 = 0x8000 + 0x2004 = 0xA004

And the objcopy -O binary has filled in the spaces with zeros so our
.bss segment is there in the binary AND it is filled with zeros!  Need
I say it again a big part of bare metal programming is knowing your
```

```
tools?


One more thing:

unsigned int apple;
void notmain ( void )
{
    apple+=7;
}


baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -Ttext 0x8000 bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm

Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   00010024    andeq   r0, r1, r4, lsr #32

Disassembly of section .bss:

00010024 <apple>:
   10024:   00000000    andeq   r0, r0, r0


We saw before that when we didnt declare a .rodata on the command line
it tacked it onto the end of .text, but in this case it didnt tack
.bss onto the end of .text it added 0x2000 bytes of padding then it
added it on there.  Why?  who knows.  The bottom line though is that
we need to take more control over how we tell the linker to do things.
In the GNU world this is through what is often called a linker script
yet another programming language that is parsed by the linker tool
where we can go to or beyond the level of crazy complication.  And
as you can guess I dont do that, I try for the minimal linker script
I dont want to be tied to a tool, I want my code to be as portable
as possible with minimal work.  Linker scripts are painful, because
so many are so complicated, few if any simple examples, it took me a
while to make to make this simple script and keep it working, I have
```

```
actually had three different solutions which I thought each time where
the simple, end all, be all, GNU linker script, they werent they worked
on one version of tools and later failed.  At this point I wouldnt be
surprised if this script also fails some day.

MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x1000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
}

baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   00008024    andeq   r8, r0, r4, lsr #32

Disassembly of section .bss:

00008024 <apple>:
    8024:   00000000    andeq   r0, r0, r0

How about that now it is all packed together nice and tight.

And to take this one step further:


unsigned int apple;
unsigned int orange=5;
const unsigned int banana=9;
void notmain ( void )
{
```

```
    apple+=7;
}

baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e59f300c    ldr r3, [pc, #12]   ; 8020 <notmain+0x14>
    8010:   e5932000    ldr r2, [r3]
    8014:   e2822007    add r2, r2, #7
    8018:   e5832000    str r2, [r3]
    801c:   e12fff1e    bx  lr
    8020:   00008028    andeq   r8, r0, r8, lsr #32

Disassembly of section .rodata:

00008024 <banana>:
    8024:   00000009    andeq   r0, r0, r9

Disassembly of section .bss:

00008028 <apple>:
    8028:   00000000    andeq   r0, r0, r0

Disassembly of section .data:

0000802c <orange>:
    802c:   00000005    andeq   r0, r0, r5


baremetal > arm-none-eabi-objcopy hello.elf -O binary kernel.img
baremetal > ls -al kernel.img
-rwxr-xr-x 1 root root 48 Sep 23 16:58 kernel.img

There we go, 12 items all packed up tight in 48 bytes of binary


00000000  01 d8 a0 e3 00 00 00 eb  fe ff ff ea 0c 30 9f e5  |.............0..|
00000010  00 20 93 e5 07 20 82 e2  00 20 83 e5 1e ff 2f e1  |. ... ... ..../.|
00000020  28 80 00 00 09 00 00 00  00 00 00 00 05 00 00 00  |(...............|
00000030
```

```
All this work so far and we have not seen the stack, we have not seen
our local variables.


bootstrap.s

.globl _start
_start:
    mov sp,#0x00010000
    bl notmain
hang: b hang

notmain.c

extern unsigned int fun ( unsigned int );
void notmain ( void )
{
    unsigned int x;

    x=fun(5);
}

fun.c

extern unsigned int more_fun ( unsigned int );
unsigned int fun ( unsigned int x )
{
    static unsigned int pear = 7;
    pear+=more_fun(x+3);
    return(pear+1);
}

more_fun.c

unsigned int more_fun ( unsigned int x )
{
    return(x+7);
}

baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-gcc -O2 -c fun.c -o fun.o
baremetal > arm-none-eabi-gcc -O2 -c more_fun.c -o more_fun.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o fun.o more_fun.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
```

```
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e92d4008    push    {r3, lr}
    8010:   e3a00005    mov r0, #5
    8014:   eb000001    bl  8020 <fun>
    8018:   e8bd4008    pop {r3, lr}
    801c:   e12fff1e    bx  lr

00008020 <fun>:
    8020:   e92d4008    push    {r3, lr}
    8024:   e2800003    add r0, r0, #3
    8028:   eb000007    bl  804c <more_fun>
    802c:   e59f3014    ldr r3, [pc, #20]   ; 8048 <fun+0x28>
    8030:   e5932000    ldr r2, [r3]
    8034:   e0800002    add r0, r0, r2
    8038:   e5830000    str r0, [r3]
    803c:   e2800001    add r0, r0, #1
    8040:   e8bd4008    pop {r3, lr}
    8044:   e12fff1e    bx  lr
    8048:   00008054    andeq   r8, r0, r4, asr r0

0000804c <more_fun>:
    804c:   e2800007    add r0, r0, #7
    8050:   e12fff1e    bx  lr

Disassembly of section .data:

00008054 <pear.4055>:
    8054:   00000007    andeq   r0, r0, r7
```

So the first thing we see is that our local global (static local)
variable pear now has its own address in memory, it did not get
optimized out.

I dont expect you to know assembly language but what I want to you to
see is a continuation of what we discussed before with respect to the
branch link instruction and the link register. The ARM instruction
set uses branch link (bl) to make function calls. The branch means
goto or jump or branch the program to some address. The link means
preserve a link back to the calling function, the hardware puts
the address of the instruciton after the branch link in the link
register so that you can return. But what happens if you have
a function that calls a function? Wont the second call overwrite the
link register, making it so you cannot return to the original
function? Yes, on the surface that is true, this is where the stack
comes in. Notice how the function fun() starts with a push and in
the brackets is the link register lr, this means save these items
on the stack and move the stack pointer. So say the stack pointer

```
was at address 0x1020 when this function was called, this means
that after the push the stack pointer is now 0x1018.  At address
0x1018 the contents of r3 will be stored and at address 0x101C the
contents of lr, the address used to return to whomever called fun().
If the first thing we did in fun() was call fun() again then
the stack pointer would go from 0x1018 to 0x1010, address 0x1010 would
get the contents of r3 and 0x1014 would get the contents of the link
register the address this instance of the fun() can needs to return,
this of course would be an infinite loop, so we didnt do that.  What
we did do is add 3 to the incoming value and call more_fun() this
branch link call to more fun modifies the link register.  More_fun
does its thing, we go through the rest of the fun() code then we pop
r3 and lr off of the stack.  Because the stack pointer has not moved
due to any other code relative to where it was when the push at the
beginnning happened, that means r3 gets back the value it had when that
push was executed and the link register also gets back its prior value,
the value we needed to return to the fun() calling function.  So that
bx lr that follows the pop returns to the proper place in notmain().
So you can see with a very small application we still need the stack
set up meaning we need the stack pointer initialized in our bootstrap
code.  The compiler assumes it has been done, if we dont and leave
that register out of our control we can get into trouble fast.

You may be asking why did I make those tiny functions separate files?
This is from experience, I knew that I was using the optimizer and
I knew what the optimizer would do.  This is important learning curve
stuff for bare metal:

notmain.c

unsigned int more_fun ( unsigned int x )
{
    return(x+7);
}
unsigned int fun ( unsigned int x )
{
    static unsigned int pear = 7;
    pear+=more_fun(x+3);
    return(pear+1);
}
void notmain ( void )
{
    unsigned int x;
    x=fun(5);
}

baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:
```

```
00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb00000a    bl  8034 <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <more_fun>:
    800c:   e2800007    add r0, r0, #7
    8010:   e12fff1e    bx  lr

00008014 <fun>:
    8014:   e59f3014    ldr r3, [pc, #20]   ; 8030 <fun+0x1c>
    8018:   e5932000    ldr r2, [r3]
    801c:   e282200a    add r2, r2, #10
    8020:   e0820000    add r0, r2, r0
    8024:   e5830000    str r0, [r3]
    8028:   e2800001    add r0, r0, #1
    802c:   e12fff1e    bx  lr
    8030:   0000804c    andeq   r8, r0, ip, asr #32

00008034 <notmain>:
    8034:   e59f300c    ldr r3, [pc, #12]   ; 8048 <notmain+0x14>
    8038:   e5932000    ldr r2, [r3]
    803c:   e282200f    add r2, r2, #15
    8040:   e5832000    str r2, [r3]
    8044:   e12fff1e    bx  lr
    8048:   0000804c    andeq   r8, r0, ip, asr #32

Disassembly of section .data:

0000804c <pear.4056>:
    804c:   00000007    andeq   r0, r0, r7
```

So you say "What is different". we still have each of the functions fun() more_fun() and notmain(), I see the local global variable pear has a home, etc. But the key difference is that notmain() has been greatly optimized. Notice how notmain does not call fun, if it doesnt call fun then that doesnt call more_fun() what the...If you follow the math in the code

notmain passes a 5 to fun.

fun passes 5+3 = 8 to morefun

morefun returns 8+7 = 15

fun saves 15 in pear
then returns 15+1 = 16

So if we wanted to optimize this code and had visibility to all of the functions we could optimize all of this code to be:

```
pear = 15;
x=16;

Actually notice how we dont do anything with the x variable in the
notmain function, we compute it but dont do anything with it?  There
is no reason to actually compute that variable, it is not used it
gets optimized out so all of this code boils down to this:

pear = 15;

And that is all that the notmain() function does, even though notmain
is not supposed to know about pear which is a local static variable
in another function, nevertheless the notmain() code is writing a 15
to pear.

I separated the files so that the compilers optimizer could not see
all of the functions and would not be able to optimize to this level.

So for example if you wanted to speed test a function, that you suspect
is slow, you might want to do something like this:

start=get_timer_tick();
answer=fun(5,6);
end=get_timer_tick();
runtime=end-start;

Where fun is some complicated algorithm or other code that you want
to speed test.  It is very important that the fun() code and this
code that calls it ARE NOT OPTIMIZED TOGETHER.  Because you hardcoded
the inputs for test purposes

fun(5,6)

where they normally might be variables:

fun(a,b)

The optimizer if allowed might simply replace all of your complicated
algorithm with:

start=get_timer_tick();
answer=42;
end=get_timer_tick();
runtime=end-start;

And this may lead you to believe that this is not the code causing
your performance problems.  Or hopefully you realize that this code
is executing way too fast and there is something wrong with your
experiment.  Knowing enough assembly code to see what is going on
will clue you into the optimization, just like in the notmain() example
above.

Lets go back to some basics and common mistakes.
```

First you may ask why am I calling the assembler and linker and gcc
all separate, cant I just put it all on one gcc command line? Sure,
you can but you are giving up control to the compiler and that
requires even more knowledge to get the command line right to get it
to build the program you want it to build. Sometimes to get the
compiler to do what you want or of you have borrowed some code you
might have to have GCC do the assembling or linking. Some folks like
to put C stuff like defines and comment symbols in their assembler code
which works fine if you feed it through gcc, but it is not assembly
language it is some sort of hybrid. Doesnt stop people from doing it,
and when you borrow that code you either have to fix the code or use the
C compiler as an assembler.

bootstrap.s

```
.globl _start
_start:
    mov sp,#0x00010000
    bl notmain
hang: b hang
```

notmain.c

```
void notmain ( void )
{
}
```

lscript

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x18000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
    .rodata : { *(.rodata*) } > ram
    .data : { *(.data*) } > ram
}
```

You might try this

```
baremetal > arm-none-eabi-gcc -Xlinker -T -Xlinker lscript bootstrap.s notmain.c -o hello.elf
/gnuarm/lib/gcc/arm-none-eabi/4.7.1/../../../../arm-none-eabi/bin/ld: cannot find crt0.o: No such file
or directory
collect2: error: ld returned 1 exit status
```

Well crt0.o is the bootstrap code the toolchain wants to use.

So lets try it this way

```
baremetal > arm-none-eabi-gcc -nostdlib -nostartfiles -ffreestanding  -Xlinker -T -Xlinker lscript
bootstrap.s notmain.c -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000000    bl  800c <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>

0000800c <notmain>:
    800c:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8010:   e28db000    add fp, sp, #0
    8014:   e28bd000    add sp, fp, #0
    8018:   e8bd0800    pop {fp}
    801c:   e12fff1e    bx  lr
```

Now I happen to always use the -nostdlib -nostartfiles -ffreestanding
with GCC when making bare metal.

Also note that I dont use

```
#include <stdio.h>
#include <stdlib.h>
```

and so on.

Well I dont use C libraries, I dont want those triggering the tools
to add more junk. Might not happen with GCC but I have seen it happen
elsewhere. Also you have to have your paths right to find those files
(that you arent using).

Here is a mistake you might make

```
baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript notmain.o bootstrap.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <notmain>:
```

```
    8000:   e12fff1e    bx  lr

00008004 <_start>:
    8004:   e3a0d801    mov sp, #65536  ; 0x10000
    8008:   ebffffc    bl  8000 <notmain>

0000800c <hang>:
    800c:   eafffffe    b   800c <hang>

Changing the order of the items on the linker command line has changed
where they are placed in the final binary.  And in this case we
are in trouble, this code wont work because the first instruction of
the boot strap is not at address 0x8000.

Now changing the linker script to have the name of the boot code in
the script and have that line before the rest of the .text

MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x18000
}

SECTIONS
{
    .text : { bootstrap.o } > ram
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
    .rodata : { *(.rodata*) } > ram
    .data : { *(.data*) } > ram
}



baremetal > arm-none-eabi-ld -T lscript notmain.o bootstrap.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000006    bl  8024 <notmain>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>
    800c:   00001541    andeq   r1, r0, r1, asr #10
    8010:   61656100    cmnvs   r5, r0, lsl #2
    8014:   01006962    tsteq   r0, r2, ror #18
    8018:   0000000b    andeq   r0, r0, fp
    801c:   01080106    tsteq   r8, r6, lsl #2
    8020:   0000012c    andeq   r0, r0, ip, lsr #2
```

```
00008024 <notmain>:
    8024:   e12fff1e    bx  lr

That fixes it, but there is other junk in our file now, not the perfect
solution.  I prefer to use ld and specify the bootstrap code first
on the command line.  And when developing a new program I disassemble
the binary before running it the first time to make sure the boot code
is where I wanted it.


Here is a situation you have a lot of data, perhaps it is a large
graphic image or a bunch of font data or something like that

bootstrap.s

.globl _start
_start:
    mov sp,#0x00010000
    bl notmain
hang: b hang

somedata.s

.space 0x10000000,0

notmain.c

void notmain ( void )
{
}

lscript

MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0xF0000000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
    .rodata : { *(.rodata*) } > ram
    .data : { *(.data*) } > ram
}


baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-as somedata.s -o somedata.o
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o somedata.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm
```

```
Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   eb000001    bl  8010 <__notmain_veneer>

00008008 <hang>:
    8008:   eafffffe    b   8008 <hang>
    800c:   00000000    andeq   r0, r0, r0

00008010 <__notmain_veneer>:
    8010:   e51ff004    ldr pc, [pc, #-4]   ; 8014 <__notmain_veneer+0x4>
    8014:   10008018    andne   r8, r0, r8, lsl r0
    ...

10008018 <notmain>:
10008018:   e12fff1e    bx  lr

You are telling me:  I dont see the problem.
The reason is the linker fixed the problem.

I am trying to put the tool in a position where it has assembled a
single instruction for the branch link, which is limited in how
far in memory it can go.  What the linker did is it created some
code near the branch link, somewhere it could reach and used that
as what I call a trampoline.  The tools have performed the branch
link at the right place so the return address is in the link register
then it used location that reads a value from memory and puts that
in the program counter meaning it branches to that address.  Being a
branch it does not modify the link register so notmain doesnt know
any better how the program got there it returns to the right place.

If we combine the two into one file

bootstrap.s

.globl _start
_start:
    mov sp,#0x00010000
    bl notmain
hang: b hang
.space 0x10000000,0

and dont use somedata.s

baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
bootstrap.o: In function `_start':
(.text+0x4): relocation truncated to fit: R_ARM_CALL against symbol `notmain' defined in .text section
in notmain.o

Now the problem is that the linker is unable to find a place close enough
```

to the bl instruction to put a trampoline so it has to error out. This is not necessarily the exact error message I was after but it will do.

The ARM instructions have quite a bit of a reach. Other instruction sets have different limitations as to how far a branch can go and how you place the object files on the command line can affect how far the branches have to go to get from one place to another and the linker may not be able to patch it.

At this point I hope you have more than enough of a feel for the kinds of things you need to know from a gnu toolchain perspective to get started with ARM bare metal programming on the Raspberry Pi.

Also, a side effect is that I hope that you can see without actually buying any hardware or running any code we were able to perform many experiments and learn many things about the tools. It doesnt matter what instruction set or computer you can often do similar things, certainly with the GNU tools, create simple functions compile and disassemble just that function, or link it with something simple enough to get the linker to stop complaining.

Now I am going to move into thumb mode, which creates a number of other problems that can be quite difficult to find.

Traditionally ARM has used 32 bit instructions, fixed instruction length. Then the thumb instruction set was added. The original thumb instruction set had a one to one relationship with a full sized ARM instruction. I have no direct knowledge but assume that the thumb instructions were converted to ARM instructions before being executed so that there only needed to be one execution unit in the processor. The thumb instructions are 16 bits wide, originally fixed length, thumb2 extensions to the thumb instruction set create a bit of a mess with 16 and 32 bit thumb instructions. The 16 bit instructions provide some cost and performance benefits for embedded systems. First off you can pack more instructions into the same amount of memory, understanding that it may take more instructions to perform the same task using thumb instructions than it would have using ARM. My experiments at the time showed about 10-15% more instructions, but half the memory so that was a fair tradeoff. I know of one platform that went so far as to use 16 bit memory busses, which actually made thumb mode run much faster than ARM mode on that platform. That platform is/was the Nintendo Gameboy Advance.

There are very specific rules for switching modes between the two modes. Specifically you have to use the bx (or blx) instruction. When you use the bx instruction the least significant bit of the address in the register you are using determines if the mode you switching to as you branch is ARM mode or thumb mode. ARM mode the bit is zero, thumb mode the bit is a 1. This may not be obvious and the ARM documents are a little misleading or incorrect as to what valid

```
bits you can have in that register.  Note that that lower bit
is stripped off it is only used by the bx instruction itself the
address in the program counter always has the lower two bits zero
for ARM mode (4 byte instructions) and the lower bit zero for
thumb instructions (2 or 4 byte instructions).  Note the bx/blx
instruction is not the only way to switch modes, sometimes you can
use the pop instruction, but bx works the same way on all ARM
architectures that I know of, the other solutions (pop for example)
vary in if/how they work for switching modes depending on the ARM
architecture in question.  So that makes for very unportable code
across ARM if you are not careful.  When in doubt just use BX.

Here again the goal is not to teach assembly but you may want to
get the ARM Architectural Reference Manual for this platform
(see the top level README file) so that you can look at the
ARM and thumb instructions as well as other things that describe at
least in part what I am talking about.  For example this flavor of
ARM boots in a normal ARM way meaning the exception table is filled
with 32 bit ARM instructions that get executed.  Address 0x00000000
contains the instruction executed on reset, 0x00000004 some other
exception and so on, one for interrupt one for fast interrupt one
for data abort, one for prefetch abort, etc.  At least the traditional
ARM exception table, in recent years both the Cortex-M which is
different and the ARM exception table are seeing changes from the past.
Anyway, I bring this up because it is important to know that in this
case all exceptions are entered in ARM mode, even if you were in thumb
mode when you were interrupted or otherwise had an exception.  The cpsr
contains a T bit which is the mode bit, when you return from the
interrupt or exception the cpsr is restored along with your
program counter and you return to the mode you were in.  This is the
exception to the rule that you use bx to change modes (or blx).

So the ARM is going to come out of reset in ARM mode and whatever
mechanism that the Raspberry Pi uses to have our code at 0x8000 run we
start running our code in full 32 bit ARM mode.

You probably know that the C language has somewhat of a standard
every so often that standard is re-written and if you want to make a
C compiler that conforms to that standard...well you conform or at
least try.  Assembly language in general does not have a standard.
A company designs a chip, which means they create an instruction set,
binary machine code instructions, and generally they create an
assembly language so that they can write down and talk about those
instructions using mnemonics instead of patterns of ones and zeros.
And not always but often if that company actually wants to sell those
processors, so they create or hire someone to create an assembler and
a compiler or few.  Assembly language, like C language, has
directives that are not actually code like #pragma in C for example
you are using that to talk to the compiler not using it as code
necessarily.  Assembly has those as well, many of them.  It is in the
processor vendors best interest to use the same assembly language
syntax for the instructions in the processor manual in the assembler
that they create or have someone create for them.  But that manual
although you might consider it a standard, is not, the machine code is
```

the hard and fast standard, the ASCII assembly language is fair game and
anyone can create their own assembly language for that processor
with whatever syntax and directives that they want. ARM has a nice
set of compiler tools, or at least when I worked at a place that paid
for the tools for a few years and tried them they were very nice and
conformed of course to the ARM documents. GNU assembler, in true
GNU assembler fashion does not like to conform to the vendors assembly
language and generally makes some sort of a mess out of it. Fortunately
the ARM mess is nowhere near as bad as the x86 mess. Subtle things
like the comment symbol are the most glaring problems with GNU assembler
for ARM. Anyway, I dont remember the syntax or directives for the
ARM tools, the ARM tools have evolved anyway. At the time I did try
to write asm that would compile on both ARMs tools and gnus tools with
minimal massaging, and you will forever see me use ;@ for comments
instead of @ because this ; is the proper, almost universal, symbol for
a comment in assembly languages from many vendors. This @ is not.
Combined like this ;@ and you get code that is commented in both worlds
equally. Enough with that rant, this asm code will continue to be GNU
assembler specific as that is the toolchain I am using, I dont know if
it works on any other assembler, I keep the directives to a bare
minimum though.

Another side effect of thumb and in particular thumb2 is that ARM
decided to change their syntax in subtle ways to come up with a unified
syntax, for example to perform the addition r0 = r0 + r1

Thumb:
add r0,r1

ARM
add r0,r0,r1

Early on you had to write all three registers, but for thumb part of
the reduction is one source and the destination have to be the same
register for many of the alu instructions. Now even not the unified but
certainly the unified syntax attempted to resolve this into a dumbed
down instruction set. Naturally the unfied cant do everythign of every
one of the flavors (ARM, thumbv1 and v2), for the most part you basically
get to write thumb code and have it assemble for ARM without complaints.
The GNU assembler has also adopted the unified syntax and relaxed its
rules on the non-unified syntax. I have not switched over to using the
unified syntax...yet. Eventually I will be forced to and then at that
time I will likely always use it...

There are games you need to play with assembly language directives
using the GNU assembler in order to get the tool to properly create
thumb address for use with the bx instruction so you dont have to
be silly and add one or or one to the address before you use it.

So our normal ARM boostrap code:

```
.globl _start
_start:
    mov sp,#0x00010000
```

```
    bl notmain
hang: b hang
```

For running in thumb mode I recommend going all the way, run everything you can in thumb. We have to have some bootstrap in ARM mode, but after that it makes your life easier from a compiling and linking perspective to go all thumb after the bootstrap. lets dive in.

bootstrap.s

```
.code 32
.globl _start
_start:
    mov sp,#0x00010000
    ldr r0,thumbstart_add
    bx r0

thumbstart_add: .word thumbstart

;@ ----- ARM above, thumb below
.thumb

.thumb_func
thumbstart:
    bl notmain
hang: b hang
```

notmain.c

```
void notmain ( void )
{
}
```

lscript

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x18000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
    .rodata : { *(.rodata*) } > ram
    .data : { *(.data*) } > ram
}
```

```
baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
```

```
baremetal > arm-none-eabi-gcc -mthumb -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008011    andeq   r8, r0, r1, lsl r0

00008010 <thumbstart>:
    8010:   f000 f802   bl  8018 <notmain>

00008014 <hang>:
    8014:   e7fe        b.n 8014 <hang>
    8016:   46c0        nop         ; (mov r8, r8)

00008018 <notmain>:
    8018:   4770        bx  lr
    801a:   46c0        nop         ; (mov r8, r8)
```

So we see the ARM instructions mov sp, ldr r0, and bx r0. These
are 32 bit instructions and most of them start with an E which makes
them kind of stand out in a crowd. The .code 32 directive tells
the assembler to assemble the following code using 32 bit arm
instructions or at least until I tell you otherwise. the .thumb
directive is me telling the assembler otherwise. Start assembling
using 16 bit thumb instructions. Yes the bl is actually two separate
16 bit instructions and are documented by ARM as such, but always shown
as a pair in disassembly. It is not a 32 bit instruction.

The .thumb_func is used to tell the assembler that the label
that follows is branch destination for thumb code, when you see this
label set the lsbit so that I dont have to play any games to switch
or stay in the right mode. You can see that the thumbstart label
is at address 0x8010, but the thumbstart_add is 0x8011, the thumbstart
address with the lsbit set, so that when it hits the bx instruction
it tells the processor that we want to be in thumb mode. Note that
bx can be used even if you are staying in the same mode, that is the key
to it, if you have used the proper address you dont care what
mode you are branching to. You can write code that calls functions
and the code making the call can be thumb mode and the code you are
calling can be ARM mode and so long as the compiler and/or you has
not messed up, it will properly switch back and forth. Problem is
the compiler doesnt always get it right. You may see or hear

```
the word interwork or thumb interwork (command line options for the
compiler/tools) which puts extra stuff in there to hopefully have
it all work out.  I prefer as you know to use few/no gcclib or
clib canned functions (which can be in the wrong mode depending on
your tools and how lucky you are when linking) and I prefer other
than the asm startup code to remain as thumb pure as possible to
minimize any of these problems.  This part of the tutorial of course
is not necessarily about staying thumb pure but showing the problems
or at least possible problems you will no doubt see when trying to use
thumb mode.

So the simple program above all worked out fine, by remembering to
place the .thumb_func directive before the label we told the assembler
to compute the right address, what if we forgot?


.code 32
.globl _start
_start:
    mov sp,#0x00010000
    ldr r0,thumbstart_add
    bx r0

thumbstart_add: .word thumbstart

;@ ----- ARM above, thumb below
.thumb

thumbstart:
    bl notmain
hang: b hang


baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008010    andeq   r8, r0, r0, lsl r0

00008010 <thumbstart>:
    8010:   f000 f802   bl  8018 <notmain>

00008014 <hang>:
```

```
    8014:   e7fe        b.n 8014 <hang>
    8016:   46c0        nop         ; (mov r8, r8)

00008018 <notmain>:
    8018:   4770        bx  lr
    801a:   46c0        nop         ; (mov r8, r8)
```

Not a single peep from the compiler tools and we have created perfectly broken code. It is hard to see in the dump above if you dont know what to look for but it will make for a very long day or very expensive waste of time playing with thumb if you dont know what to look for. that little 0x8010 being loaded into r0 and then the bx r0 in ARM mode is telling the processor to branch to address 0x8010 AND STAY IN ARM MODE. But the instructions at 0x8010 and the ones that follow are thumb mode, they might line up with some sort of ARM instruction and the ARM may limp along executing gibberish, but at some point in a normal sized program it will hit a pair of thumb instructions whose binary pattern are not a valid ARM instruction and the arm will fire off the undefined instruction exception. One wee little bit is all the difference between success and massive failure in the above code.

Now lets try mixing the modes and see what the tool does. I am running a somewhat cutting edge gcc and binutils as of this writing:

```
baremetal > arm-none-eabi-gcc --version
arm-none-eabi-gcc (GCC) 4.7.1
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

baremetal > arm-none-eabi-as --version
GNU assembler (GNU Binutils) 2.22
Copyright 2011 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `arm-none-eabi'.
```

I have been using the gnu tools for ARM since the 2.95.x days of gcc. starting with thumb in the 3.x.x days pretty much every version from then to the present. And there have been good ones and bad ones as to how the mixing of modes is resolved. I have to say these newer versions are doing a better job, but I know in recent months I did trip it up, will see if I can again.

Fixing our bootstrap and not using the -mthumb option, builds ARM code:

```
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm
```

```
Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008011    andeq   r8, r0, r0, lsl r0

00008010 <thumbstart>:
    8010:   f000 f806   bl  8020 <__notmain_from_thumb>

00008014 <hang>:
    8014:   e7fe        b.n 8014 <hang>
    8016:   46c0        nop         ; (mov r8, r8)

00008018 <notmain>:
    8018:   e12fff1e    bx  lr
    801c:   00000000    andeq   r0, r0, r0

00008020 <__notmain_from_thumb>:
    8020:   4778        bx  pc
    8022:   46c0        nop         ; (mov r8, r8)
    8024:   eafffffb    b   8018 <notmain>
```

very nicely handled. after thumbstart they use a bl instruction
as we had in the assemblly language code so that the link register
is filled in not only with a return address but the return address
with the lsbit set so that we return to the right mode with a bx lr
instruction. Instead of branching right to the ARM code though
which would not work you cannot use bl to switch modes, they
branch to what I call a trampoline, when they hit
__notmain_from_thumb the link register is prepped to return to address
0x8014. I am not teaching you assembly just how to see what is going
on, but this next thing is advanced even for assembly programmers.
In whichever mode the program counter points to two instructions ahead
so in this case we are running instruction 0x8020 bx pc in thumb mode
thumb mode is 2 bytes per instruction, two instructions ahead is the
address 0x8024 and note that that address has a zero in the lsbit so
this is a cool trick, the linker by adding these instructions at a
four byte aligned address (lower two bits are zero) 0x8020 then doing
a bx pc, and sticking a nop in between although I dont think it matters
what is there. The bx pc causes a switch to ARM mode and a branch to
address 0x8024, which being a trampoline to bounce off of, that instruction
bounces us back to 0x8018 which is the ARM instruction we wanted
to get to. this is all good, this code will run properly.

You may or may not know that compilers for a processor follow a "calling
convention" or binary interface or whatever term you like. It is a set

```
of rules for generating the code for a function so that you can have
functions call functions call functions and any function can
return values and the code generated will all work without having to
have some secret knowledge into the code for each function calling it.
Conform to the calling convention and the code will all work together.
Now the conventions are not hard and fast rules any more than assembly
language is a standard for any particular processor.  These things
change from time to time in some cases.  For the ARM, in general across
the compilers I have used the first four registers r0,r1,r2,r3 are
used for passing the first up to 16 bytes worth of parameters, r0 is
used for returning things, etc.   I find it surprising how often
I see someone who is trying to write a simple bit of assembly what
the calling convention is for a particular processor using a particular
compiler.  Most often gcc for example.  Well why dont you ask the
compiler itself it will tell you, for example:

unsigned int fun ( unsigned int a, unsigned int b )
{
    return((a>>1)+b);
}


baremetal > arm-none-eabi-gcc -O2 -c fun.c -o fun.o
baremetal > arm-none-eabi-objdump -D fun.o

fun.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <fun>:
   0:   e08100a0    add r0, r1, r0, lsr #1
   4:   e12fff1e    bx  lr

So what did I just figure out?  Well if I had that function in C and
used that compiler and linked in that object code it would work with
other code created by that compiler, so that object code must follow
the calling convention.  What I figured out is from that trivial experiment
is that if I want to make a function in assembly code that uses two
inputs and one output (unsigned 32 bits each) then the first parameter,
a in this case, is passed in r0, the second is passed in r1, and the
return value is in r0.  let me jump to a complete different processor
for a second.


Disassembly of section .text:

00000000 <fun>:
   0:   b8 63 00 41     l.srli r3,r3,0x1
   4:   44 00 48 00     l.jr r9
   8:   e1 64 18 00     l.add r11,r4,r3

This is not ARM but some completely different instruction set, and the
compiler for it has a different calling convention.  What I see here is
```

```
that the first parameter is passed in register r3, the second parameter
is passed in r4 and the return value goes back in r11.  and it just
so happens that the link register is r9.

Yes, it is true that I have not yet figured out what registers
I can modify without preserving them and what registers I have to
preserve, etc, etc.  You can figure that out with these simple experiments
with practice.  Because sometimes you may think you have found the
docment describing the calling convention only to find you have not.
And as far as preservation, if in doubt preserve everything but the
return registers...

So if you have looked at my work you see that I prefer to perform
singular memory accesses using hand written assembly routines like
PUT32 and GET32.  Not going to say why here and now, I have mentioned
it elsewhere and it doesnt matter for this discussion.  Lets accept
it and move on to use it, a quick thumb experiment:


baremetal > arm-none-eabi-gcc -mthumb -O2 -c fun.c -o fun.o
baremetal > arm-none-eabi-objdump -D fun.o

fun.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <fun>:
   0:   0840        lsrs    r0, r0, #1
   2:   1808        adds    r0, r1, r0
   4:   4770        bx  lr
   6:   46c0        nop         ; (mov r8, r8)

r0 is first paramter, r1 second, and return value is r0.

So to create a PUT32 in thumb mode, since we already have some
assembly in our project, lets just put it there:

bootstrap.s

.code 32
.globl _start
_start:
    mov sp,#0x00010000
    ldr r0,thumbstart_add
    bx r0

thumbstart_add: .word thumbstart

;@ ----- ARM above, thumb below
.thumb

.thumb_func
thumbstart:
```

```
    bl notmain
hang: b hang

.thumb_func
.globl PUT32
PUT32:
    str r1,[r0]
    bx lr


And use it in notmain.c

void PUT32 ( unsigned int, unsigned int );
void notmain ( void )
{
    PUT32(0x0000B000,0x12345678);
}

And make notmain ARM code



baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-gcc -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008011    andeq   r8, r0, r1, lsl r0

00008010 <thumbstart>:
    8010:   f000 f818   bl  8044 <__notmain_from_thumb>

00008014 <hang>:
    8014:   e7fe        b.n 8014 <hang>

00008016 <PUT32>:
    8016:   6001        str r1, [r0, #0]
    8018:   4770        bx  lr
    801a:   46c0        nop         ; (mov r8, r8)

0000801c <notmain>:
    801c:   e92d4008    push    {r3, lr}
    8020:   e3a00a0b    mov r0, #45056  ; 0xb000
```

```
    8024:   e59f1008    ldr r1, [pc, #8]    ; 8034 <notmain+0x18>
    8028:   eb000002    bl  8038 <__PUT32_from_arm>
    802c:   e8bd4008    pop {r3, lr}
    8030:   e12fff1e    bx  lr
    8034:   12345678    eorsne  r5, r4, #125829120  ; 0x7800000

00008038 <__PUT32_from_arm>:
    8038:   e59fc000    ldr ip, [pc]    ; 8040 <__PUT32_from_arm+0x8>
    803c:   e12fff1c    bx  ip
    8040:   00008017    andeq   r8, r0, r7, lsl r0

00008044 <__notmain_from_thumb>:
    8044:   4778        bx  pc
    8046:   46c0        nop         ; (mov r8, r8)
    8048:   eafffff3    b   801c <notmain>
    804c:   00000000    andeq   r0, r0, r0

So we start in arm, use 0x8011 to swich to thumb mode at address 0x8010
trampoline off to get to 0x801C entering notmain in ARM mode.  and we
branch link to another trampoline.  This one is not complicated as
we did this ourselves right after _start.  Load a register with
the address orred with one.  0x8017 fed to bx means switch to thumb
mode and branch to 0x8016 which is our PUT32 in thumb mode.

lets go the other way, PUT32 in ARM mode called from thumb code


baremetal > arm-none-eabi-as bootstrap.s -o bootstrap.o
baremetal > arm-none-eabi-gcc -mthumb  -O2 -c notmain.c -o notmain.o
baremetal > arm-none-eabi-ld -T lscript bootstrap.o notmain.o -o hello.elf
baremetal > arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008019    andeq   r8, r0, r9, lsl r0

00008010 <PUT32>:
    8010:   e5801000    str r1, [r0]
    8014:   e12fff1e    bx  lr

00008018 <thumbstart>:
    8018:   f000 f802   bl  8020 <notmain>

0000801c <hang>:
    801c:   e7fe        b.n 801c <hang>
```

```
    801e:   46c0        nop         ; (mov r8, r8)

00008020 <notmain>:
    8020:   b508        push    {r3, lr}
    8022:   20b0        movs    r0, #176    ; 0xb0
    8024:   0200        lsls    r0, r0, #8
    8026:   4903        ldr r1, [pc, #12]   ; (8034 <notmain+0x14>)
    8028:   f7ff fff2   bl  8010 <PUT32>
    802c:   bc08        pop {r3}
    802e:   bc01        pop {r0}
    8030:   4700        bx  r0
    8032:   46c0        nop         ; (mov r8, r8)
    8034:   12345678    eorsne  r5, r4, #125829120  ; 0x7800000
```

And we did it, this code is broken and will not work. Can you see the problem? PUT32 is in ARM mode at address 0x8010. Notmain is thumb code. You cannot use a branch link to get to ARM mode from thumb mode you have to use bx (or blx). The bl 0x8010 will start executing the code at 0x8010 as if it were thumb instructions, and you might get lucky in this case and survive long enogh to run into the thumbstart code which in this case puts you right back into notmain sending you into an infinite loop. One might hope that at least the ARM machine code at 0x8010 is not valid thumb machine code and will cause an undefined instruction exception which if you bothered to make an exception handler for you might start to see why the code doesnt work.

It was very easy to fall into this trap, and very very hard to find out where and why the failure is until you have lived the pain or been shown where to look. Even with me showing you where to look you may still end up spending hours or days on this. But as you do know as an experienced programmer each time you spend hours or days on some bug, you learn from that experience and the next time you are much faster at recognizing the problem and where to look. If you happen to get bitten a few times you should get very fast at finding the problem.

If I add this

notmain.c

```
extern unsigned int fun ( unsigned int, unsigned int );
extern void PUT32 ( unsigned int, unsigned int );
void notmain ( void )
{
    fun(123,456);
    PUT32(0x0000B000,0x12345678);
}
```

and this

```
unsigned int fun ( unsigned int a, unsigned int b )
```

```
{
    return((a>>1)+b);
}


dwelch-desktop baremetal # arm-none-eabi-gcc -O2 -c fun.c -o fun.o
dwelch-desktop baremetal # arm-none-eabi-ld -T lscript bootstrap.o notmain.o fun.o -o hello.elf
dwelch-desktop baremetal # arm-none-eabi-objdump -D hello.elf

hello.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:   e3a0d801    mov sp, #65536  ; 0x10000
    8004:   e59f0000    ldr r0, [pc]    ; 800c <thumbstart_add>
    8008:   e12fff10    bx  r0

0000800c <thumbstart_add>:
    800c:   00008019    andeq   r8, r0, r9, lsl r0

00008010 <PUT32>:
    8010:   e5801000    str r1, [r0]
    8014:   e12fff1e    bx  lr

00008018 <thumbstart>:
    8018:   f000 f802   bl  8020 <notmain>

0000801c <hang>:
    801c:   e7fe        b.n 801c <hang>
    801e:   46c0        nop         ; (mov r8, r8)

00008020 <notmain>:
    8020:   b508        push    {r3, lr}
    8022:   21e4        movs    r1, #228    ; 0xe4
    8024:   0049        lsls    r1, r1, #1
    8026:   207b        movs    r0, #123    ; 0x7b
    8028:   f000 f80e   bl  8048 <__fun_from_thumb>
    802c:   20b0        movs    r0, #176    ; 0xb0
    802e:   0200        lsls    r0, r0, #8
    8030:   4902        ldr r1, [pc, #8]    ; (803c <notmain+0x1c>)
    8032:   f7ff ffed   bl  8010 <PUT32>
    8036:   bc08        pop {r3}
    8038:   bc01        pop {r0}
    803a:   4700        bx  r0
    803c:   12345678    eorsne  r5, r4, #125829120  ; 0x7800000

00008040 <fun>:
    8040:   e08100a0    add r0, r1, r0, lsr #1
    8044:   e12fff1e    bx  lr

00008048 <__fun_from_thumb>:
    8048:   4778        bx  pc
```

```
    804a:   46c0        nop         ; (mov r8, r8)
    804c:   eafffffb    b   8040 <fun>
```

fun() which is in ARM mode, when called from notmain() which is thumb
mode is handled properly. So there is something there that tells the
linker that fun is ARM and needs a mode change.

When we use .thumb_func for thumb functions in assembly that triggers
the linker to do the right thing. I wonder if there is something
in ARM functions in assembly that we can use to do the same thing.

This is another one of my personal preferences: when using thumb mode
on an ARM booting system I use the minimal ARM code to get into thumb
mode in the bootstrap code then everywhere else I stay in thumb mode
as far as I know. If there is a time where I need ARM mode then I
am careful to see if the tools changed mode properly or I may do my
own mode change the tools dont have to get it right.

this is a rough draft, if/when I complete this draft I will at some point
go back through and rework it to improve it.