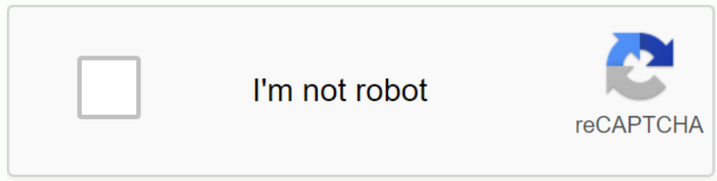


Continue

# Bare metal programming in c pdf

This repository is a tutorial ebook on programming bar metal ARM system. More specifically, it relates to the ARMv7A version of the ARM Universal Express platform, emulated regularly on pc via QEMU. You can explore the repository or read things in a row. Content Is also the latest version of the PDF. These sections can be perceived as the first edition of the ebook, providing all the way from getting started to a work program that uses hardware features. Chapter 0: Introduction. A brief introduction to the topic and ebook. Chapter 1: Setup. A short section discussing the preparation of the Linux environment for further development. Chapter 2: First boot. Basic use of QEMU and cross-compiler toolchain to run the simplest code. Chapter 3: Add a Bootloader. Building a very popular U-Boot bootloader, and getting it to run your code. Chapter 4: Preparation of environment C. This section discusses the necessary work to get the collection code to code C from startup. Chapter 5: Build &; harmonisation system. Here we show how the work can be simplified by adding a CMake-based build system, and how a bald metal program can be matched. Chapter 6: Improvement of the UART driver. This section saves the UART device driver. Chapter 7: Interrupts. The section goes through setting up an ARM Generic Terminate Controller, using it to get in and respond to interruptions. Also the UART driver gets customized to use the interrupt. Chapter 8 - WIP: Planning. Work is underway. Storage storage structure The repository consists of two top-level folders. Doc folder contains the actual tutorial sections. The Src folder contains the source code for each section. So, for example, src / 04_cenv is the source code, as it seems after completing chapter 4. Additionally, the src folder contains several shared things. src/common_uboot has a stripped down version of U-Boot used for samples. Have fun, and feel free to tweak and experiment, that is a great way to learn! Page 2 Watch 14 Star 179 Fork 39 At this time you can not perform this action. You're signed in with a different tab or window. Reload to update the session. You've disconnected on another tab or window. Reload to update the session. We use custom third-party analytics cookies to understand how you use GitHub.com to create better products. learn more, find out more. We use custom third-party analytics cookies to understand how you use GitHub.com to create better products. You can always update the selection by clicking Cookie Preferences at the bottom of the page. For more information, see the privacy statement. We use essential cookies to perform essential features of the website, such as they are used to sign in. Learn more Always active We use analytics cookies to understand how you use our websites so that we can do better, such as they are used for information about pages, you are visiting, collect and how many clicks are needed to perform To learn more, I'm surprised how often I'm asked about how you can go about programming bare metal – i.e. without any kind of OS support. Historically, it was a common approach - maybe there was no other choice. But modern engineers, faced with the idea of filling a completely empty memory with their own code, are often a little flummoxed... In the early days of embedded systems development, the software was quite minimal and often something late. Typically, it was developed by the same engineer(s) who developed the hardware and, of course, their code interacted very closely with the electronics. They understood all the nuances of hardware behavior, so it was not considered a specific challenge. As the systems became a little more complex, software specialists began to participate, but they tended to be engineers with a lot of knowledge and understanding of hardware, so they were also quite happy programming close to hardware. As the complexity increased, the only software engineer became a team. Different team members would have different kinds of experiences. Those with good hardware knowledge envelop this experience with software modules that provided a clean interface and concealed the complexity of hardware interaction; these modules were called drivers. With increasingly powerful microprocessors/microcontrollers and larger memories, the need for a rational program structure has led to the adoption of real-time operating systems (RTOSes), which have enabled the multi-task model to be used. It was a natural progression for drivers to become part of RTOS. But what about programming bare metal today? While the largest part of modern embedded software designs are implemented using a certain type of OS, there are several circumstances when doing without - programming on bare metal - can be a reasonable solution: If the application is very simple and is implemented, perhaps, on a low-end processor. If you need to extract every last cycle of cpu power application and overhead introduced OS is unacceptable. (1) there is a real possibility and makes sense. (2) is much less likely because powerful CPU are very easily accessible and cheap; careful choice of OS, focusing on high performance RTOS products, would be the most meaningful. In both cases, consideration should be made of possible future software improvements. If further development is likely, starting with the scalable program structure is worth investing in. Although Raspberry-Pi has a good Linux distribution, Pi is about software development, and sometimes we want a real-time system without an operating system. I decided it would be great to make a tutorial at Linux to get into this great piece of hardware resources in a similar vein to Cambridge University Tutorials, which are perfectly written. However, they do not create os as supposedly, and they and not migrate to C. I just start with nothing but a collector to get us going, but move to C as soon as possible. C compiler simply converts c syntax collector and then collects this executable code for us anyway. I highly recommend going through cambridge university Raspberry Pi tutorials because they are great. If you want to know a little collector too, then really head there! These pages provide a similar experience, however, with writing code C and understanding the process behind it. TODO¶ Why is a 16MiB card required when we are not using anywhere near? Compatibility¶ There are quite a few versions of RPi these days. This part of the tutorial supports the following models: RPi model RPi model B RPi Zero RPi Zero W RPi Model B + RPi 2 Model B RPi 3 Model B RPi 4 Model B Note It is not a mistake that RPI 3 model B + is not included in this list. ACK LED is only available through the mailbox interface (available from tutorial 4) and therefore can not be used directly gpio peripheral, which we will use in this part of the tutorial. Cross compilation for Raspberry Pi (BCM2835/6/7/BCM23711)¶ ARM has now taken over the arm of the GCC embedded project and is scheduled for release, so pop through arm gcc download section and pick up toolchain. I just grabbed 7.3.1 from the download page and I locally installed it on my Linux machine to use with this tutorial. This is what I get when I run this on my command line, wielding decompressed archive: ~/arm-tutorial-rpi/compiler/gcc-arm-none-eabi-7-2018-q2-update/bin $./arm-none-eabi-gcc-version arm-none-eabi-GCC (GNU tools arm embedded processors 7-2018-q2-update) 7.3.1 20180622 \ (release) [ARM/embedded-7-affiliate review 261907] Copyright (C) 2017 Free Software Foundation, Inc. It is free software; copy conditions. There are no guarantees; even on the right of merchantability or fitness for a particular purpose. Cool. You can use compiler / get_compiler.sh script as a short cut to get compiler and tutorial script will take advantage of it if you do. Compiler version¶ Previously with before stabbing in this tutorial I always related and recommended a fixed, known working version of the compiler, because everything then just worked out of the box. But now I say - just get the latest and then we can set the tutorial as things break. NOTE: The dismantling list for you may be slightly different from those generated by the GCC version used in this tutorial if you use a different compiler version or option set. RPI1 Compiler Flag¶ eLinux page gives us optimal GCC settings, for the original Raspberry-Pi (V1) compilation code: -Ofast -mfpu=vfp-mfloat-abi=hard-march=armv6zk-mtune=arm1176jzf-s It should be noted that -Ofast can cause problems with some compilation, so it is probably better that we stick to the more traditional -O2 optimization setting. Other flags tell GCC what type of floating point unit we have, tell it to produce solid floating point code (GCC can create software floating point support instead), and tells GCC what ARM processor architecture we have so that it can produce optimal and compatible capture /machine code. RPI2 Compiler Flags¶ Raspberry-Pi 2 we know that the architecture is different. ARM1176 from the original pi was replaced by a quad-core Cortex A7 processor. Therefore, in order to assemble effectively Raspberry-Pi 2 we use another set of compiler options: -O2 -mfpu = neon-vfpv4 -mfloat-abi = hard-march = armv 7-a-mtune=cortex-a7 From cortex A7 ARM specifications you can see, that it contains vfpv4 (see section 1.2.1) floating point processor and NEON engine. The settings are gleaned from the GCC ARM options page. RPI3 Compiler flags¶ Like this: -O2 -mfpu=crypto-neon-fp-armv8-mfloat-abi=hard-march=armv8-a+crc-mcpu=cortex -a53 RPI4 Compiler flags¶ From raspberry pi foundation page RPi4 we can gather some information from technical specifications about what we need to do to collect RPi4 code. All four processors are A72. From arm documentation we can see that they implement the architecture of armv8. It's the same as the A53 found in RPi3 so we can go ahead and use the same crypto-neon-fp-armv8 floating point unit option RPI4. This is described in detail by v8 architectural programmers guide -O2 -mfpu = crypto-neon-fp-armv8-mfloat-abi=hard-march=armv8-a+crc-mcpu=cortex-a72 Schemes you think would be a place to get an ACT LED GPIO port number, but unfortunately they are so often they may also not bother releasing them. Seriously - what they released is a joke. Instead we get it from some device tree source code RPi4. Also make sure you have the latest software, fixes are always entered! In order to use the C compiler, we need to understand what the compiler does and what the link does to generate executable code. The compiler converts C sentences to a collector and performs optimization of assembly instructions. It's a fact all the C compiler does! C compiler then indirectly calls the collector to collect this file (usually temporary) into the object file. It will move the device code along with the character information used by the link. These days c compiler pipes assembly to collector, so there is no intermediate file, how to create files is much slower than transferring data from one program to another through a pipe. Linker's job is to link everything to an executable file. The pairing program requires a mapping script. The Link Builder script tells the link builder how to manage various entity files. The link link will resolve the characters to addresses when it organizes all entities according to the mapping script rules. We are approaching here that program C is not just the code we entered. there are some basic things that have to happen in the C code run. For example, some variables need to be initialized based on certain values, and some variables need to be initialized to 0. All this takes care of the object file, which is usually indirectly linked by the linker, because the linker script will reference it. An object file called crt0.o (C runtime zero) This code uses characters so that links can resolve to clear the Start area where the initialized variables begin and end to the zero section of this memory. It usually sets the stack pointer, and it always includes a call to the _main. Here's an important note: The characters in code C are prepended with an underscore in the collector version of the code generation underscore. So when the start of program C is the main symbol, the collector we need to turn to it, because it's

the collector version that is _main. Github¶ All tutorials source is a repo from Github. So go clone or fork now, so you have the whole code to collect and change as you work through tutorials. git clone if you are in Windows - these days I will say just get with the program and get yourself linux installed. If you enter the world of Raspberry Pi and/or embedded devices, Linux will be your friend and will give you everything you need. This tutorial is used to support both Linux and Windows, but I do not have Windows installs on the left and therefore can not cover disable Windows. If someone thinks that, I fully support them to update the tutorial - everything github. Let's have a compilation of one of the easiest programs that we can search for. Allow me to compile and link this program (Part 1/armc-00): part-1/armc-00¶ int main(void) ( o (1) { } return 0; } Compilation¶ In order to compile the code (I understand, there is not much of that code!) We can use build.sh script in the tutorial directory. Go to Part 1/armc-00 and start ./build.sh. Without any arguments this script will only show you what he expects to run. arm-tutorial-rpi/part-1/armc-00 $./build.sh usage: build.sh <pi-model> pi-model options: rpi0, rpi1, rpi1bp, rpi2, rpi3, rpi3bp, rpi4 Since there are various compiler flags of different RPI models, it is necessary to tell the script what RPI you have in order to use the correct flags to form with. V1 panels are equipped with Broadcom BCM2835 (ARM1176) and the V2 board uses BCM2836 (ARM Cortex A7). RPI3 uses cortex-A53. Due to processor differences, we use different build commands to create different RPI models. Let's look at how the compiler command line looks like the appearance of different RPI models. Let's just focus on RPI-specific options rather than adding all the options here. RPI0 (PiZero) and RPI1¶ arm-none-eabi-gcc \ -mfloat-abi=hard \ -mfpu=vfp \ -march=armv6zk \ -mtune=arm1176jzf-s \ main.c -o main.elf RPI2¶ arm-none-eabi-gcc \ -moflat-abi=hard \ </pi-model>\ -march = armv7-a \ -mtune =cortex-a7 \ main.c -o main.elf RPI3¶ arm-none-eabi-gcc \ -mfloat-abi=hard \ -mfpu crypto=-neon-fp-armv8 \ -march=armv8-a+crc\ -mcpu=cortex-a53 \ main.c -o main.elf RPI4¶ arm-none-eabi-gcc \ -mfloat-abi=hard \ -mfpu=crypto-neon-fp-armv8 \ -march=armv8-a+crc \ -mcpu=cortex-a72 \ main.c -o main.elf IT IS EXPECTED TO FAIL: Using build script, we can collect the master code using RPI-specific options. Here we gather RPI3. (I shorten the output, making it easier to read on the screen). valvers-new/arm-tutorial-rpi/part-1/armc-00 $./build.sh rpi3 arm-none-eabi-gcc -mfloat-abi=hard \\mfpu=1crypto-neon -fp-armv8 \ -march =armv8-a+crc \ -mcpu=cortex-a53 \ armc-00.c \ -o kernel.armc-00.rpi3.elf GCC successfully compiles the code (it does not contain C errors), but the linker fails with the following message: .../hard/libc.a(lib_a-exit.o): In the exit function: exit.c:(.text.exit + 0x1c): An undefined reference to _exit collect2: Error: ld returned 1 exit status So with our one-line command above we call C compiler, collector and linker. C compiler makes most of the menial tasks for us to make life easier for us, but because we're embedded engineers (aren't we?) we need to know how compiler, collector and linker work at a very low level, as we usually work with custom systems, which we have described closely in the tool chain. So there is a missing symbol of _exit. This symbol is a reference to the C library that we use. It's actually a system call. It's designed to be implemented by the OS. In our case, we are on our own OS not we have the only thing running, and in fact we never go out so we don't really have to worry about it. System calls can be empty, they just have to be placed in order to linker to resolve the character. So library C has a system call requirement. Sometimes they're already implemented as empty features or installed for fixed features. For a list of system calls, see the newlib documentation for system calls. Newlib is an open source, and lightweight C library that can be composed of different flavors. C Library is what gives all the C features found in standard C header files such as stdio.h, stlib.h, string.h, etc. at this point I want to note that the standard Hello World example will not work here without the OS, and this is exactly an unimpl implemented system calls to prevent this being our first example. The lowest part of printf(...) contains the write function , this feature is used by all the functions in library C that need to be saved to the file. In printf's case, it must write to the stdout file. Typically, when an OS running stdout produces an output visible on the screen that can be piped to another file system file OS. In OS stdout usually prints to UART so that you can see the output of the program on the remote screen, for example, computer running the terminal application. We will discuss writing a series of outlets later in the tutorial, let's move on ... The easiest way to solve a link problem is to provide a minimum exit feature to satisfy the link. Since it will never be used, all we have to do is close the linker and let it solve _exit. So now, again with build.sh script armc-01 we can collect another version of the code, part-1/armc-01.c int main (void) { o (1) { } return 0; } void(int code) ( o (1) } ; NOTE: If you're wondering, the C compiler prefixes emphasize the functions of generated characters, so we don't have an underline, otherwise we end up with a feature that linker sees as __exit. If we write this collector file, we need to add an underscore prefix ourselves. As we can see, the compilation is successful and we get the kernel * .elf file generated by the compiler. Currently, the elf file is 37k. part-1/armc-01 $./build.sh rpi3 arm-no-eabi-gcc-mfloat-abi=hard \ -mfpu=crypto-neon-fp-armv8 \ -march=armv8-a+c -mcpu =cortex-a53 \ armc-01.c \ -o kernel.armc-01.rpi3.elf part-1/armc-01 $ls -lh total 16K-r 1 Brian Brian 366 Sep 21 00:19 armc-01.c -rwxr-xr-x 1 Brian Brian 2.3K September 21 00:00 4 build.sh -rwxr-xr-x 1 Brian 36K Sep 21 00:45 kernel.armc-01.rpi3.elf It is important to have an infinite loop output function. In library C, which is not intended for use with the operating system (and thus arm-NONE-eabi-*), _exit is marked as noreturn. We need to make sure he doesn't come back otherwise we'll get a warning about it. _exit prototype always includes an output code int too. Yes, it's a little oxymoronic! Now using the same build command above we get a clean build! Yay! But there is definitely a problem to give the system after the C library we will have to provide linker scripts and your C startup code. In order to miss that from the beginning and just get up and running we just use the GCC option to exclude any C startup routines that does not include the removal required too. GCC options that are -nostartfiles familiarity with the processor¶ As Cambridge tutorials we copied our original example to illuminate the LED in order to know that our code works correctly. It's almost always embedded creator of Hello World!. Usually we flash the LED to make sure the code works and know we're getting the clocks at the speed we think we should. Raspberry-Pi Boot Process¶ First, let's look at how raspberry-pi processor boots. BCM2385 from Broadcom includes two processors that we need to know about, one videocore™ GPU, which is why Raspberry-Pi makes such a good media center, and the other arm core that runs the operating system. Both of these processors share a peripheral bus and must also share certain intermittent resources. Although part of the case means that some interruptible sources do not ARM processor because they've taken the GPU. The GPU restarts or is enabled and includes code read on the first FAT disk SD card on the MMC bus. It searches and loads a file that is named bootcode.bin into memory and starts executing this code. The Bootcode.bin bootloader in turn searches for an SD card file named start.elf and config.txt file to set various kernel settings before searching for the SD card again in the kernel.img file, which it then loads into memory at a specific address (0x8000) and starts the ARM processor executing in that memory location. The GPU is now up and running and ARM will start to come up with the code contained in kernel.img. Start.elf file contains code that runs on the GPU to provide most OpenGL requirements, etc. So in order to run your code, you must first collect your code in the executable file and name it kernel.img and put it on the FAT formatted SD card, which has a GPU bootloader (bootcode.bin, and start.elf) it as well. The latest Raspberry-Pi software can be found on GitHub. The boot loader is located in the boot pack directory. All other software provided is closed binary video drivers. They are compiled for use under Linux that speed up graphics drivers there. Since we do not use Linux these files are useless to us, only bootloader firmware is available. All this means that the processor already exists and works when it starts to execute our code. Clock sources and PLL settings are already decided and programmed bootloader, which eases this problem from us. We just start messing with device registryes from an already running core. This is something I haven't used too much, usually the first thing my code would create the correct clock and PLL settings to initiate the processor, but the GPU has configured the basic clocking scheme for us. The first thing we will have to set up is the GPIO controller. There are no drivers we can rely on because there is no OS running, all the bootloader did is run the processor into working state, ready to start loading the OS. You need to get raspberry-Pi BCM2835 peripherals datasheet, and make sure to pay attention to errors that too, because it is not perfect. This gives us the information we need to manage BCM2835 IO peripherals. I'll guide us using GPIO Peripheral – there are always some gotcha's: Raspberry-Pi 2B 1.2 uses BMC2837 and so you want to get raspberry-Pi BCM2837 on the edge of the datasheet. Note: The 2837 Peripherals document is only a modified version of the original 2835 document with addresses updated to suit the 2837 primary peripheral address. For more information, see rpi issue 325. We will use the GPIO peripheral, so it would be natural to go directly to that documentation and start writing the code, but first we need to read some basic information about It is important to slightly tick the information in the virtual address. On page 5 of the BCM2835 Peripherals page, we see a map of the processor IO. Again, as embedded engineers we need to have an IO map to know how to deal with peripherals processors and in some cases how to organize our linker scenarios when there are many address spaces. VC CPU bus addresses are related to broadcom video core CPU. While video core CPU is what bootloads from SD cards, execution is passed to the ARM core by the time our kernel.img code is called. So we are not interested in vc CPU bus addresses. ARM physical addresses are processors in a green IO map when the ARM Memory Control Unit (MMU) is not used. If MMU is used, virtual address space is something we would be interested in. MMU also does not work before the OS kernel is running because it was not initialized and the kernel is running in kernel mode. Therefore, addresses on the bus are accessible via their ARM physical address. We can see from the IO map that the VC CPU address 0x7E000000 is associated with arm physical address 0x20000000 original Raspberry Pi. This is important! If you carefully read the two peripheral data sheets, you will see a subtle difference between them, especially , Raspberry-Pi 2 has an ARM IO base set at 0x3F000000 instead of the original 0x200000000 original Raspberry-Pi. Unfortunately, for us software engineers the Raspberry-Pi Foundation doesn't seem good at securing the documents we need, in fact, their approach shows that they think we're magicians and don't really need any. Please, if you are a forum member, campaign for more documents. As engineers, especially in the industry we can't take this from the manufacturer, we want to go elsewhere! In fact, we did our job and use the TI Cortex A8 from Beaglebone Black, a very good and well documented SoC! In any case, the main address can be collected from the search for uboot patches. Raspberry Pi 2 uses BCM2836 so we can look for that and you boot and we come along patch to support Raspberry-Pi 2. In the guide below, we are dealing with the GPIO Peripheral Section Manual (Chapter 6, page 89). RPi4 has a peripheral base associated with 0xFE000000. The peripheral address space looks arranged the same as the previous pis. Visual Output and running code¶ Finally, let's get on and see some of our code running raspberry-pi. We will continue to use the first example of Cambridge tutorials lighting OK LED on raspberry-pi board. GPIO peripheral has the primary address in the BCM2835 manual 0x7E200000. We know to get to know our processor that this translates to ARM physical address 0x20200000 (0x3F200000 RPI2 and RPI3, and 0xFE200000 RPI4). This is the first registry of the GPIO set of peripheral registers, GPIO function select 0 registry. To use the IO pin, we need to configure gpio peripherals. Raspberry-Pi schematic diagrams OK LED is a wired gpio16 line (sheet 2, B5). LED wired active LOW - this is quite common practice. This means that to enable the LED we have to output 0 (the pin is connected to the 0V processor) and disable it we output 1 (the pin is connected to the VDD processor). Unfortunately, again, lack of documentation is prevalent and we do not have schemes raspberry-Pi 2 or plus models! This is important because the GPIO line has been re-jigged and as Florin noted in the comments section, the Raspberry Pi Plus configuration has a GPIO47 LED, so I added brackets below rpi B+ models changes (which includes RPI 2). Back to the CPU guide and we see that the first thing we need to do is set the GPIO pin to the output. This is done by setting the GPIO16 (GPIO47 RPI+) function to the output. Bits 18 to 20 GPIO function select 1 record control GPIO16 pin. Bits 21 to 23 GPIO function Select 4 register control GPIO47 pin. (RPI B+) Bits 27 to 29 GPIO function Select 2 register control GPIO29 pin. (RPI3 B+) GPIO42 pin. (RPI4) (Article 17(1)) C, we will create an index in the registry and use the index to save the value to the registry. We will mark the registry as volatile, so that the compiler makes clear what I say. If we do not mark the registry as volatile, the compiler is free to see that we do not have access to this registry again and thus for all intentions and purposes the data that we write will not be used in the program and optimization freely throw away write, because it has no effect. However, the effect is really necessary, but only externally visible (GPIO pin mode changes). We inform the compiler through volatile keywords not to take anything for granted on this variable and just do as I say with it: We will use pre-processor definitions to change the base address of GPIO peripheral depending on what RPI model is designed for. #if defined( RPI0 ) || defined ( RPI1 ) #define GPIO_BASE 0x20200000UL #elif defined( RPI2 ) || defined ( RPI3 ) #define GPIO_BASE 0x3F200000UL #elif defined( RPI4) #define GPIO_BASE 0xFE200000UL #else #error Unknown RPI model! #endif In order to determine GPIO16 as output, then we need to write 1 value in the corresponding function bits select registry. Here we can rely on the fact that this registry is set to 0 after restart and therefore all we need to do is set: /* Assign GPIO peripheral address (Using ARM physical address) * / gpio = (unsigned int*)GPIO_BASE; gpio[LED_GPFSEL] |= (1 << LED_GPFBIT); This code looks a bit embarrassing, but we'll tidy up and optimize later. For now we just want to get to the point where we can ignite the LED and understand why it burns! ARM GPIO peripherals have an interesting way to do IO. This is actually a little different from many other CPU IO installations. There is a SET registry and a CLEAR registry. Writing from 1 to bits in the SET registry will set the corresponding GPIO pins 1 (logic large), and writing 1 but bits in the CLEAR registry will clear the corresponding GPIO pins 0 (logic low). There are reasons for this implementation through the registry, where each bit is pin and bit value directly related to the pins output level, but that's the scope for this tutorial. So, in order to illuminate the LED, we need to output 0. We need to write 1 bit 16 in the CLEAR registry: Putting what we learned in the minimal example above gives us a program that compiles and links to an executable file that should give us raspberry-pi that lights OK LED when it is powered. Here's the full code we dial part-1/armc-02 /* Primary address GPIO peripheral (ARM physical address) */ #if defined( RPI0 ) || defined ( RPI1 ) #define GPIO_BASE 0x20200000UL #elif defined( RPI2 ) || defined( RPI3) #define GPIO_BASE 0x3F200000UL #elif defined( RPI4 ) / * This comes from a Linux source code: */ #define GPIO_BASE 0xFE200000UL #else #error Unknown RPI model! #endif /* TODO: Expand this area to RPi4 if necessary */ #if defined( RPIBPLUS) || defined ( RPI2 ) 21 #define LED_GPFSEL GPIO_GPFSEL4 #define LED_GPFBIT 15 #define LED_GPSET GPIO_GPSET1 #define LED_GPCLR GPIO_GPCLR1 #define LED_GPIO_BIT #elif defined ( RPI4 ) /* RPi4 model has a GPIO 42 */ #define LED_GPFSEL GPIO_GPFSEL4 #define LED_GPFBIT 6 #define LED_GPSET GPIO_GPSET1 #define LED_GPCLR GPIO_GPCLR1 #define LED_GPIO_BIT 10 #else #define LED_GPFSEL GPIO_GPFSEL1 #define LED_GPFBIT 18 #define LED_GPSET GPIO_GPSET0 #define LED_GPCLR GPIO_GPCLR0 #define LED_GPIO_BIT 16 #endif #define GPIO_GPFSEL0 0 #define GPIO_GPFSEL1 #endif #define GPIO_GPFSEL0 APS LED 2 #define GPIO_GPFSEL2 2 #define GPIO_GPFSEL3 #define GPIO_GPFSEL2 2 #define GPIO_GPFSEL4 4 #define GPIO_GPFSEL5 5 #define GPIO_GPSET0 7 #define GPIO_GPSET1 8 #define GPIO_GPCLR0 #define GPIO_GPCLR1 10 #define GPIO_GPCLR1 13 #define GPIO_GPLEV0 13 #define GPIO_GPLEV1 14 #define GPIO_GPEDS0 17 #define GPIO_GPREN0 17 #define GPIO_GPREN0 16 #define GPIO_GPEDS1 17 #define GPIO_GPREN0 20 #define GPIO_GPREN1 #define GPIO_GPFEN0 20 #define GPIO_GPFEN0 20 #define GPIO_GPFEN1 #define GPIO_GPFEN0 25 #define GPIO_GPHEN0 25 #define GPIO_GPHEN1 26 #define GPIO_GPLEN0 28 #define GPIO_GPLEN1 29 #define GPIO_GPAREN0 31 #define GPIO_GPAREN1 32 32 #define GPIO_GPAFEN0 34 #define GPIO_GPAFEN1 35 #define GPIO_GPPUD 37 #define GPIO_GPPUDCLK0 38 #define GPIO_GPPUDCLK1 39 /** GPIO registry set */ non-registrable unsigned gpio; /** Simple loop variable */ volatile unsigned int tim; /** Main function - we never come back from here * / int main (void) __attribute__ ((naked)); int main(void) { /* Assign GPIO peripheral address (using ARM physical address) */ gpio = GPIO_BASE; /* Write 1 to GPIO16 init nibble function Select 1 GPIO peripheral registry to enable GPIO16 as output */ gpio [LED_GPFSEL] |= (1 (1 (1 LED_GPFBIT); /* Never go out because there is no OS to go out to! */ o (1) { for (tim = 0; tim < 500000; tim++) , /* Set up a small LED GPIO pin (Turn on the original Pi, and turn off plus for models )*/ gpio[LED_GPCLR] = (1 << LED_GPIO_BIT << LED_GPIO_BIT LED_GPSET <); Now we've assembled with a non-start files option as well: part-1/armc-02 $./build.sh rpi3 arm-no-eabi-gcc -nostartfiles \ -mfloat-abi = hard \ -mfpu = crypto-neon-fp-arm \ -march=armv8-a+crc \ -mcpu=cortex-a53 \ armc-02.c \ -o kernel.armc-02.rpi3.elf Linker gives us a warning, which we will handle later, but the most important linker solved the problem for us. This is a warning that we will see and ignore: .../arm-none-eabi/bin/ld: warning: you cannot find the character of the record _start; default up to 00000000008000, as we can see from the compilation, the standard output is elf format, which is basically an executive wrapped with information that the OS binary loader may need to know. We need a binary ARM executable file that only includes computer code. We can extract this using objcopy utility: arm-none-eabi-objcopy kernel.elf-O binary kernel.img Quick note on ELF format¶ ELF is a file format used in some OS, including Linux, which wraps computer code with meta-data. Metadata can be useful. Linux and indeed most OS these days, running executable does not mean that the file will be loaded into memory and then the processor starts running from the address at which the file was uploaded. Typically, there is an executable loader that uses formats such as ELF to learn more about executable, such as functional call interfaces that may vary between different executables, meaning that the code can use different calling conventions that use different registryes for different values when you call features within the program. This can determine whether the executable loader can even allow the program to load into memory or not. If any of the required libraries are not available, the executable loader will not allow the file to be loaded and started again. All this is designed (and does) to increase the stability and compatibility of the system. However, we do not have an OS and the boot loader does not have any loader other than disk reading, directly copying the kernel.img file to memory 0x8000, which is when the ARM processor starts to execute the computer code. Therefore, we need to strip disable ELF metadata and just leave only the collected machine code kernel.img file ready for execution. Back to our example¶ This gives us a kernel.img binary file that should contain only ARM computer code. It should be tens of bytes long. You will notice that kernel.elf with the other hand is ~ 34Kb. Rename your SD card is something like old.kernel.img and save the new kernel.img to the SD card. Normal start-up is ok LED must be switched on, then extinguish. If it remains extinguished something happened to the building or link your program. Otherwise, if the led remains enabled, your application is successfully executed. The flashing LED is probably more suitable to make sure our code really works. Let's quickly change the code to a roughly blink LED and then we'll look at sorting the C library issues we've had before, as the C Library is too useful to have no access to it. Compile the code in part 1/armc-03. The code list is identical to Part 1/armc-02, but create scripts to use objcopy to convert ELF formatted binary to green binary ready to deploy sd card. You can see that the binary image is now in the folder and there is a lot of healthy size for some code, who does so little: part-1/armc-03 $./build.sh rpi3bp arm-none-eabi-gcc -g \ -nostartfiles \ -mfloat-abi=hard \ -O0 \ -DIOBPLUS \ -DRPI3 \ -mfpu =crypto-neon-fp-armv8 \ -march=armv8-a+crc \ -mcpu=cortex-a53 \ armc-03.c \ -o kernel.armc-03.rpi3bp.elf .../arm-none-eabi/bin/ld: Warning: you cannot find the entry character _start; default 0000000000000000 arm-none-eabi-objcopy kernel.armc-03.rpi3bp.elf-O binary kernel.img Kernel.img file is only arm machine code and so there are only a few hundred bytes. part-1/armc-03 $ll total 28 drwxr-xr-x 2 Brian Brian 4096 Sep 21 01:05 ./ drwxr-xr-x 6 Brian Brian 4096 Sep 21 00:19 .. / -rw-r--r-- 1 Brian Brian 3894 Sep 21 00:19 armc-03.c -rwxr-xr-x 1 Brian 2805 Sep 21 01: 05 build.sh* -rwxr-xr-x 1 brian 16777216 Sep 21 01:05 card.armc-03.rpi3bp.img -rwxr-xr-x 1 brian brian 35208 Sep 21 01:05 kernel.armc-03.rpi3bp.elf* -rwxr-xr-x 1 Brian 268 Sep 21 21 S01:05 kernel.armc-03.rpi3bp.img* SD card generation¶ The next step is how to get this kernel.img file to an SD card, to start the card and run the newly collected code. A script called make_card.sh was run build.sh script execution. Take a look at the build.sh script to see the call near the end of the script. It does the job of generating an image of an SD card that can be saved directly to an SD card. /card/make_card.sh script is worthy of appearance. It can generate a card image without the need for user super user priveleges. It always uses the latest firmware available from the RPi Foundation GitHub Repository Writing image on the card can be done using a cat until you know what an SD card device is. If you want, you can use the write_card.sh script in the card directory, which you can use to interactively select an SD card. If you want to do things manually, you can put an SD card and then run dmesg | tail view messages that will show which device link was used sd card or else use lsblk list all block DON'T GET AN SD CARD WRONG OR YOU COMPLETELY DESTROY ANOTHER DRIVE! Once you know what drive to use, you can just cat the image to disk using cat kernel.armc-03.rpibp.img > / dev /sdg for example. I've included kernel binaries for each Raspberry-Pi board so you can load a pre-built binary and see an LED flash before compiling your kernel to make sure the development process works for you. After this tutorial, you have to create your own binary files! While the code may seem written a bit weird, please stick with it! There are reasons why it is written as it is. Now you can experiment a little from the main starting point, but beware - automatic variables will not work, nor have variables entered, because we do not yet have C run time support. This will be when we start with Step 2 Bare metal programming Raspberry-Pi! Raspberry-Pi!