Czech University of Life Sciences, Prague

Department of Information Engineering

Faculty of Economics and Management

Master thesis

Real-time application support on the Raspberry PI platform

Analysis of Real Time Linux performance on BeagleBoneBlack board and demonstrate applications

Author **Thesis Supervisor**

Mina Samaan **Doc. Ing. Vojtěch Merunka, Ph.D.**

©MinaSamaan@Prague2014

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Department of Information Engineering

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Saber Samaan Mina Nabil

Informatics

Thesis title

Real-time application support in the Raspberry Pi platform

Objectives of thesis

The goal of thesis is an application of the technology of microcomputer technology Raspberry Pi. A proper OS for this platform will be selected, prefferably HelenOS or Minus or similar. The application will be analysed, designed and implemented as a real-time application, which can demonstrate the usage of this technology in farming area farming.

Methodology

C-programming, hardware installing, open-source community collaboration

Schedule for processing

03/2013 - 09/2013 State of the art description, collect knowledge and tools.
10/2013 - 12/2013 The implementation
01/2014 - 03/2014 Thesis completing and writing.

The proposed extent of the thesis
50-80 pages

Keywords
computer hardware, real-time applications, microcomputers, special OS, Raspberry Pi

Recommended information sources

C: The Complete Reference, 4th Ed.Herbert Schildt | ISBN-13: 978-0072121247

Real-Time Concepts for Embedded Systems ISBN:1578201241

Linux for Embedded and Real-Time Applications ISBN 0-7506-7932-8

The Diploma Thesis Supervisor
Merunka Vojtěch, doc. Ing., Ph.D.

Last date for the submission
March 2014

Electronic approval: October 30. 2013

Ing. Martin Pelikán, Ph.D.
Head of the Department

Electronic apporoval: December 5. 2013

Ing. Martin Pelikán, Ph.D.
Dean

Declaration

I declare that I have worked on my diploma thesis titled "Real-time application support on the Raspberry PI platform" subtitled "Analysis of Real Time Linux performance on BeagleBoneBlack board VS demonstrate applications" by myself and I have used only the sources mentioned at the end of the thesis.

……………………………………………………………

Acknowledgement

First of all, I would like to express my sincere gratitude to the diplomatic corporation Between the Government of Czech Republic and Egypt that has offered me the Opportunity to pursue post graduate studies. I would like also to thank the entire members of the teaching and administrative staff of the Faculty of Economics and Management (PEF), Czech University of Life Sciences. Special appreciation for the officers of International relation office in PEF for their time providing good information and material needs for my studies. I am most grateful to Doc. Ing. Vojtěch Merunka, Ph.D. who was my lecturer and supervisor for my Diploma Thesis. I am grateful for his advice and direction in helping me finish this diploma writing.

Finally, my biggest thanks to my family and my friends who have encouraged me so much and special thanks to my soul mate Mary who helped me going through all of that. My entire study program would undoubtedly not have been a success in the absence of all of you.

Whoever does not love does not know God, because God is love. 1 John 4:8

Abstract

The market for cheap single-board computers is becoming one of the most surprisingly competitive spaces in the tech industry. On the heels of the million-selling Raspberry Pi [4], a variety of companies and small groups started creating their own tiny computers for programmers and hobbyists.

Now we have a new entrant that may provide the best bang for the buck for many types of users. It's called the BeagleBoneBlack and it's the latest in the line of "Beagle" devices that first appeared in 2008, courtesy of Texas Instruments. On sale now for $45, BeagleBoneBlack sports a 1GHz Sitara AM335x ARM Cortex-A8 processor from Texas Instruments [5].

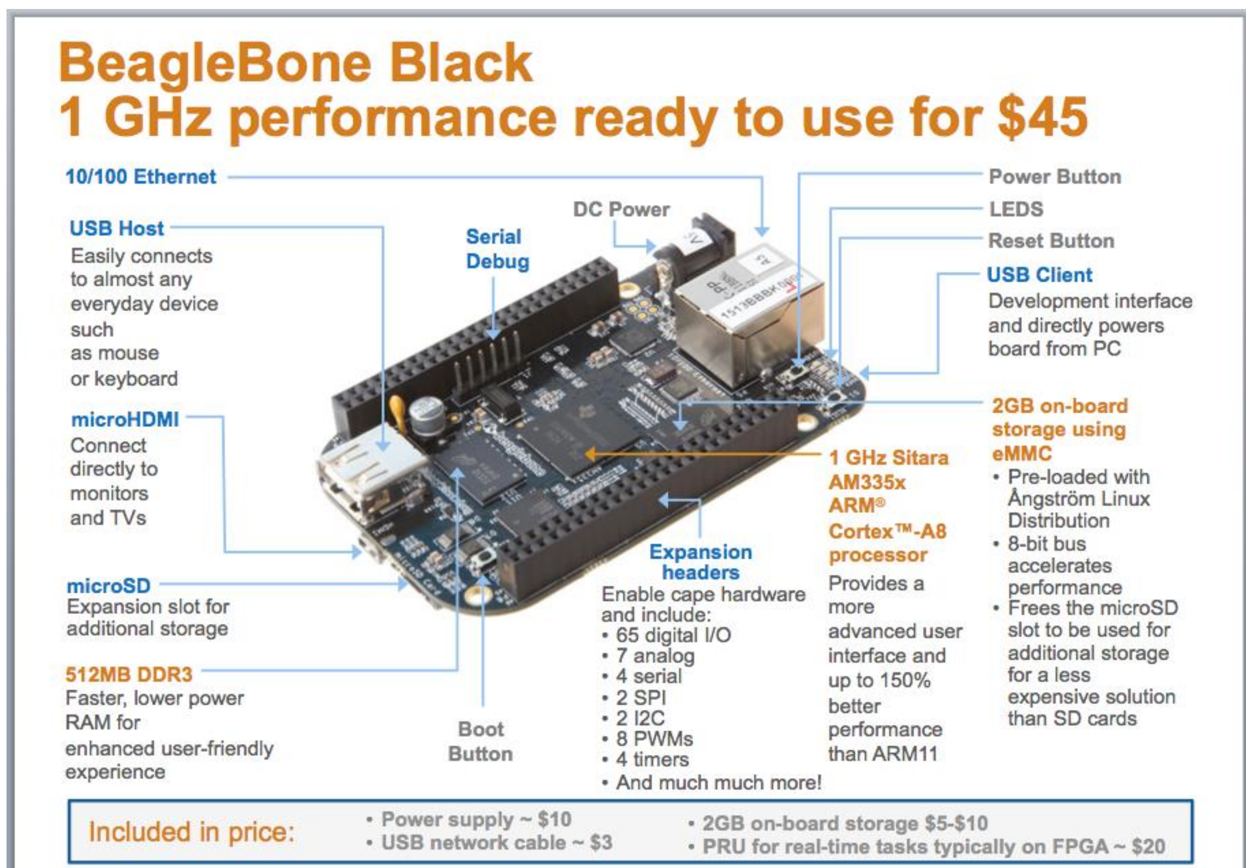


Figure 1 – BeagleBoneBlack layout

The release of this cheap computer opens the door for many applications that were very hard to implement before. Imaging doing home automation or agriculture applications like auto collect samples from soil by distributed sensors connected to your tiny computer or auto control the environment of greenhouse farm.

But this tiny computer requires an operating system to operate properly and this impose a difficulty on using them especially if you have a complex application.

The board comes already preinstalled with Linux, but what if your application needs real time requirements.

Linux has much strength as an operating system. Yet those developing real-time systems with Linux must also overcome a number of problems with the standard Linux components, including:[7]

- Limited number of fixed priority levels
- No support for priority inheritance
- Limited QoS (Quality of Service) support
- Lack of support for high-resolution timers
- No support for periodic tasks
- Potentially non-preemptible kernel with possibly long system calls
- Limited support for non-desktop systems

A real-time system is one that provides guaranteed system response times for events and transactions—that is, every operation is expected to be completed within a certain rigid time period. A system is classified as hard real-time if missed deadlines lead to system failure and soft real-time if the system can tolerate some missed time.[8]

In this thesis, I will analysis the performance of real time Linux performance and compare it to commercial RTOS available in market; at the end I will demonstrate an application of controlling greenhouse farm using BeagleBoneBlack board running Linux. Our solution will be much cheaper than commercial solutions that cost thousands of dollars because we used off the shelf components and free software, so the cost is minimal with the same application.

One important advantage of our system is that the board hosts an Ethernet Port which can be used to access the board over network connection and monitor all the peripherals connected to it and control them. This gives us the advantage to control our application from a remote site as if we are in the same location as the board, that would save us a lot of time and money to move every time to the field to change configurations or give the computer new instructions. This is a demonstration how a connected world can make our life easier to maintain and getting ready for the internet of things.

There are dozens of other applications can be done by the cheap computer boards, and the integration of the board and the availability of free software for it, makes it suitable even for non-professionals to use it with ease.

Also, in the twenty first century, the world moves towards more secure computing and Linux put security among its priorities and because the board runs Linux, it can be used as an entry educational platform for beginners who want to shift their skills from closed source operating systems like windows to Linux for more freedom. Linux can also be found within many

consumer electronics devices. Whether they're inside a cell phone, cable box, or exercise bike, embedded Linux systems blur the definition between computer and device.

Thesis Roadmap:

- Building Linux Kernel for BeagleBoneBlack after applying RT patch to make the kernel Real-time.
- Run workbench tests on the platform.
- Demonstrate some applications that are important in the agriculture field and require Real-time performance.

Note to the reader

Throughout this thesis, I intended to walk through the building process of a complete Linux Kernel for a specific platform and I explained step by step in detailed descriptions. I hope my thesis would be a good reference for anyone who might be interested in using Linux in any embedded system project. This thesis is a demonstration to encourage more adoption of free software in future Projects.

Table of content

1-Introduction…………………………………………………………………………………………..1

1-1-Processors in embedded systems ………………………………………………………………1

1-1-1-System on a chip………………………………………………………………………………2

1-1-2-Structure………………………………………………………………………………………..2

1-2-Embedded operating system………………………………………………………………...3

1-2-1-The Kernel……………………………………………………………………………………..4

1-2-2-Microkernel……………………………………………………………………………………5

1-2-2-Monolithic kernel……………………………………………………………………………..6

1-3-Why Linux? 6

1-3-1-GPL ……………………………………………………………………………………………7

1-3-2-Standards Based ……………………………………………………………………………...8

1-3-3-Process Isolation and Control ……………………………………………………………….8

1-3-4-Manage and Isolate Tasks …………………………………………………………………...9

1-3-5-Peripheral Support ……………………………………………………………………………9

1-3-6-Security ………………………………………………………………………………………10

1-4-Single-board computer ……………………………………………………………………..10

1-4-1-Raspberry Pi …………………………………………………………………………………10

1-4-1-1-About …………………………………………………………………………………………11

1-4-2-BeagleBoneBlack …………………………………………………………………………..12

1-4-3-Why we Use BeagleBoneBlack over the Raspberry PI? ……………………………...……13

1-4-4-Ångström distribution ………………………………………………………………………15

1-5-Linux and Real-time …………………………………………………………………………15

1-5-1-Soft Real-time ………………………………………………………………………………15

1-5-2-Hard Real-time ……………………………………………………………………………..15

1-5-3-Linux scheduling ……………………………………………………………………………16

1-5-4-Latency ……………………………………………………………………………16
1-5-5-Kernel preemption ……………………………………………………………………17
1-5-6-Real-time implementation in Linux ………………………………………………18
2-Building the Linux Kernel ………………………………………………………………19
2-1-Bios Vs bootloader ………………………………………………………………………19
2-2-Anatomy of BBB ……………………………………………………………………………19
2-2-1-Processor …………………………………………………………………………………20
2-2-2-DDR3L Memory ……………………………………………………………………………20
2-2-3- On Board Flash Memory …………………………………………………………………20
2-3-Typical embedded Linux setup ……………………………………………………………21
2-4-The building Process ………………………………………………………………………22
2-4-1-Installing the cross compiler & GIT ………………………………………………22
2-4-2-Getting the sources ……………………………………………………………………23
2-4-3-Configuring the build script …………………………………………………………24
2-4-4-applying the patches and configuring the kernel ……………………………25
2-4-4-1-RealTime Kernel Performance analysis ……………………………………………27
2-4-5-Compiling the kernel ……………………………………………………………………30
2-5-Deploying the new kernel …………………………………………………………………30
2-6-Logging to the Board over Secure shell SSH ………………………………………31
2-7-Benchmarking the Kernel ……………………………………………………………………32
2-7-1-FTRACE results ……………………………………………………………………………32
2-8-Comparing Linux vs. commercial RTOS ……………………………………………………33
2-8-1-VxWorks ………………………………………………………………………………………33
2-8-2-Windows CE .NET ……………………………………………………………………………33
2-8-3-QNX Neutrino RTOS …………………………………………………………………………33
2-9-Conclusion ……………………………………………………………………………………33

3-Case study Project: Farm Automation System ……………………………………………………………34
3-1-Irrigation ……………………………………………………………………………………………34
3-2-System prototype ……………………………………………………………………………………35
3-3-Component cost ……………………………………………………………………………………37
3-3-1- 5V Stepper Motor 28BYJ-48 with Drive Test Module Board ULN2003 5 Line 4 Phase ..37
3-3-2- TMP36 Temperature Sensor ………………………………………………………………………37
3-3-3- Soil Moisture Sensor …………………………………………………………………………38
3-3-4-The BeagleBoneBlack ………………………………………………………………………………38
3-4-Setting up IO Python Library on BeagleBoneBlack ……………………………………………39
3-5-Connecting TMP36 Temperature Sensor ……………………………………………………………40
3-5-1-Tools needed ……………………………………………………………………………………41
3-5-2-Wiring ………………………………………………………………………………………………41
3-5-3-Writing a program to read temperature ……………………………………………………42
3-6-Connecting the stepper motor ……………………………………………………………………44
3-6-1-Wiring Diagram ……………………………………………………………………………………44
3-6-2-Step sequences ……………………………………………………………………………………46
3-6-3-Python code …………………………………………………………………………………………46
3-7- Connecting the Soil Moisture Sensor …………………………………………………………48
3-7-1-Interface description ……………………………………………………………………………48
3-7-2- Instructions for use ……………………………………………………………………………49
3-7-3-Python code …………………………………………………………………………………………50
3-8-The final scenario ………………………………………………………………………………51
Conclusion ………………………………………………………………………………………………51
References ………………………………………………………………………………………………52
List of figures …………………………………………………………………………………………54

1-Introduction

An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today.[9][10]

Modern embedded systems are often based on microcontrollers (i.e CPUs with integrated memory and/or peripheral interfaces)[4] but ordinary microprocessors (using external chips for memory and peripheral interface circuits) are also still common, especially in more complex systems. In either case, the processor(s) used may be types ranging from rather general purpose to very specialized in certain class of computations, or even custom designed for the application at hand. A common standard class of dedicated processors is the digital signal processor (DSP).

The key characteristic, however, is being dedicated to handle a particular task. Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance. Some embedded systems are mass-produced, benefiting from economies of scale.

Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, and largely complex systems like hybrid vehicles, MRI, and avionics. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

1-1-Processors in embedded systems

Embedded processors can be broken into two broad categories. Ordinary microprocessors (μP) use separate integrated circuits for memory and peripherals. Microcontrollers (μC) have many more peripherals on chip, reducing power consumption, size and cost. In contrast to the personal computer market, many different basic CPU architectures are used, since software is custom-developed for an application and is not a commodity product installed by the end user. Both Von Neumann as well as various degrees of Harvard architectures is used. RISC as well as non-RISC processors are found. Word lengths vary from 4-bit to 64-bits and beyond, although the most typical remain 8/16-bit. Most architecture comes in a large number of different variants and shapes, many of which are also manufactured by several different companies.

Numerous microcontrollers have been developed for embedded systems use. General-purpose microprocessors are also used in embedded systems, but generally require more support circuitry than microcontrollers.

1-1-1-System on a chip

A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions—all on a single chip substrate. A typical application is in the area of embedded systems.

The contrast with a microcontroller is one of degree. Microcontrollers typically have under 100 kB of RAM (often just a few kilobytes) and often really are single-chip-systems, whereas the term SoC is typically used for more powerful processors, capable of running software such as the desktop versions of Windows and Linux, which need external memory chips (flash, RAM) to be useful, and which are used with various external peripherals. In short, for larger systems, the term system on a chip is a hyperbole, indicating technical direction more than reality: increasing chip integration to reduce manufacturing costs and to enable smaller systems. Many interesting systems are too complex to fit on just one chip built with a process optimized for just one of the system's tasks.

1-1-2-Structure

A typical SoC consists of:

- A microcontroller, microprocessor or DSP core(s). Some SoCs—called multiprocessor system on chip (MPSoC)—include more than one processor core.
- Memory blocks including a selection of ROM, RAM, EEPROM and flash memory.
- Timing sources including oscillators and phase-locked loops.
- Peripherals including counter-timers, real-time timers and power-on reset generators.
- External interfaces including industry standards such as USB, FireWire, Ethernet, USART, SPI.
- Analog interfaces including ADCs and DACs.
- Voltage regulators and power management circuits.

These blocks are connected by either a proprietary or industry-standard bus such as the AMBA bus from ARM Holdings. DMA controllers route data directly between external interfaces and memory, bypassing the processor core and thereby increasing the data throughput of the SoC.

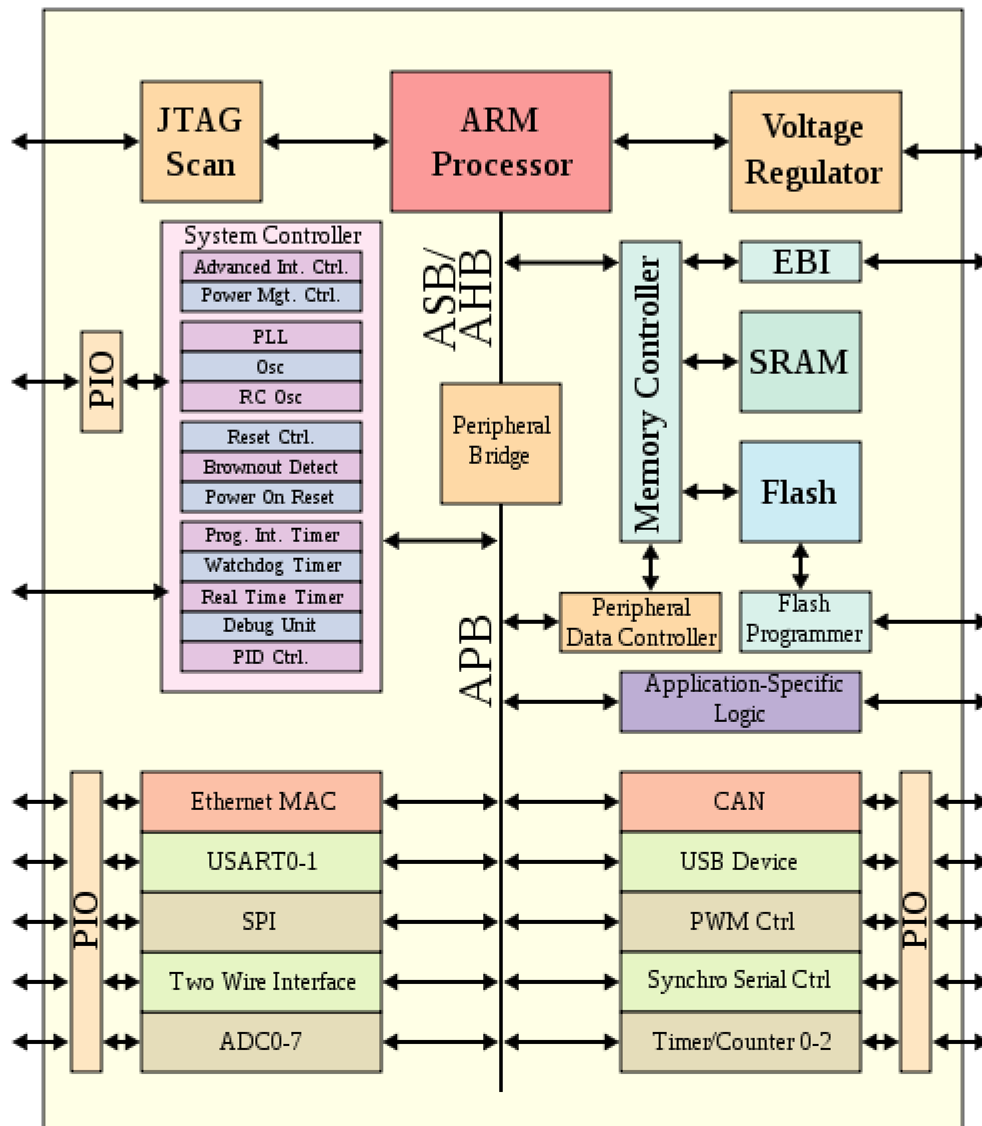


Figure 2 - ARM SOC Architecture

1-2-Embedded operating system

An embedded operating system is an operating system for embedded computer systems. These operating systems are designed to be compact, efficient at resource usage, and reliable, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run. They are frequently also referred to asreal-time operating systems, and the term RTOS is often used as a synonym for embedded operating system.[11]

Usually, the hardware running an embedded operating system is very limited in resources such as RAM and ROM therefore systems made for embedded hardware tend to be very specific, which means that due to the available resources (low if compared to non-embedded systems)

these systems are created to cover specific tasks or scopes. In order to get advantage of the processing power of the main (or only) CPU, system creators often write them in assembly. This machine efficient language "squeezes" the potentiality in terms of speed and determinism, which means maximizing the responsiveness of the operating system. Though, it not an absolute rule that all embedded operating systems are written in assembly language, as many of them are written in more portable languages, like C.

An important difference between most embedded operating systems and desktop operating systems is that the application, including the operating system, is usually statically linked together into a single executable image. Unlike a desktop operating system, the embedded operating system does not load and execute applications.[11] This means that the system is only able to run a single application.

1-2-1-The Kernel

In computing, the kernel is a computer program that manages input/output requests from software and translates them into data processing instructions for the central processing unit and other electronic components of a computer. The kernel is a fundamental part of a modern computer's operating system.[12]

When a computer program (in this case called a process) makes requests of the kernel, the request is called a system call. Various kernel designs differ in how they manage system calls (time-sharing) and resources. For example, a monolithic kernel executes all the operating system instructions in the same address space to improve the performance of the system. A microkernel runs most of the operating system's background process in user space, to make the operating system more modular and, therefore, easier to maintain.

For computer programmers, the kernel's interface is a low-level abstraction layer.

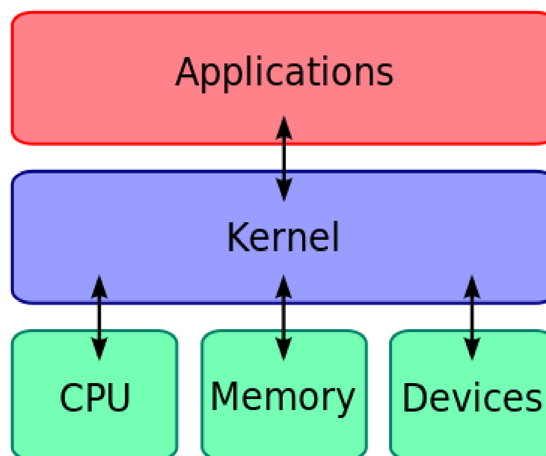


Figure 3 - Kernel_Layout

1-2-2-Microkernel

In computer science, a microkernel (also known as μ-kernel or Samuel kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). If the hardware provides multiple rings or CPU modes, the microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode).[14] Traditional operating system functions, such as device drivers, protocol stacks and file systems, are removed from the microkernel to run in user space. In source code size, microkernels tend to be under 10,000 lines of code, as a general rule. MINIX's kernel, for example has fewer than 6,000 lines of code.[13]

Microkernels were developed in the 1980s as a response to changes in the computer world, and to several challenges adapting existing "mono-kernels" to these new systems. New device drivers, protocol stacks, file systems and other low-level systems were being developed all the time. This code was normally located in the monolithic kernel, and thus required considerable work and careful code management to work on. Microkernels were developed with the idea that all of these services would be implemented as user-space programs, like any other, allowing them to be worked on monolithically and started and stopped like any other program. This would not only allow these services to be more easily worked on, but also separated the kernel code to allow it to be finely tuned without worrying about unintended side effects. Moreover, it would allow entirely new operating systems to be "built up" on a common core, aiding OS research.

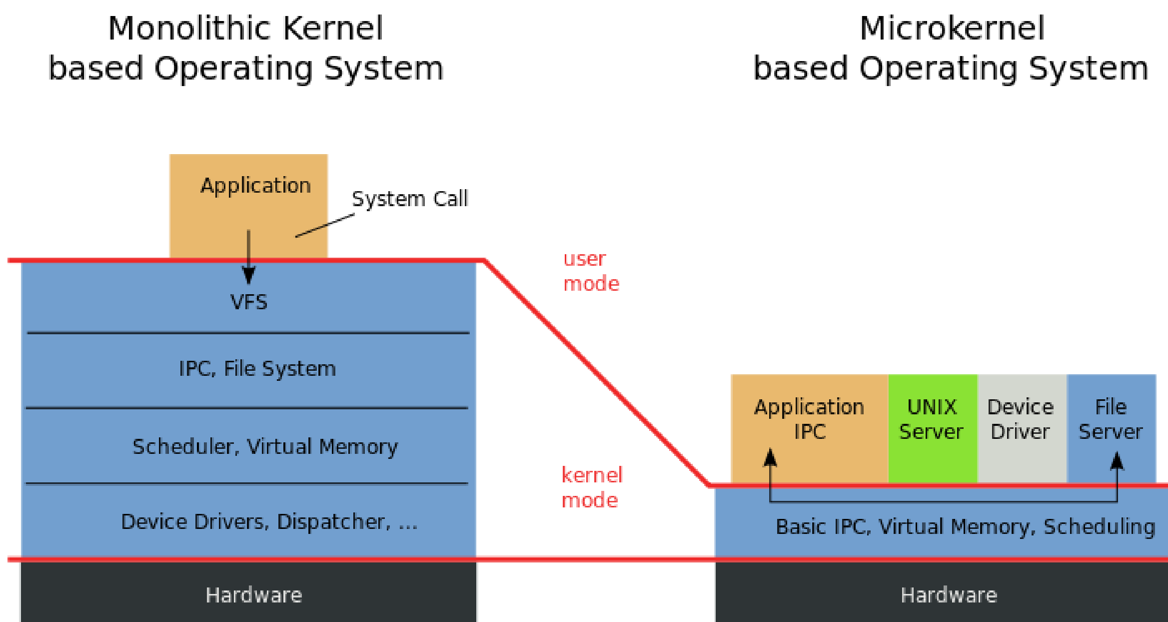


Figure 4 –Monolithic Kernel Vs Microkernel

1-2-2-Monolithic kernel

A monolithic kernel is an operating system architecture where the entire operating system is working in kernel space and is alone in supervisor mode. The monolithic model differs from other operating system architectures (such as the microkernel architecture) in that it alone defines a high-level virtual interface over computer hardware. A set of primitives or system calls implement all operating system services such as process management, concurrency, and memory management. Device drivers can be added to the kernel as modules. [2]

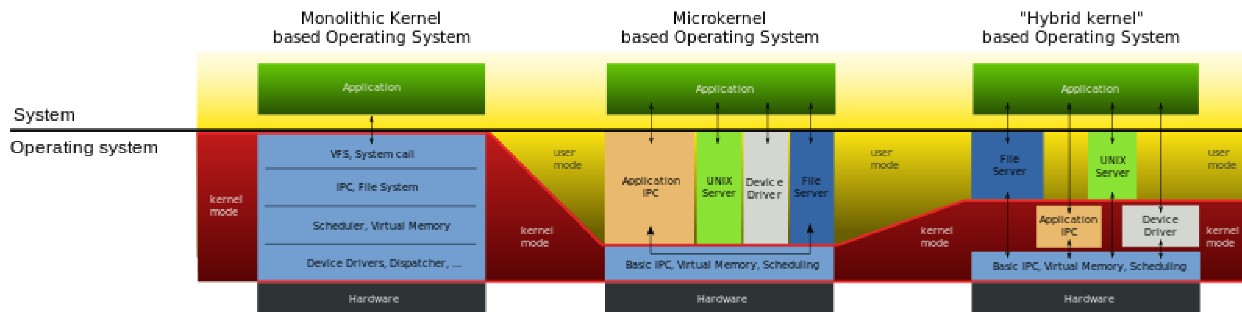


Figure 5 – Hybrid Kernel

1-3-Why Linux?

Because of the numerous economic and technical benefits, we are seeing strong growth in the adoption of Linux for embedded devices. This trend has crossed virtually all markets and technologies. Linux has been adopted for embedded products in the worldwide public switched telephone network, global data networks, and wireless cellular handsets, as well as radio node controllers and backhaul infrastructure that operate these networks. Linux has enjoyed success in automobile applications, consumer products such as games and PDAs, printers, enterprise switches and routers, and many other products. Tens of millions of cell phones are now shipping worldwide with Linux as the operating system of choice. The adoption rate of embedded Linux continues to grow, with no end in sight.[2]

Here are some of the reasons for the growth of embedded Linux:[2]

• Linux supports a vast variety of hardware devices, probably more than any

Other OS.

• Linux supports a huge variety of applications and networking protocols.

• Linux is scalable, from small consumer-oriented devices to large, heavy-iron,carrier class switches and routers.

• Linux can be deployed without the royalties required by traditional proprietary embedded operating systems.

• Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.

• An increasing number of hardware and software vendors, including virtually all the top tier chip manufacturers and independent software vendors (ISVs), now support Linux.

For these and other reasons, we are seeing an accelerated adoption rate of Linux in many common household items, ranging from high-definition televisions to cellular handsets.

1-3-1-GPL [2]

One of the fundamental factors driving the adoption of Linux is the fact that it is open source. For a fascinating and insightful look at the history and culture of the open source movement

The Linux kernel is licensed under the terms of the GNU GPL (General Public License), which leads to the popular myth that Linux is free. In fact, the second paragraph of the GNU GPL Version 3 declares: “When we speak of free software, we are referring to freedom, not price.” Most professional development managers agree: You can download Linux without charge, but development and deployment with any OS on an embedded platform carries an (often substantial) cost. Linux is no different in this regard.

The GPL is remarkably short and easy to read. Here are some of its most important characteristics:

• The license is self-perpetuating.

• The license grants the user freedom to run the program.

• The license grants the user the right to study and modify the source code.

• The license grants the user permission to distribute the original code and his modifications.

• The license is viral. In other words, it grants these same rights to anyone to whom you distribute GPL software.

When software is released under the terms of the GPL, it must forever carry that license.Even if the code is highly modified, which is allowed and even encouraged by the license, the GPL mandates that it must be released under the same license. The intent of this feature is to guarantee freedom of access to the software, including modified versions of the software (or derived works, as they are commonly called).

No matter how the software was obtained, the GPL grants the licensee unlimited distribution rights, without the obligation to pay royalties or per-unit fees. This does not mean that vendors can’t charge for their GPL software—this is a reasonable and common business practice. It means that once in possession of GPL software, it is permissible to modify and redistribute it, whether or not it is a derived (modified) work. However, as dictated by the GPL, the authors of the modified work are obligated to release the work under the terms of the GPL if they decide to

do so. Any distribution of a derived work, such as shipment to a customer, triggers this obligation.

1-3-2-Standards Based [1]

The Linux operating system and accompanying open source projects adhere to industry standards; in most cases, the implementation available in open source is the canonical, or reference, implementation of a standard. A reference implementation embodies the interpretation of the specification and is the basis for conformance testing. In short, the reference implementation is the standard by which others are measured.

If you're new to the notion of a reference implementation, it may be a little confusing. Take for example the Portable Operating System Interface for Unix (POSIX) for handling threads and inter-process communication, commonly called pthreads. The POSIX group, part of the Institute of Electrical and Electronics Engineers (IEEE) is a committee that designs APIs for accomplishing the tasks of interacting with a thread but leaves the implementation of that standard to another group. In practice, when work begins on a standard, one or more of the participants on the committee volunteer to create the code to bring the standard to life, creating the reference implementation. The reference implementation includes a test suite; other implementations consider the passage of the test suite as evidence that the code works as per the specification. Using standards-based software is not only about quality but also about independence. Basing a project on software that adheres to standards reduces the chances of lock-in due to vendor-specific features. A vendor may be well meaning, but the benefits of those extra features are frequently outweighed by the lack of interoperability and freedom that silently become part of the transaction and frequently don't receive the serious consideration they merit. Standards are increasingly important in a world where many embedded devices are connected, many times to arbitrary systems rather than just to each other. The Ethernet is one such connection method, but others abound, like Zigbee, CANbus, and SCSI, to name a few.

1-3-3-Process Isolation and Control [1]

The Linux kernel, at its most basic level, offers these services as a way of providing a common API for accessing system resources:

- Manage tasks, isolating them from the kernel and each other
- Provide a uniform interface for the system's hardware resources
- Serve as an arbiter to resources when contention exists

These are very important features that result in a more stable environment versus an environment where access to the hardware and resources isn't closely managed. For example, in the absence of an operating system, every program running has equal access to all available RAM. This means an overrun bug in one program can write into memory used by another program, which will then fail for what appear to be mysterious, unexplainable reasons until all the code on the system is examined. The notion of resource contention is more complex than just making sure two processes don't attempt to write data to the serial port simultaneously—the

scarcest resource is time, and the operating system can decide what tasks run when in order to maximize the amount of work performed. The following sections look at each item in more detail.

1-3-4-Manage and Isolate Tasks [1]

Linux is a multitasking operating system. In Linux, the word process describes a task that the kernel tracks for execution. The notion of multitasking means the kernel must keep some data about what is running, the current state of the task, and the resources it's using, such as open files and memory. For each process, Linux creates an entry in a process table and assigns the process a separate memory space, file descriptors, register values, stack space, and other process specific information. After it's created, a process can't access the memory space of another process unless both have negotiated a shared memory pool; but even access to that memory pool doesn't give access to an arbitrary address in another process.

Processes in Linux can contain multiple execution threads. A thread shares the process space and resources of the process that started it, but it has its own instruction pointer. Threads, unlike processes, can access each other's memory space. For some applications, this sharing of resources is both desired and convenient; however, managing several threads' contention for resources is a study unto itself. The important thing is that with Linux, you have the design freedom to use these process-control constructs. Processes are isolated not only from each other but from the kernel as well. A process also can't access arbitrary memory from the kernel. Access to kernel functionality happens under controlled circumstances, such as syscalls or file handles. A syscall , short for system call, is a generic concept in operating system design that allows a program to perform a call into the kernel to execute code. In the case of Linux, the function used to execute a system call is conveniently named syscall().When you're working with a syscall, as explained later in this chapter, the operation works much like a regular function call for an API. Using a file handles, you can open what appears to be a file to read and write data. The implementation of a file still reduces to a series of syscalls; but the file semantics make them easier to work with under certain circumstances. The complete separation of processes and the kernel means you no longer have to debug problems related to processes stepping on each other's memory or race conditions related to trying to access shared resources, such as a serial port or network device. In addition, the operating system's internal data structures are off limits to user programs, so there's no chance of an errant program halting execution of the entire system. This degree of survivability alone is why some engineers choose Linux over other lighter-weight solutions.

1-3-5-Peripheral Support [1]

Linux supported more than 200 network adapters, 5 vendors of flash memory, and 10 USB mass storage devices. Because SOC vendors use Linux as their testing system for the chip, support for the chip itself implies support for the components on the device. Wide device support is an artifact of the fact that Linux runs on millions of desktops and servers, representing a customer base that device manufacturers can't ignore. Plus, the open nature of Linux allows device vendors to create drivers without getting a development license from an operating system vendor.

What really differentiates peripheral support on Linux is that the drivers are (primarily) written in C and use the kernel's API to implement their functionality. This means that once a driver has been written for the x86 Linux kernel, it frequently requires zero or a small amount of work for a different processor.

1-3-6-Security [1]

Security means access to data and resources on the machine as well as maintaining confidentiality for data handled by the computer. The openness of Linux is the key to its security. The source code is available for anyone and everyone to review; therefore, security loopholes are there for all to see, understand, and fix.

Security has a few different dimensions, all of which may be necessary for an embedded, or any other, system. One is ensuring that users and programs have the minimal level of rights to resources in order to be able to execute; another is keeping information hidden until a user with the correct credentials requests to see or change it. The advantage of Linux is that all of these tools are freely available to you, so you can select the right ones to meet your project requirements.

1-4-Single-board computer

A single-board computer (SBC) is a complete computer built on a single circuit board, with microprocessor(s), memory, input/output (I/O) and other features required of a functional computer. Single-board computers were made as demonstration or development systems, for educational systems, or for use as embedded computer controllers. Many types of home computer or portable computer integrated all their functions onto a single printed circuit board.

Single board computers were made possible by increasing density of integrated circuits. A single-board configuration reduces a system's overall cost, by reducing the number of circuit boards required, and by eliminating connectors and bus driver circuits that would otherwise be used. By putting all the functions on one board, a smaller overall system can be obtained, for example, as in notebook computers. Connectors are a frequent source of reliability problems, so a single-board system eliminates these problems.

1-4-1-Raspberry Pi

The Raspberry Pi is a credit-card-sized single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of promoting the teaching of basic computer science in schools.

The Raspberry Pi has a Broadcom BCM2835 system on a chip (SoC), which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and was originally shipped with 256 megabytes of RAM, later upgraded to 512 MB. It does not include a built-in hard disk or solid-state drive, but uses an SD card for booting and persistent storage. [15]

Figure 6 – Raspberry PI

1-4-1-1-About [16]

The idea behind a tiny and cheap computer for kids came in 2006, when Eben Upton, Rob Mullins, Jack Lang and Alan Mycroft, based at the University of Cambridge's Computer Laboratory, became concerned about the year-on-year decline in the numbers and skills levels of the A Level students applying to read Computer Science. From a situation in the 1990s where most of the kids applying were coming to interview as experienced hobbyist programmers, the landscape in the 2000s was very different; a typical applicant might only have done a little web design.

Something had changed the way kids were interacting with computers. A number of problems were identified: the colonization of the ICT curriculum with lessons on using Word and Excel, or writing webpages; the end of the dot-com boom; and the rise of the home PC and games console to replace the Amigas, BBC Micros, Spectrum ZX and Commodore 64 machines that people of an earlier generation learned to program on.

There isn't much any small group of people can do to address problems like an inadequate school curriculum or the end of a financial bubble. But we felt that we could try to do something about the situation where computers had become so expensive and arcane that programming experimentation on them had to be forbidden by parents; and to find a platform that, like those old home computers, could boot into a programming environment. From 2006 to 2008, we designed several versions of what has now become the Raspberry Pi.

Although the primary goal of the Raspberry Pi was to educate young student, its cheap price made hype among the amateurs and hobbyists to make cheap embedded projects with it. It’s been just over two year since the initial release of this single-board computer, and already makers have made some incredible projects with it. The Pi is the brain behind multiple home security systems; it is responsible for collecting and displaying data; the Pi has been used as an educational tool for music, mathematics, and geography, among other subjects; artists are building interactive and crowd-sourced installations; enclosures for the Pi range from simple to complex, with all-in-ones emerging as a sub-genre all their own; the Pi can control and automate various systems around the home or office. And so on.

Since the coming of the Raspberry Pi Model B, single-board computers (SBCs) have become a prevalent force in the development world. These pocket-sized devices have taken the online maker community in particular by storm, providing PC functionality to a plethora of open-source projects in amazingly compact, cost-effective, and low-power platforms.

It's not an overstatement to say these tiny computers have engendered a technological revolution of their own by pushing the limits of technological creativity achievable in the palm of one’s hand. As an added benefit, SBCs have served as cheaply obtainable educational tools for teaching the ever-important concepts of computer science to the younger generation. Test engineers, those seeking to build one-off projects, and hobbyists have embraced, and appreciate, this mini computer platform. Similar to how the smartphone changed how we use phones, SBCs are poised to change how we approach embedded systems development.

1-4-2-BeagleBoneBlack

The BeagleBoneBlack, a Texas Instruments-powered SBC, is a member of the BeagleBoard family of development boards. By featuring TI's low-cost Sitara AM335x ARM Cortex-A8

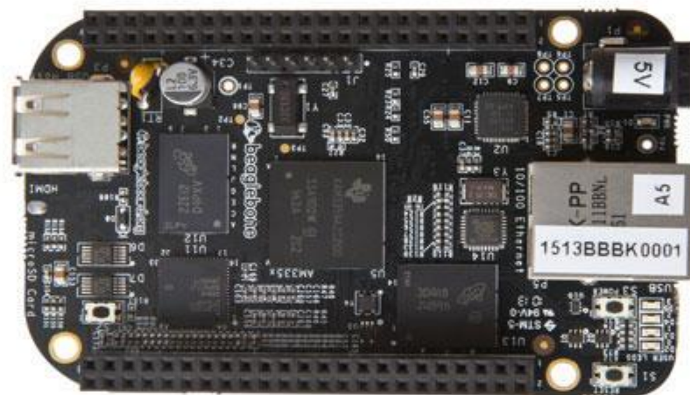


Figure 7 - BeagleBoneBlack

microprocessor [5], the BeagleBoneBlack intends to offer developers a cost-effective solution for builds requiring a plethora of expansion options such as add-on boards. As most development boards of its kind, the BBB supports most Linux distributions and comes with the Angstrom distribution preinstalled.

The BeagleBoneBlack is equipped with 256 MB x 16 DDR3L SDRAM (4 GB), 32 kB EEPROM, and 2 GB eMMC flash as the primary boot source. An onboard microSD slot can also be used for booting and memory storage in addition to the provided serial and USB booting modes. Other onboard interfaces include HDMI, 10/100 Ethernet, serial (for debugging), PC USB, USB 2.0 host port, EtherCAT, and Profibus [17]. Some key applications of the BBB have included motor drives, data backup, data acquisition, robotics, and Twitter printers.

It's a step up from the Raspberry Pi, with quite robust OS support and expansion options. That's why we will use it in our project.

1-4-3-Why we Use BeagleBoneBlack over the Raspberry PI?

These days, a typical microcontroller-based board costs around $20, while the BeagleBoneBlack retails for $45 at the time of press. Other than a more powerful processor, what are you getting for the extra money?[6]

Built-in networking [3]

Not only does the BeagleBoneBlack have an on-board Ethernet connection, but all the basic networking tools that come packaged with Linux are available. You can use services like FTP, Telnet, SSH, or even host your own web server on the board.

Remote access [3]

Because of the built-in network services, it makes it much easier to access electronics projects remotely over the Internet. For example, if you have a data-logging project, you can download the saved data using an FTP client or you can even have your project email you data automatically. Remote access also allows you to log into the device to update the code.

Timekeeping [3]

Without the need for extra hardware, the board can keep track of the date and time of day and can be updated by pinging Internet time servers using the network time protocol (NTP), ensuring that it's always accurate.

Filesystem [3]

Just like our computers, embedded Linux platforms have a built-in file-system, so storing, organizing, and retrieving data is a fairly trivial matter.

Use many different programming languages [3]

You can write your custom code in almost any language you're most comfortable with: C, C++, Python, Perl, Ruby, or even a shell script.

Multitasking [3]

Unlike a typical 8-bit microcontroller, embedded Linux platforms are capable of sharing the processor between concurrently running programs and tasks. This means that if your project needs to upload a large file to a server, it doesn't need to stop its other functions until the upload is over.

Comparing Raspberry Pi and BeagleBone Black

	BeagleBone Black	Raspberry Pi
Base Price	45	35
Processor	1GHz TI Sitara AM3359 ARM Cortex A8	700 MHz ARM1176JZFS
RAM	512 MB DDR3L @ 400 MHz	512 MB SDRAM @ 400 MHz
Storage	2 GB on-board eMMC, MicroSD	SD
Video Connections	1 Micro-HDMI	1 HDMI, 1 Composite
Supported Resolutions	1280×1024 (5:4), 1024×768 (4:3), 1280×720 (16:9), 1440×900 (16:10) all at 16 bit	Extensive from 640×350 up to 1920×1200, this includes 1080p
Audio	Stereo over HDMI	Stereo over HDMI, Stereo from 3.5 mm jack
Operating Systems	Angstrom (Default), Ubuntu, Android, ArchLinux, Gentoo, Minix, RISC OS, others...	Raspbian (Recommended), Ubuntu, Android, ArchLinux, FreeBSD, Fedora, RISC OS, others...
Power Draw	210-460 mA @ 5V under varying conditions	150-350 mA @ 5V under varying conditions
GPIO Capability	65 Pins	8 Pins
Peripherals	1 USB Host, 1 Mini-USB Client, 1 10/100 Mbps Ethernet	2 USB Hosts, 1 Micro-USB Power, 1 10/100 Mbps Ethernet, RPi camera connector

Figure 8 – BeagleBoneBlack Vs Raspberry PI

Linux software [3]

Much of the Linux software that's already out there can be run on the BeagleBone. For example, when I needed to access a USB webcam for one of my projects, I simply downloaded and compiled an open source command line program which let me save webcam images as JPG files.

Linux support [3]

There's no shortage of Linux support information out on the web and community help sites like Stack Overflow come in handy when a challenge comes along.

1-4-4-Ångström distribution

BBB board shipped preinstalled with the Ångström distribution

The Ångström distribution is a Linux distribution for a variety of embedded devices. The distribution is the result of a unification of developers from theOpenZaurus, OpenEmbedded, and OpenSIMpad projects. Amongst other options the user interface in one of the reference builds achievable with BitBake is the GPE Palmtop Environment.

Ångström uses opkg for package management.

The problem with this distribution is that it is not RealTime, so we will have to build the Linux kernel from source after applying RT patch and install it on SD card to run on the board.

1-5-Linux and Real-time [1]

A system is real time when timeliness is a dimension of correctness; that means a correct answer delivered late is the same as an answer that has never been delivered. Real-time systems abound in the real world. To a degree, all systems are real time, because they have real-world deadlines: an airline ticketing program needs to issue tickets before the plane leaves, for example. Meeting deadlines is one aspect of real-time systems; the other is that the system behaves in a predictable manner. If the aforementioned airline reservation system issues tickets within 24 hours and does so for every ticketing request, the system can be considered real time. [1]

1-5-1-Soft Real-time [2]

Most agree that soft real time means that the operation has a deadline. If the deadline is missed, the quality of the experience could be diminished but not fatal. Your desktop workstation is a perfect example of soft real-time requirements. When you are editing a document, you expect to see the results of your keystrokes on the screen immediately. When playing your favorite mp3 file, you expect to have high-quality audio without any clicks, pops, or gaps in the music. [2]

In general terms, humans cannot see or hear delays of less than a few tens of milliseconds. Of course, musicians will tell you that music can be colored by delays smaller than that. If a deadline is missed by these so-called soft real-time events, the results may be undesirable, leading to a lower level of “quality” for the experience, but not catastrophic. [2]

1-5-2-Hard Real-time [2]

Hard real time is characterized by the results of a missed deadline. In a hard real-time system, if a deadline is missed, the results are often catastrophic. Of course, catastrophic is a relative term. If your embedded device is controlling the fuel flow to a jet aircraft engine, missing a deadline to respond to pilot input or a change in operational characteristics can lead to disastrous results. [2]

Note that the deadline’s duration has no bearing on the real-time characteristic. Servicing the tick on an atomic clock is such an example. As long as the tick is processed within the 1-second

window before the next tick, the data remains valid. Missing the processing on a tick might throw off our global positioning systems by feet or even miles! [2]

With this in mind, we draw on a commonly used set of definitions for soft and hard real time. With soft real-time systems, the value of a computation or result is diminished if a deadline is missed. With hard real-time systems, if a single deadline is missed, the system is considered to have failed by definition, and this may have catastrophic consequences. [2]

	Hard Real Time	Soft Real Time
Worst case performance	Same as average performance.	Some multiple of average performance.
Missed deadline	Work not completed by the deadline has no value.	Output of the system after the deadline is less valuable.
Consequences	Adverse real-world consequences: mechanical damage, loss of life or limb.	Reduction in system quality that is tolerable or recoverable, such as a garbled phone conversation.
Typical use cases	Avionics, medical devices, transportation control, safety-critical industrial control.	Networking, telecommunications, entertainment.

[1] **Figure 9 – Hard Vs Soft Real-time**

1-5-3-Linux scheduling

UNIX and Linux were both designed for fairness in their process scheduling. That is, the scheduler tries its best to allocate available resources across all processes that need the CPU and guarantee each process that it can make progress. This very design objective is counter to the requirement for a real-time process. A real-time process must be given absolute priority to run when it becomes ready to run. Real time means having predictable and repeatable latency. [2]

1-5-4-Latency

Real-time processes are often associated with a physical event, such as an interrupt arriving from a peripheral device. The next figure illustrates the latency components in a Linux system. Latency measurement begins upon receipt of the interrupt we want to process. This is indicated by time t0. Sometime later, the interrupt occurs, and control is passed to the interrupt service routine (ISR), as indicated by time t1. This interrupt latency is almost entirely dictated by the maximum interrupt off time 1 —the time spent in a thread of execution that has hardware interrupts disabled. [2]

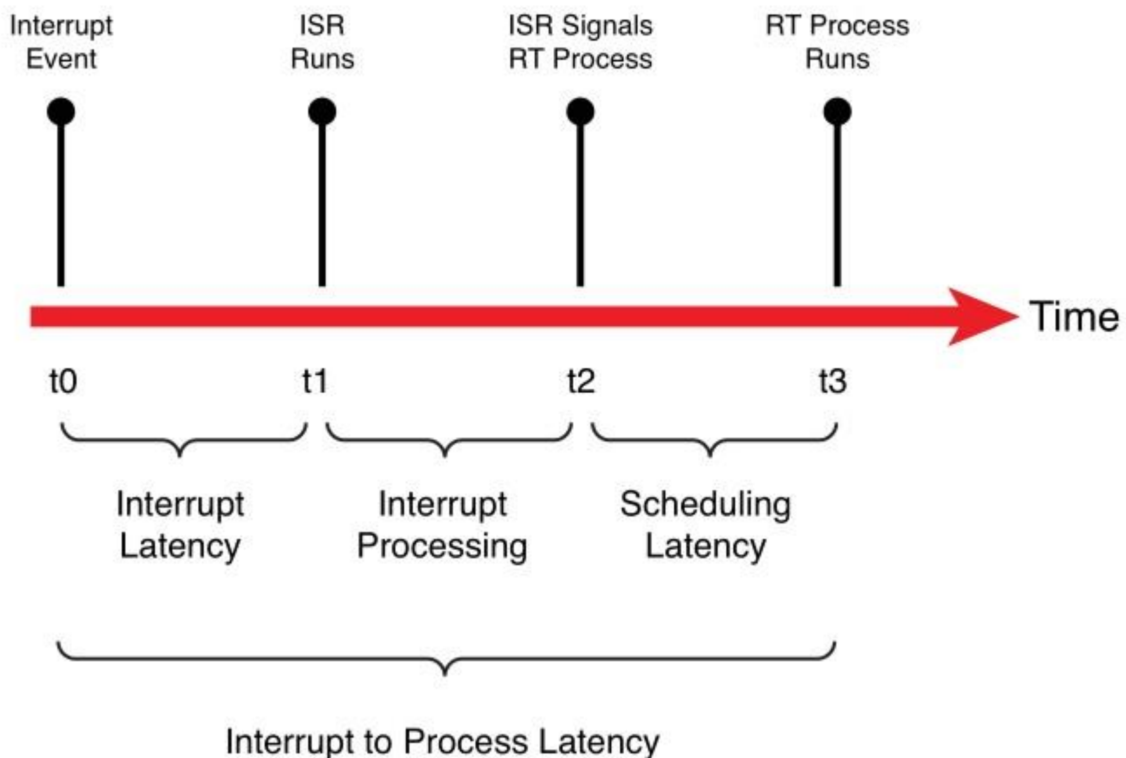


Figure 10 – Interrupt to Process Latency

1-5-5-Kernel preemption

Under Linux, user-space programs have always been preemptible : the kernel interrupts userspace programs to switch to other threads, using the regular clock tick. So, the kernel doesn't wait for user-space programs to explicitly release the processor (which is the case in cooperative multitasking). This means that an infinite loop in an user-space program cannot block the system. [2]

However, until 2.6 kernels, the kernel itself was not preemtible : as soon as one thread has entered the kernel, it could not be preempted to execute another thread. The processor could be used to execute another thread when a syscall was terminated, or when the current thread explicitly asked the scheduler to run another thread using the schedule() function. This means that an infinite loop in the kernel code blocked the entire system, but this is not really a problem: the kernel code is designed so that there are no infinite loops.[18]

However, this absence of preemption in the kernel caused several problems with regard to latency and scalability. So, kernel preemption has been introduced in 2.6 kernels, and one can enable or disable it using the CONFIG_PREEMPT option. If CONFIG_PREEMPT is enabled, then kernel code can be preempted everywhere, except when the code has disabled local interrupts. An infinite loop in the code can no longer block the entire system.
If CONFIG_PREEMPT is disabled, then the 2.4 behavior is restored.[18]

1-5-6-Real-time implementation in Linux [1]

There has been a tremendous amount of work to make Linux a soft real-time system. The current project embodying this work is the CONFIG_PREEMPT_RT patch available at http://rt.wiki.kernel.org. This project first focused on reducing the latency in the Linux kernel by making it more preemptable. When the Linux kernel was first ported to a multiprocessor system (that is, a machine that could execute more than one thread of code concurrently), the kernel's race conditions were solved by using a single lock called the Big Kernel Lock (BKL); this resulted in serialization throughout the kernel, because many unrelated routines used this lock to quell race conditions. The result was that some code waited for resources that weren't necessarily used by the lock elsewhere. In addition to lacking granularity, the BKL was a spinlock , meaning the code requesting the lock waited in a loop: the process waiting for the next lock wouldn't yield, resulting in a system that couldn't be scheduled in a real-time sense. Preemption is vital to real-time systems. Interruption is necessary to meet deadlines if a lower priority task is monopolizing resource that a higher-priority task needs or when the lock is taking time while the processor could be doing some other activity. This project includes more than just preemption:

- High-resolution timers: This part of the patch has been recently accepted into the mainline kernel. High-resolution timers are frequently a requirement in a real-time system when a task needs to occur more frequently than the 1 millisecond resolution offered with Linux. A working high-resolution timer, however, depends on a platform that has a dependable high-resolution clock. Not all processors have the required hardware, or the clock itself may have enough jitter that it isn't useful for a timing-sensitive application.[1]

- Priority inheritance mutexes: A mutex (short for mutually exclusive) is a lock put around code so that only one thread can execute that code at a time. The enclosed code is called a critical section . Mutexes that have priority inheritance reduce the amount of time in a priority inversion, because when a thread with a lower priority is waiting to run the critical section, the thread holding the mutex runs at the priority of the waiting thread.[1]

- Schedulable soft IRQ threads: Interrupts have two parts: the actual hardware interrupt called by the processor, called the top half, and the processing that can be deferred until later, or the bottom half. When the top half is running, the processor is doing nothing other than running that code; that means a timer may miss a deadline, resulting in jitter. By splitting the interrupt handling into two parts, the top half can do as little work as possible so as not to interfere with other threads. For example, an interrupt attached to a GPIO pin should collect that the pin has been fired and defer processing to a thread that can be scheduled.[1]

2-Building the Linux Kernel

in the following sections we will walk through the Linux building process, but first we need to prepare our working environment and distinguish between some idioms

2-1-Bios Vs bootloader [2]

When power is first applied to the desktop computer, a software program called the BIOS immediately takes control of the processor. (Historically, BIOS was an acronym meaning Basic Input/output Software, but the term has taken on a meaning of its own as the functions it performs have become much more complex than the original implementations.) The BIOS might actually be stored in Flash memory (described shortly) to facilitate field upgrade of the BIOS program itself. [2]

The BIOS is a complex set of system-configuration software routines that have knowledge of the low-level details of the hardware architecture. Most of us are unaware of the extent of the BIOS and its functionality, but it is a critical piece of the desktop computer. The BIOS first gains control of the processor when power is applied. Its primary responsibility is to initialize the hardware, especially the memory subsystem, and load an operating system from the PC's hard drive. [2]

In a typical embedded system (assuming that it is not based on an industry standard x86 PC hardware platform), a bootloader is the software program that performs the equivalent functions. [2]

When power is first applied to a processor board, many elements of hardware must be initialized before even the simplest program can run. Each architecture and processor has a set of predefined actions and configurations upon release of reset, which includes fetching initialization code from an onboard storage device (usually Flash memory). This early initialization code is part of the bootloader and is responsible for breathing life into the processor and related hardware components.[2]

The board comes with Das U-Boot installed, so after building the kernel Das U-Boot will boot and load the new kernel into memory and execute it.

2-2-Anatomy of BBB

Below is the high level block diagram of the BeagleBoneBlack. [19]

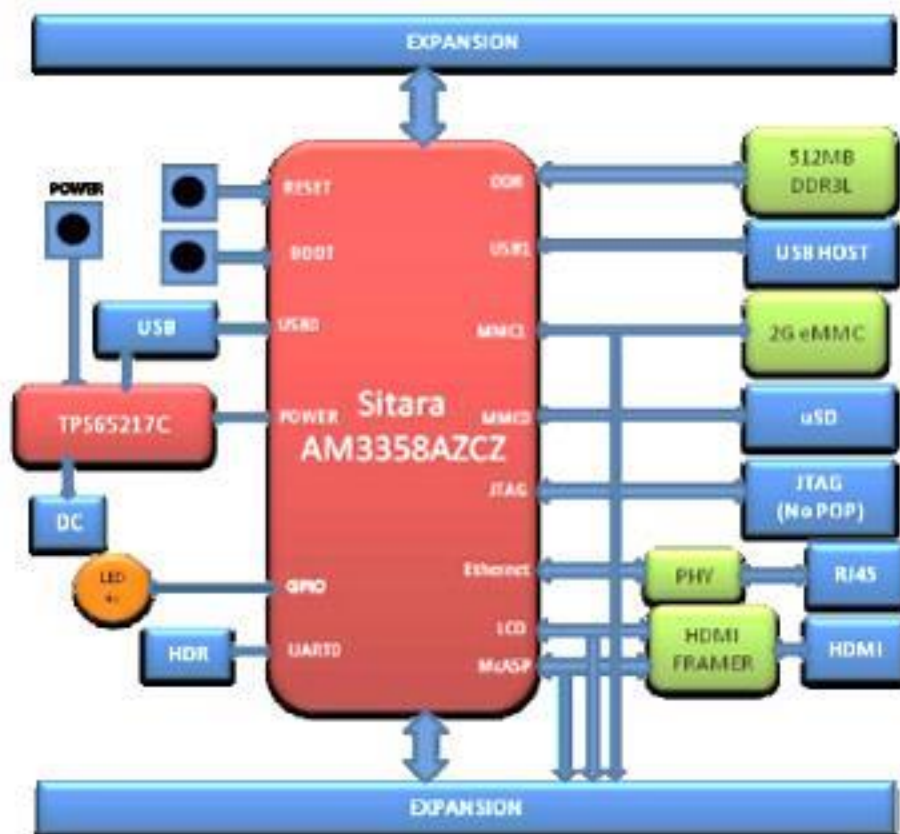


Figure 11 – Anatomy of BBB

2-2-1-Processor

For the initial release, the board uses the Sitara XAM3359AZCZ processor in the 15x15 package.[17]

2-2-2-DDR3L Memory

The BeagleBoneBlack uses a single MT41K256M16HA-125 512MB DDR3L device from Micron that interfaces to the processor over 16 data lines, 16 address lines, and 14 control lines.[19]

2-2-3- On Board Flash Memory

The BeagleBoneBlack sports on-board flash memory and can therefore be booted without a

MicroSD card inserted. In the technical manuals, this memory is referred to as the eMMC.[19]

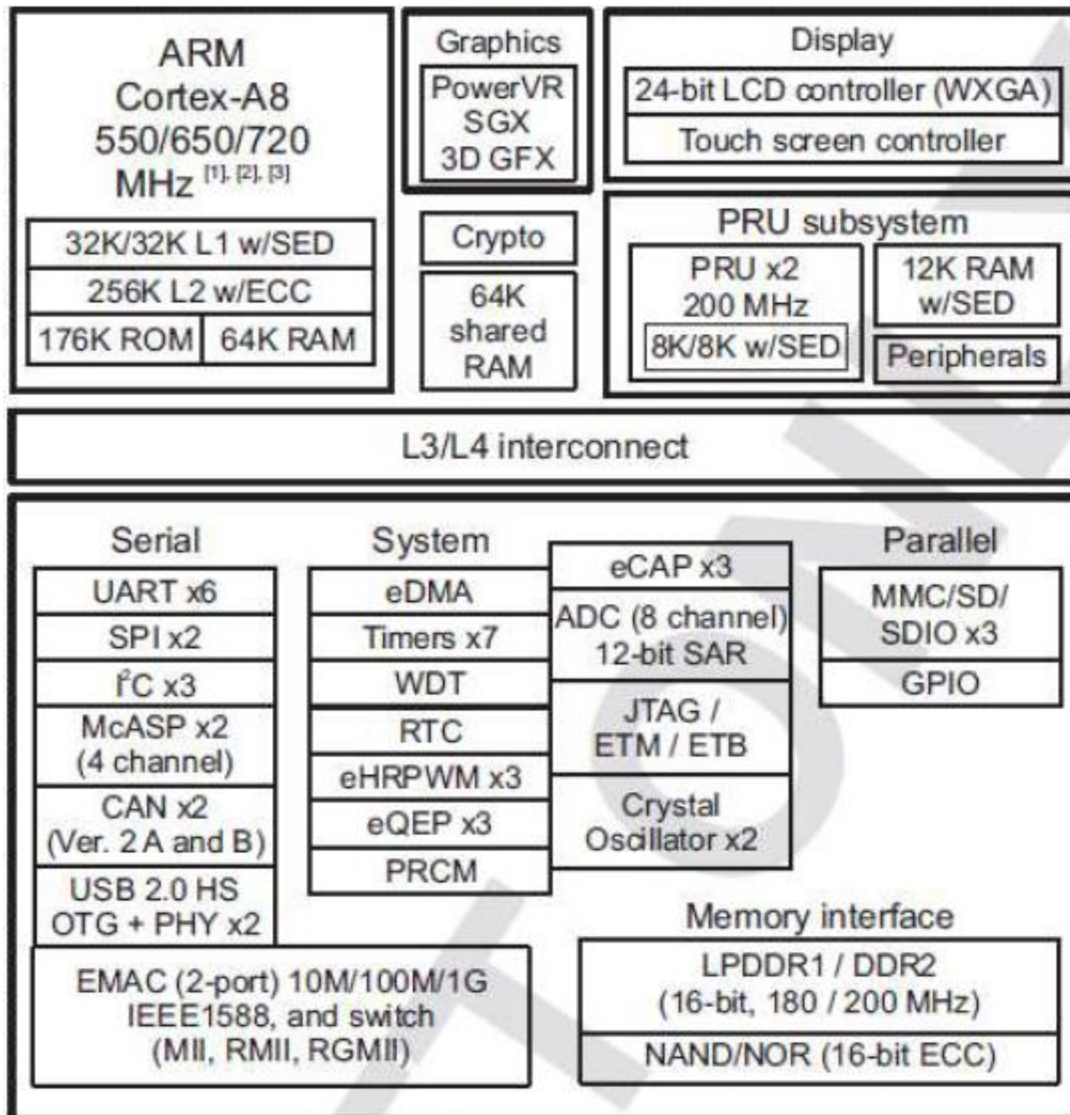


Figure 12 – ARM Sitara Architecture

2-3-Typical embedded Linux setup

Next figure is a common arrangement. It shows a host development system, running your favorite desktop Linux distribution, such as Red Hat, SUSE, or Ubuntu Linux. The embedded Linux target board is connected to the development host via an RS-232 serial cable. You plug the target board's Ethernet interface into a local Ethernet hub or switch, to which your development host is also attached via Ethernet. The development host contains your development tools and utilities along with target files, which normally are obtained from an embedded Linux distribution.[2]

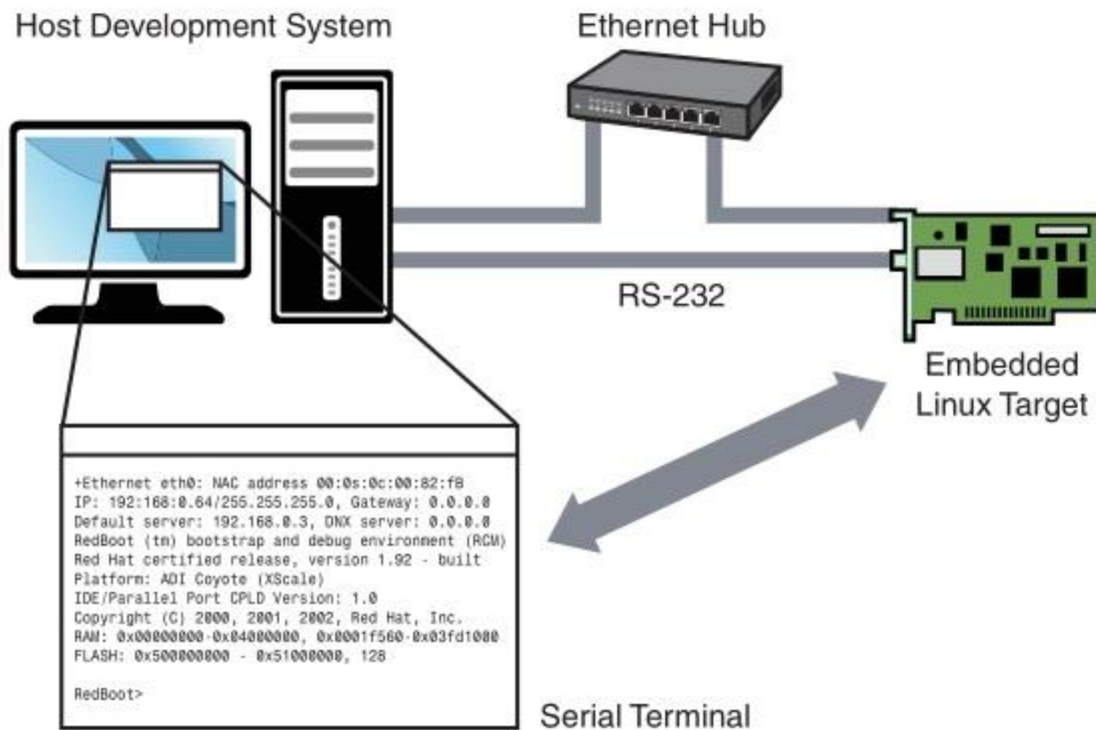


Figure 13 – Linux Lap setup

2-4-The building Process

To build the Kernel we will follow the following steps:

2-4-1-Installing the cross compiler & GIT

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. Cross compiler tools are used to generate executable for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system.

We are using Ubuntu Linux running on Intel X86 machine and we will generate executable code running on ARM processor, so we are cross compiling.

On the shell:

mns@ubuntu:~$ sudo apt-get install gcc-arm-linux-gnueabi

The command will fetch the cross compiler from Ubuntu Repository source.

Next , we create a new directory in our home folder to build everything inside it.

mns@ubuntu:~$ mkdir beaglebb

Then we install Git

mns@ubuntu:~$ sudo apt-get install git

2-4-2-Getting the sources

Then we fetch the Patches for the BeagleBoneblack

mns@ubuntu:~$ cd beaglebb/

mns@ubuntu:~/beaglebb$ git clone git://github.com/RoberCNelson/linux-dev.git

```
mns@ubuntu:~$ cd beaglebb/
mns@ubuntu:~/beaglebb$ git clone git://github.com/RoberCNelson/linux-dev.git
Cloning into 'linux-dev'...
fatal: remote error:
  Repository not found.
mns@ubuntu:~/beaglebb$ git clone git://github.com/RobertCNelson/linux-dev.git
Cloning into 'linux-dev'...
remote: Reusing existing pack: 23377, done.
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 23392 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (23392/23392), 21.27 MiB | 522.00 KiB/s, done.
Resolving deltas: 100% (12979/12979), done.
Checking connectivity... done
mns@ubuntu:~/beaglebb$
```

Figure 14 – cloning Github Repository

Next we need to check out the source for our board

mns@ubuntu:~/beaglebb$ cd linux-dev/

mns@ubuntu:~/beaglebb/linux-dev$ git checkout origin/am33x-v3.2 -b am33x-v3.2

```
mns@ubuntu:~/beaglebb$ cd linux-dev/
mns@ubuntu:~/beaglebb/linux-dev$ git checkout origin/am33x-v3.2 -b am33x-v3.2
Branch am33x-v3.2 set up to track remote branch am33x-v3.2 from origin.
Switched to a new branch 'am33x-v3.2'
mns@ubuntu:~/beaglebb/linux-dev$
```

Figure 15 – Checkout source

Next, we will clone the Linux Kernel source with RT patch

mns@ubuntu:~/beaglebb$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git

```
mns@ubuntu:~/beaglebb$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git
Cloning into 'linux-stable-rt'...
remote: Counting objects: 3933481, done.
remote: Compressing objects: 100% (589799/589799), done.
remote: Total 3933481 (delta 3336777), reused 3909353 (delta 3312729)
Receiving objects: 100% (3933481/3933481), 814.43 MiB | 155.00 KiB/s, done.
Resolving deltas: 100% (3336777/3336777), done.
Checking connectivity... done
Checking out files: 100% (37624/37624), done.
mns@ubuntu:~/beaglebb$
```

Figure 16 – clone the RT Kernel source

A typical RTOS uses priorities. The highest priority task wanting the CPU always gets the CPU within a fixed amount of time after the event waking the task has taken place. On such an RTOS the latency of a task only depends on the tasks running at equal or higher priorities, all other tasks can be ignored. On a normal OS (such as normal Linux) the latencies depend on everything running on the system, which of course makes it much harder to be convinced that the deadlines will be met every time on a reasonably complicated system. This is because preemption can be switched off for an unknown amount of time. The high priority task wanting to run can thus be delayed for an unknown amount of time by low priority tasks running with preemption switched off. [20]

Traditionally, the Linux kernel will only allow one process to preempt another only under certain circumstances:

- When the CPU is running user-mode code
- When kernel code returns from a system call or an interrupt back to user space
- When kernel code code blocks on a mutex, or explicitly yields control to another process

If kernel code is executing when some event takes place that requires a high priority thread to start executing, the high priority thread cannot preempt the running kernel code, until the kernel code explicitly yields control. In the worst case, the latency could potentially be hundreds milliseconds or more.[20]

RT Linux kernel has an additional configuration option, CONFIG_PREEMPT, which causes all kernel code outside of spinlock-protected regions and interrupt handlers to be eligible for non-voluntary preemption by higher priority kernel threads. With this option, worst case latency drops to (around) single digit milliseconds, although some device drivers can have interrupt handlers that will introduce latency much worse than that. If a real-time Linux application requires latencies smaller than single-digit milliseconds, use of the CONFIG_PREEMPT_RT patch is highly recommended. [20]

2-4-3-Configuring the build script

mns@ubuntu:~/beaglebb$ cd linux-dev/

mns@ubuntu:~/beaglebb/linux-dev$ ls

build_deb.sh LICENSE patch.sh repo_maintenance system.sh.sample version.sh

build_kernel.sh   patches README   scripts   tools

mns@ubuntu:~/beaglebb/linux-dev$ cp system.sh.sample system.sh

mns@ubuntu:~/beaglebb/linux-dev$ vi system.sh

When editing system.sh using VI editor, we change the following values

CC=arm-linux-gnueabi-   //cross compiler used

LINUX_GIT=/home/mns/beaglebb/linux-stable-rt/   //path of linux source

##For TI: OMAP3/4/AM35xx   //location of memory address of the kernel

ZRELADDR=0x80008000

2-4-4-applying the patches and configuring the kernel

mns@ubuntu:~/beaglebb/linux-dev$ ./build_kernel.sh

the script will apply the patches of Beagle bone black on the source and pop out menuconfig of the kernel

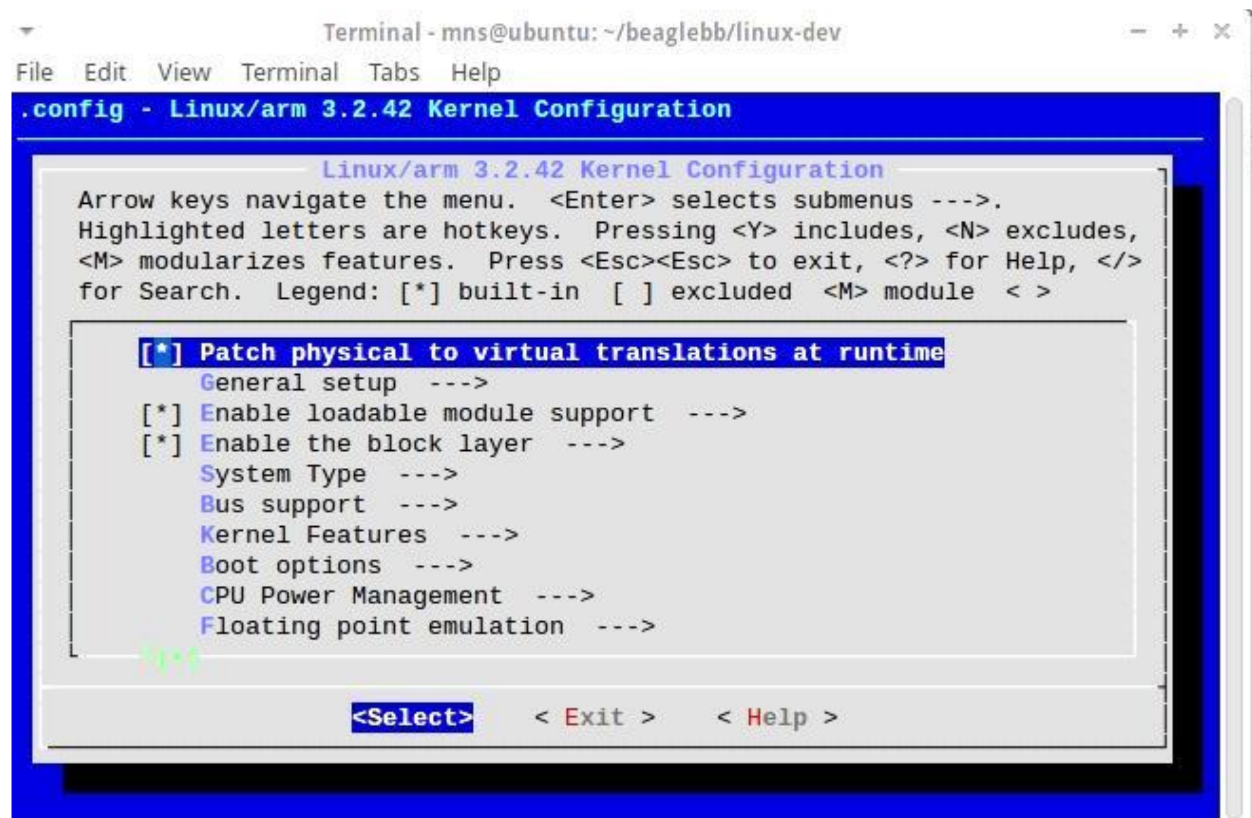


Figure 17 – Kernel Feature

We select Kernel Features----->Preemption Model------>Fully Preemptible Kernel (RT)

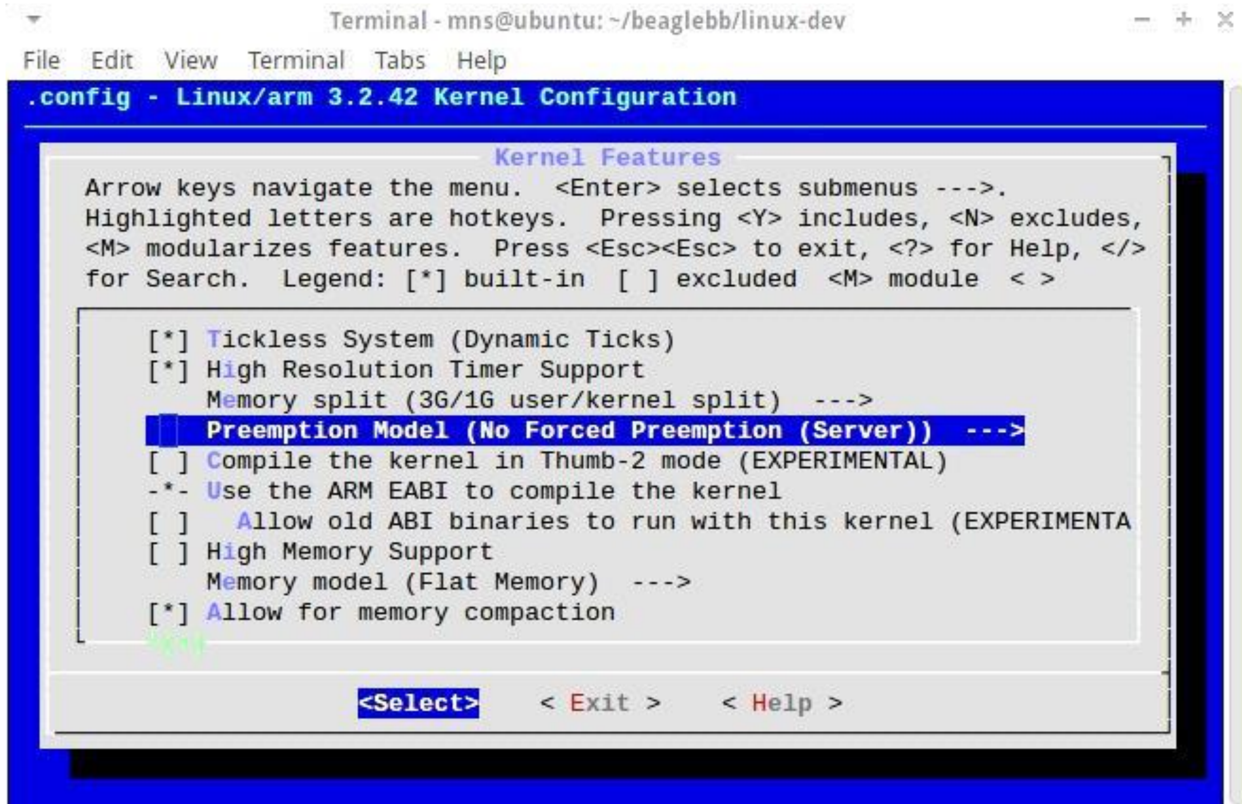


Figure 18 – Preemption Model

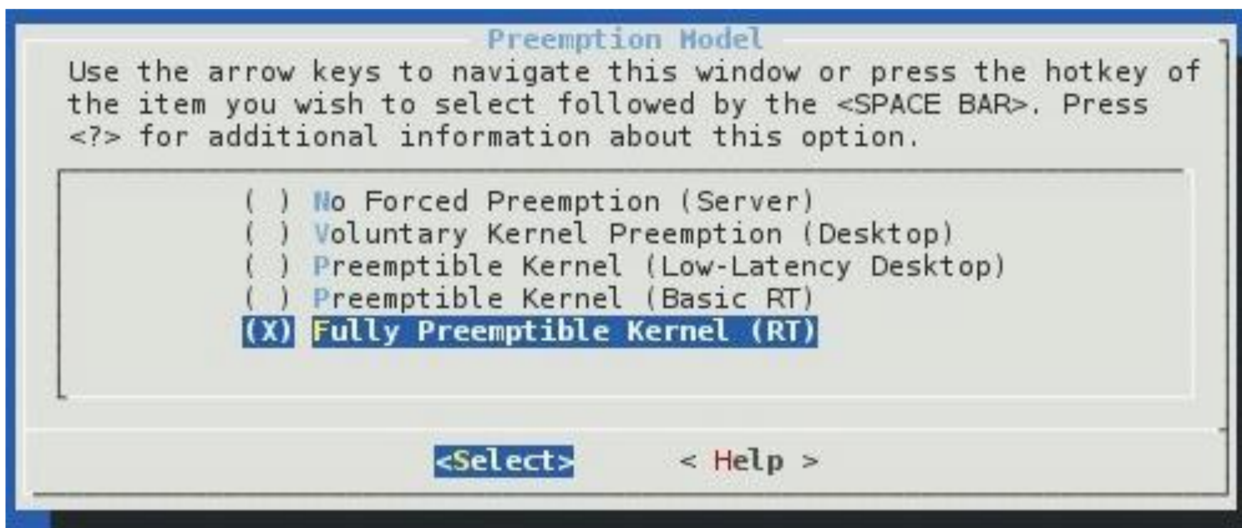


Figure 19 – Fully Preemptible Kernel

2-4-4-1-RealTime Kernel Performance analysis

Ftrace is a powerful set of tracing tools that can give the developer a detailed look at what is going on inside the kernel.

Ftrace must be enabled in the kernel configuration before it can be used.

We select Kernel Hacking----->Tracers

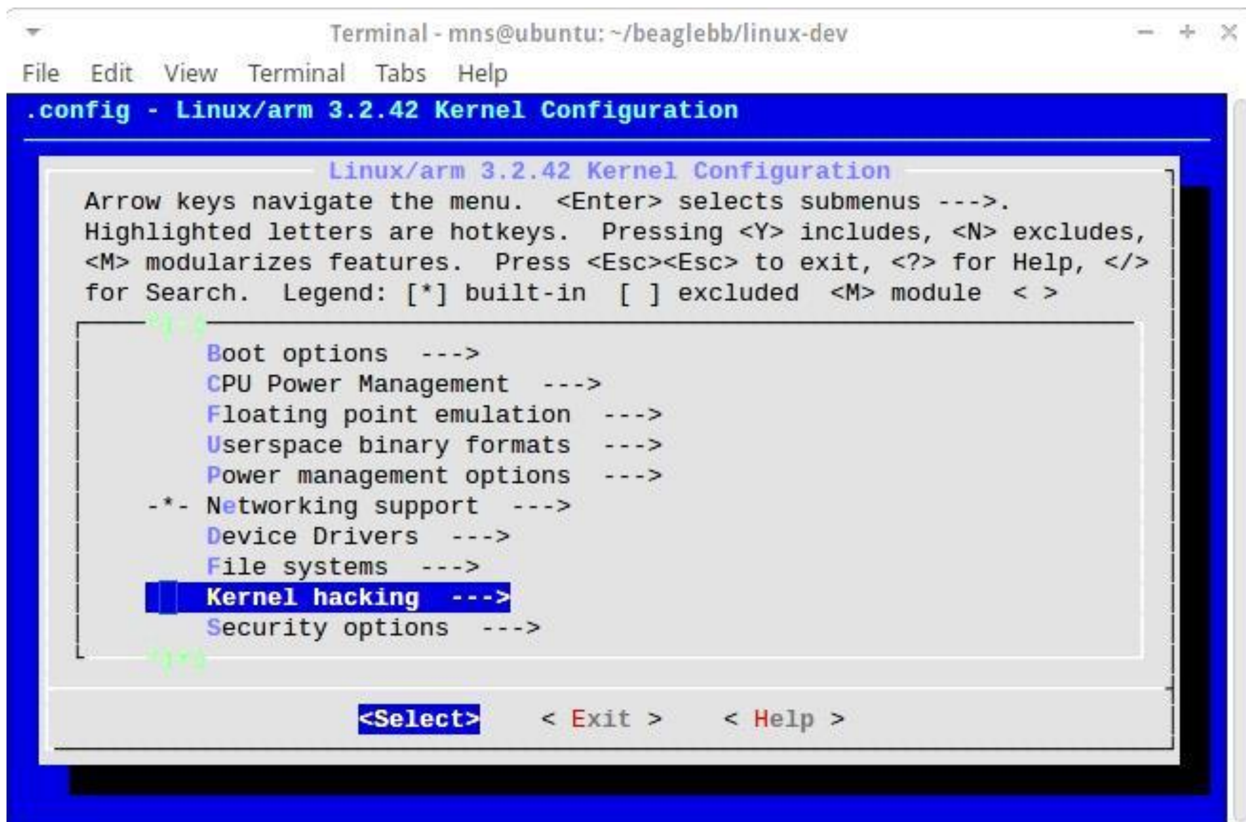


Figure 20 – Kernel Hacking

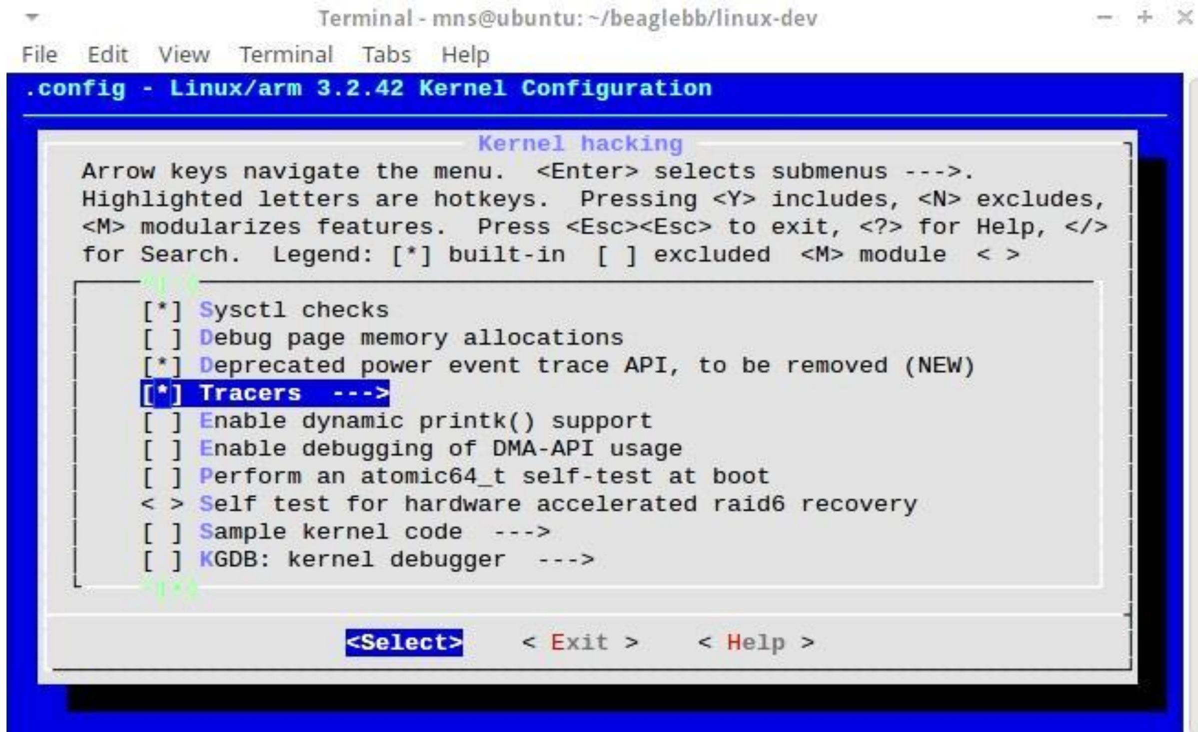


Figure 21 – Tracers

Then we enable Interrupts off latency Tracer, Preemption off latency Tracer, scheduling latency Tracer.

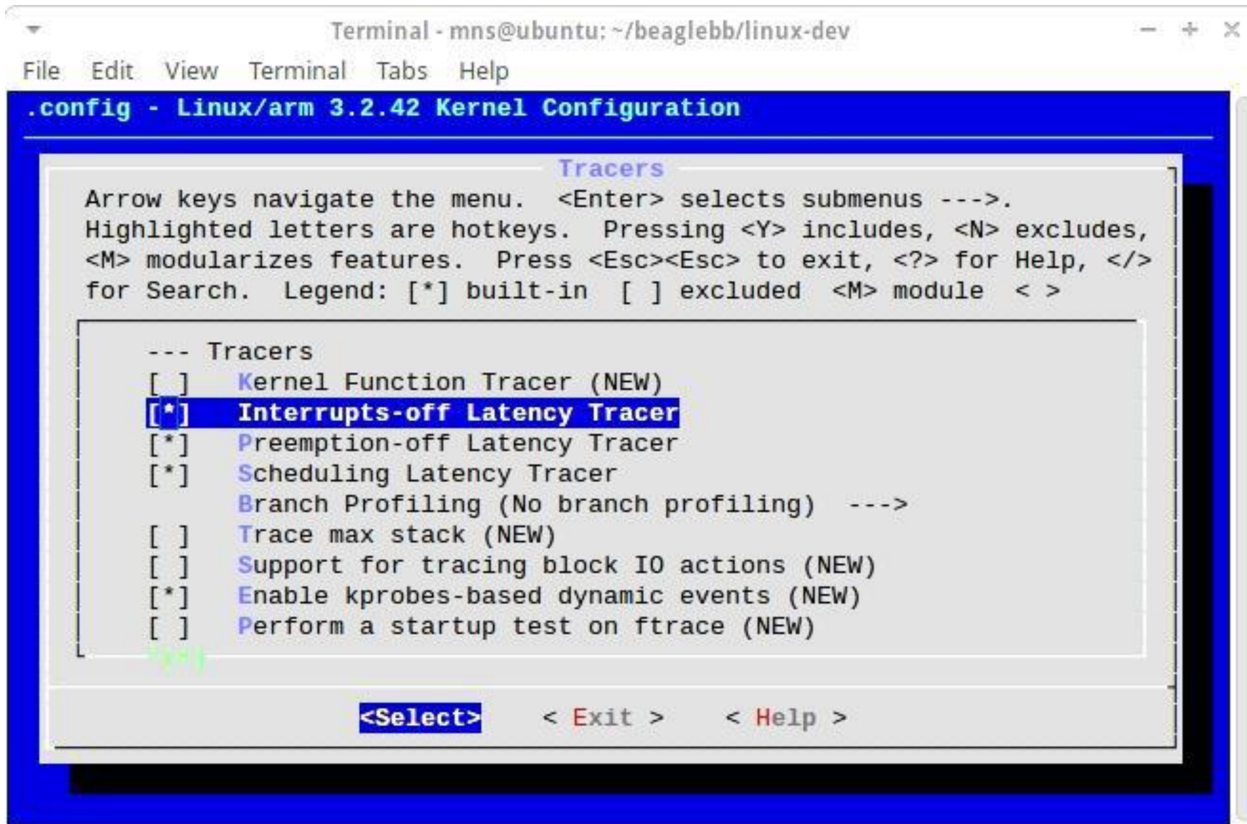


Figure 22 – Tracers options

Finally we exit the menuconfig and save the new configuration

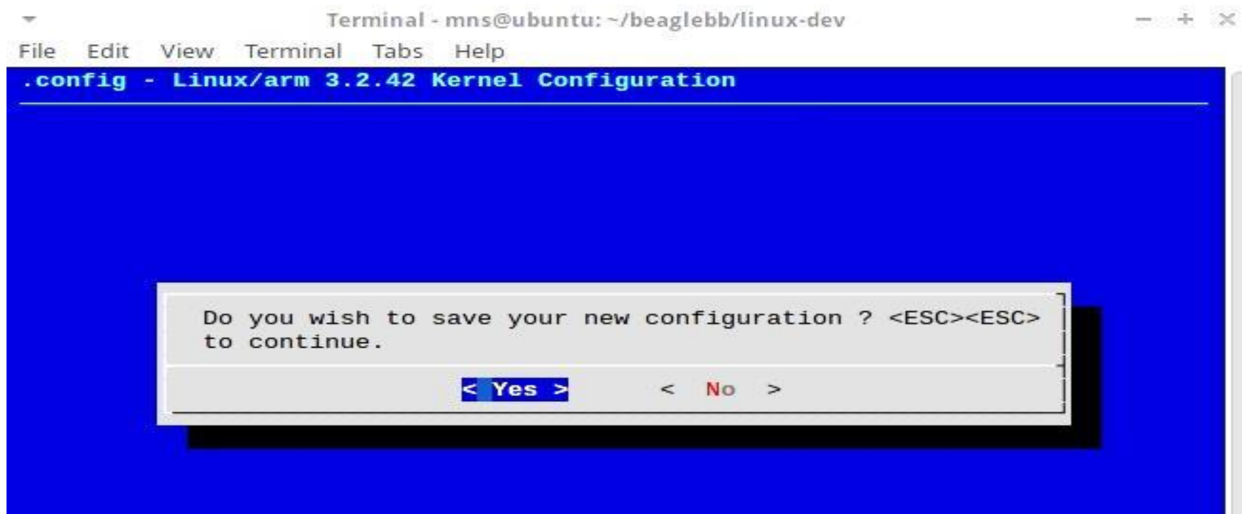


Figure 23 – Save new Configuration

2-4-5-Compiling the kernel

Once we exit, the script will use the new configuration and compile the kernel using ARM cross compiler

this is equivalent to this command

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-

and to build any kernel modules:

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules

When the compilation process finishes, the new Kernel will be located on the path /home/user/beaglebb/linux-dev/KERNEL/arch/arm/boot/

2-5-Deploying the new kernel

After compiling the Kernel, we need to deploy our Kernel on the beagleboneBlack. The easiest way to do so is to install it on a SD card and boot from it.

We will use Ubuntu root file systems with our kernel

We grab the rootFS from http://rcn-ee.net/deb/rootfs/saucy/ubuntu-13.10-console-armhf-2013-11-15.tar.xz

mns@ubuntu:~/beaglebb/ubuntu$ wget http://rcn-ee.net/deb/rootfs/saucy/ubuntu-13.10-console-armhf-2013-11-15.tar.xz

```
mns@ubuntu:~/beaglebb$ mkdir ubuntu
mns@ubuntu:~/beaglebb$ cd ubuntu/
mns@ubuntu:~/beaglebb/ubuntu$ wget http://rcn-ee.net/deb/rootfs/saucy/ubuntu-13.
10-console-armhf-2013-11-15.tar.xz
--2014-03-15 16:06:13--  http://rcn-ee.net/deb/rootfs/saucy/ubuntu-13.10-console
-armhf-2013-11-15.tar.xz
Resolving rcn-ee.net (rcn-ee.net)... 69.163.128.251
Connecting to rcn-ee.net (rcn-ee.net)|69.163.128.251|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 141256568 (135M) [application/x-tar]
Saving to: 'ubuntu-13.10-console-armhf-2013-11-15.tar.xz'

100%[======================================>] 141,256,568  344KB/s   in 7m 23s

2014-03-15 16:13:37 (312 KB/s) - 'ubuntu-13.10-console-armhf-2013-11-15.tar.xz'
saved [141256568/141256568]

mns@ubuntu:~/beaglebb/ubuntu$
```

Figure 24 – Get root file system

Then we extract the files:

mns@ubuntu:~/beaglebb/ubuntu$ xz -d ubuntu-13.10-console-armhf-2013-11-15.tar.xz

mns@ubuntu:~/beaglebb/ubuntu$ tar -xf ubuntu-13.10-console-armhf-2013-11-15.tar

```
mns@ubuntu:~/beaglebb/ubuntu$ ls
ubuntu-13.10-console-armhf-2013-11-15.tar.xz
mns@ubuntu:~/beaglebb/ubuntu$ xz -d ubuntu-13.10-console-armhf-2013-11-15.tar.xz

mns@ubuntu:~/beaglebb/ubuntu$ tar -xf ubuntu-13.10-console-armhf-2013-11-15.tar
mns@ubuntu:~/beaglebb/ubuntu$
```

Figure 25 – Extract files

mns@ubuntu:~/beaglebb/ubuntu/ubuntu-13.10-console-armhf-2013-11-15$ sudo ./setup_sdcard.sh --mmc dev/sde --uboot bone

Till now we have an SD Card with normal Ubuntu Distribution, but we need it install our compiled Kernel on it.

We run the following command on the terminal to copy the kernel onto the SD card

mns@ubuntu:~/beaglebb/linux-dev$ ./tools/install_kernel.sh

the script will detect the card and install the Kernel automatically.

Now, we have a working image on the card.

2-6-Logging to the Board over Secure shell SSH

We insert the SD card into the board and press the S2 jumper to force the board to boot from the SD card and connect the USB cable to the PC

We connect using SSH

The default IP address is 192.168.7.2

UserName: ubuntu

Password: temppwd

```
mns@ubuntu:~$ ssh 192.168.7.2 -l ubuntu
ubuntu@192.168.7.2's password:
Welcome to Ubuntu 13.10 (GNU/Linux 3.8.13-bone37 armv7l)

 * Documentation:  https://help.ubuntu.com/
Last login: Sat Jan 25 00:53:43 2014 from ubuntu.local
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@arm:~$
```

Figure 26 – Logging using SSH

We enter the command uname –a to get the Linux version

```
root@arm:/home/ubuntu# uname -a
Linux arm 3.8.13-bone37 #1 SMP Thu Jan 23 03:10:52 UTC 2014 armv7l armv7l armv7l
 GNU/Linux
root@arm:/home/ubuntu#
```

Figure 27 – Kernel Name

2-7-Benchmarking the Kernel

One of the most critical measurements of interest to real-time developers is how long it takes to get the high-priority task running after it has been signaled to do so. When a real-time process (one with the SCHED_FIFO or SCHED_RR scheduling attribute) is running in your system, it is by definition sharing the processor with other tasks. When an event needs servicing, the real-time task is woken up. In other words, the scheduler is informed that it needs to run.[2] Wakeup timing is the time from the wakeup event until the task actually gets the CPU and begins to run.

2-7-1-FTRACE results

Ftrace has a wakeup and wakeup_rt trace facility. This facility records and traces the longest latency from wakeup to running while the tracer is enabled.[2]

Ftrace has its controllers in the debugfs system. This is usually mounted in /sys/kernel/debug. If it is not already mounted, then you can mount it yourself with:

mount -t debugfs nodev /sys/kernel/debug

Note: you must run mount in superuser mode denoted by #

When this is complete, you should find a directory under /sys/kernel/debug called tracing . This tracing directory contains all the controls and output sources for Ftrace data.[2]

As suggested in the kernel documentation, we will use a symlink called /tracing to simplify reporting and interacting with the Ftrace subsystems:

ln -s /sys/kernel/debug/tracing /tracing

From here on, we will reference /tracing instead of the longer /sys/kernel/debug/tracing.

Ftrace has a wakeup and wakeup_rt trace facility. This facility records and traces the longest latency from wakeup to running while the tracer is enabled.[2]

We run a simple C test program that creates and writes to a file. Prior to the file I/O, the test program elevates itself to SCHED_RR with priority 99 and sets up the tracing system using writes to stdio, similar to issuing the following commands from the shell:

echo 0 > /tracing/tracing_enabled

echo 0 > /tracing/tracing_max_latency (resets the max record back to zero)

echo wakeup > /tracing/current_tracer

echo 1 > /tracing/tracing_enabled

After our test program issues these commands, tracing has completed. Notice in this case that the maximum wakeup latency is reported as 7 microseconds.

When the test program runs, it performs the file I/O and then sleeps. This guarantees that it will yield the processor even if it did not block on I/O.

The maximum latency is provided separately in another trace file. To display the maximum wakeup latency during a particular tracing run, simply issue this command:

root@speedy:~# cat /tracing/tracing_max_latency

7

2-8-Comparing Linux vs. commercial RTOS

We will list now some commercial RTOS systems benchmarking to compare them to linux

2-8-1-VxWorks

Average latency 1.7 microseconds, maximum latency 6.8 microseconds on a 200MHz Pentium machine [21]

2-8-2-Windows CE .NET

Average latency 2.4 microseconds, maximum latency 5.6 microseconds on a 200MHz Pentium machine [21]

2-8-3-QNX Neutrino RTOS

Average latency 1.6 microseconds, maximum latency 4.1 microseconds on a 200MHz Pentium machine [21]

2-9-Conclusion

It is clear that the commercial RTOS like QNX has much better response time than RTLINUX.

We would recommend using RTLinux in Soft Real-time systems only, If there is a system with hard Real-time requirement, It is recommended to use Commercial RTOS.

3-Case study Project: Farm Automation System

The market for cheap single-board computers is becoming one of the most surprisingly competitive spaces in the tech industry. The release of this cheap computer opens the door for many applications that were very hard to implement before.

In this project, we will implement a cheap Farm Automation System to monitor a farm and control the irrigation process.

We will use BeagleBoneBlack as a control computer with some sensors.

The reduction of water scarcity is a goal of many countries and governments. That's why we should not waste water. It is know that agriculture consume a huge amount of water resources. By conserving the water used in agriculture or use it in a better way, we would save a lot of resources.[22]

Fifty years ago (as of 2010), the common perception was that water was an infinite resource. At that time, there were fewer than half the current numbers of people on the planet. People were not as wealthy as today, consumed fewer calories and ate less meat, so less water was needed to produce their food. They required a third of the volume of water we presently take from rivers. Today, the competition for water resources is much more intense. This is because there are now more than seven billion people on the planet, their consumption of water-thirsty meat and vegetables is rising, and there is increasing competition for water from industry, urbanization and biofuel crops. To avoid a global water crisis, farmers will have to strive to increase productivity to meet growing demands for food, while industry and cities find ways to use water more efficiently.[22]

3-1-Irrigation

Irrigation is the artificial application of water to the land or soil. It is used to assist in the growing of agricultural crops, maintenance of landscapes, and revegetation of disturbed soils in dry areas and during periods of inadequate rainfall. Additionally, irrigation also has a few other uses in crop production, which include protecting plants against frost [23], suppressing weed growth in grain fields[24] and preventing soil consolidation.

Various types of irrigation techniques differ in how the water obtained from the source is distributed within the field. In general, the goal is to supply the entire field uniformly with water, so that each plant has the amount of water it needs, neither too much nor too little. The modern methods are efficient enough to achieve this goal. Water use efficiency in the field can be determined as follows:[25]

Field Water Efficiency (%) = (Water Transpired by Crop ÷ Water Applied to Field) x 100

Water is lost from plant environment in 2 processes: [26]

1. Evaporation (from soil with high temperature) and

2. Transpiration (water loss from plant tissue to reduce its temperature), we add the two amounts in Evapotranspiration ET

In irrigation process we try to add the same amount of water that has evaporated, so Irrigation water requirements for plants = evapotranspiration

The crop water need (ET crop) is defined as the depth (or amount) of water needed to meet the water loss through evapotranspiration. In other words, it is the amount of water needed by the various crops to grow optimally.[26]

The crop water need always refers to a crop grown under optimal conditions, i.e. a uniform crop, actively growing, completely shading the ground, free of diseases, and favorable soil conditions (including fertility and water). The crop thus reaches its full production potential under the given environment.[26]

The crop water need mainly depends on:

- The climate: in a sunny and hot climate crops need more water per day than in a cloudy and cool climate

- The crop type: crops like maize or sugarcane need more water than crops like millet or sorghum

- The growth stage of the crop; fully grown crops need more water than crops that have just been planted.

For evaporation, we can predict it using heat and humidity sensor especially if we are growing plants in a green house

For Transpiration, we can use predetermined sheets for each plant.

3-2-System prototype

Our system has a water barrel (or any water source like tape water) connected to a nozzle with a valve to open or close it, controlled by the BeagleBoneBlack (BBB) unit that reads a soil moisture sensor buried in the soil. When the soil humidity gets below a threshold, it turns on the pump until a humidity ceiling is reached.

We connect the heat and Humidity sensor to the BBB for more accurate evaporation determination.

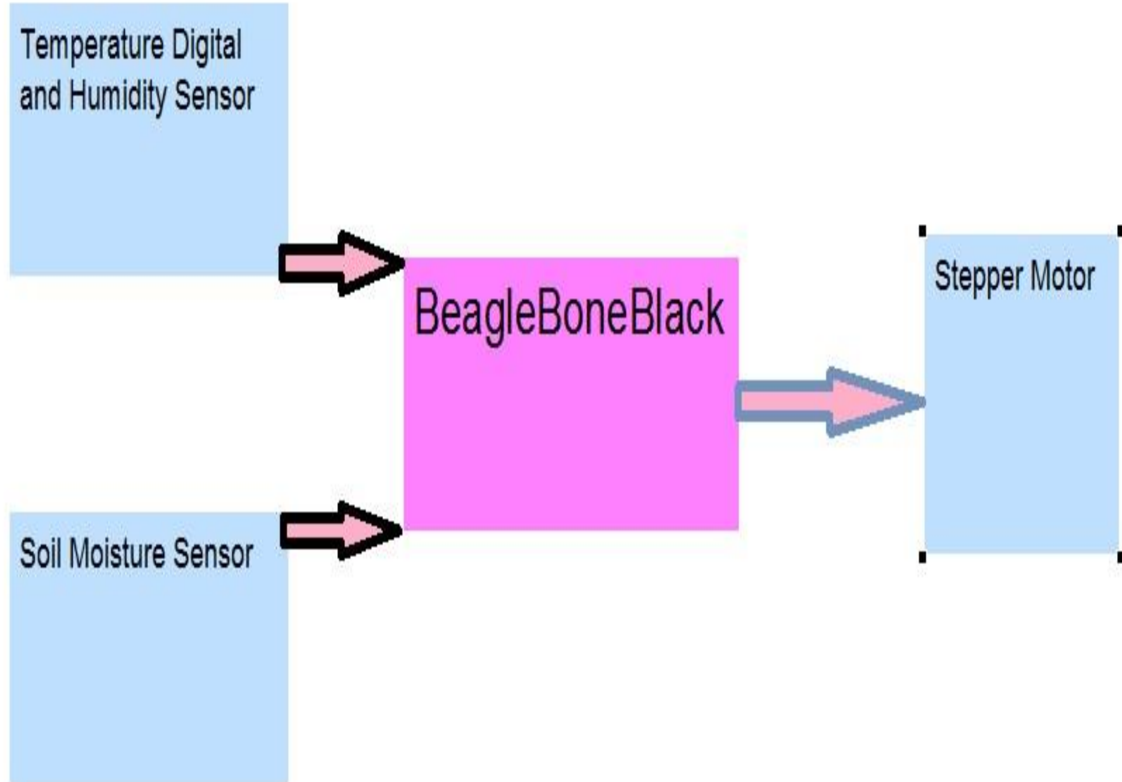


Figure 28 – System Layout

In the heart of the systems lies our board BBB, it has two inputs the Temperature and humidity sensor (THS) and Soil Moisture sensor.

It has one output: a stepper motor which used to open the hydraulic valve to irrigate the soil.

One great feature of the BBB is that it has a build in Ethernet port and a full network capability. We can use this feature to remote monitor our system without human intervention.

When we deploy the system in a farm or a greenhouse for irrigation, we will achieve a great savings in resources especially water, moreover we can irrigate many greenhouse farms at the same time without human intervention.

3-3-Component cost

We can buy most of the electronic component for our system from electronics shop or from online Retailers.

I have ordered the component from e-bay for convenience. The listed prices as it is on e-bay website.

3-3-1- 5V Stepper Motor 28BYJ-48 with Drive Test Module Board ULN2003 5 Line 4 Phase

Price: US $2.19

This is a 5v Stepper Motor with Gear Reduction, so it has good torque for its size, but relatively slow motion.

We can use the stepper motor to open a valve of water and close it.

3-3-2- TMP36 Temperature Sensor

Price: US $2.00

These sensors use a solid-state technique to determine the temperature. That is to say, they don't use mercury (like old thermometers), bimetallic strips (like in some home thermometers or stoves), nor do they use thermistors (temperature sensitive resistors). Instead, they use the fact as temperature increases, the voltage across a diode increases at a known rate. (Technically, this is actually the voltage drop between the base and emitter - the Vbe - of a transistor.) By precisely amplifying the voltage change, it is easy to generate an analog signal that is directly proportional to temperature. There have been some improvements on the technique but, essentially that is how temperature is measured.

- Temperature range: -40°C to 150°C / -40°F to 302°F
- Output range: 0.1V (-40°C) to 2.0V (150°C) but accuracy decreases after 125°C
- Power supply: 2.7V to 5.5V only, 0.05 mA current draw

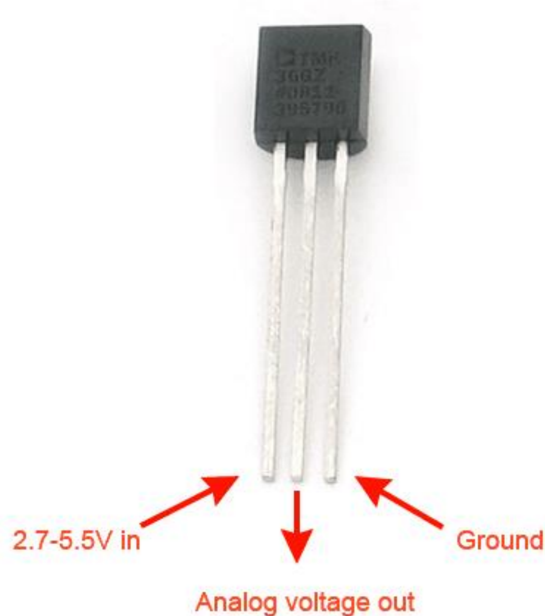


Figure 29 – Temperature Sensor

3-3-3- Soil Moisture Sensor

Price: US $1.51

- This is a simple water sensor, can be used to detect soil moisture.
- Module Output is high level when the soil moisture deficit or output is low.
- Can be used in module plant watered device, and the plants in your garden no need people to manage.
- Operating voltage: 3.3V~5V.
- Dual output mode, analog output more accurate.

3-3-4-The BeagleBoneBlack

You can buy the BBB for 45$, but if you want a cheaper solution, you can use Raspberry PI, it costs only 35$ and it similar to BBB.

3-4-Setting up IO Python Library on BeagleBoneBlack

The BeagleBoneBlack is unique in that it has quite a few pins that are available on easy to use pin headers, as well as being a fairly powerful little system. There are 2 x 46 pins available to use.

Some of the functionality that is available:

- 7 Analog Pins
- 65 Digital Pins at 3.3V
- 2x I2C
- 2x SPI
- 2x CAN Bus
- 4 Timers
- 4x UART
- 8x PWM
- A/D Converter

Accessing these pins requires low level C programming which is hard and prone to errors, luckily we can use a python library to access these pins from a higher level language. [27]

Now that you're connected to the BBB, you'll want to start with setting the date and time so that it's accurate. Copy and paste the following into your terminal.

#sudo ntpdate pool.ntp.org

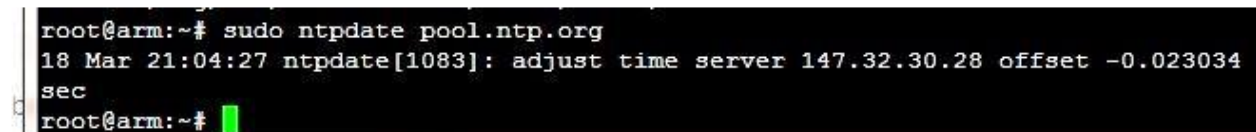


Figure 30 – Adjust time

Next install the dependencies:

#sudo apt-get update

#sudo apt-get install build-essential python-dev python-setuptools python-pip python-smbus –y

execute the command to install BBIO:

#sudo pip install Adafruit_BBIO

3-5-Connecting TMP36 Temperature Sensor

Using the TMP36 is easy; from the datasheet simply connect the left pin to power (2.7-5.5V) and the right pin to ground. Then the middle pin will have an analog voltage that is directly proportional (linear) to the temperature. The analog voltage is independent of the power supply.

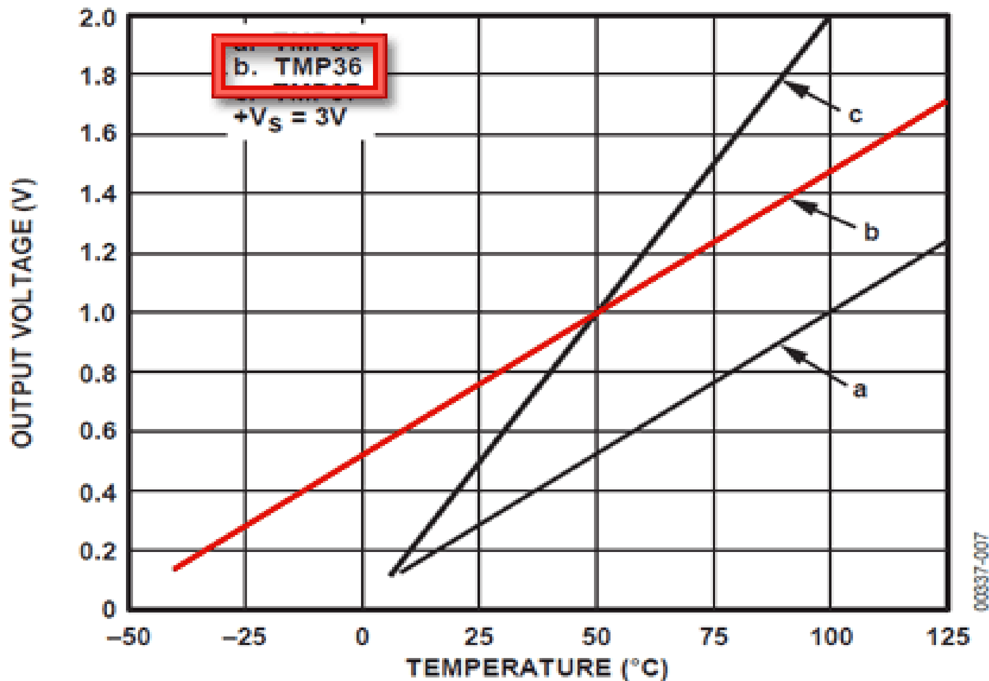


Figure 31 – Voltage Vs Temperature

To convert the voltage to temperature, simply use the basic formula:

Temp in °C = [(Vout in mV) - 500] / 10

So for example, if the voltage out is 1V that means that the temperature is ((1000 mV - 500) / 10) = 50 °C

3-5-1-Tools needed

We will need some tools to do the connections:

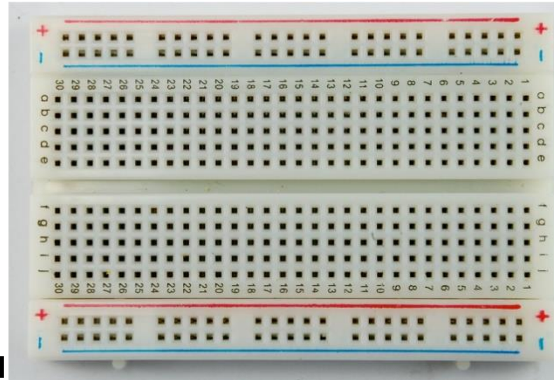
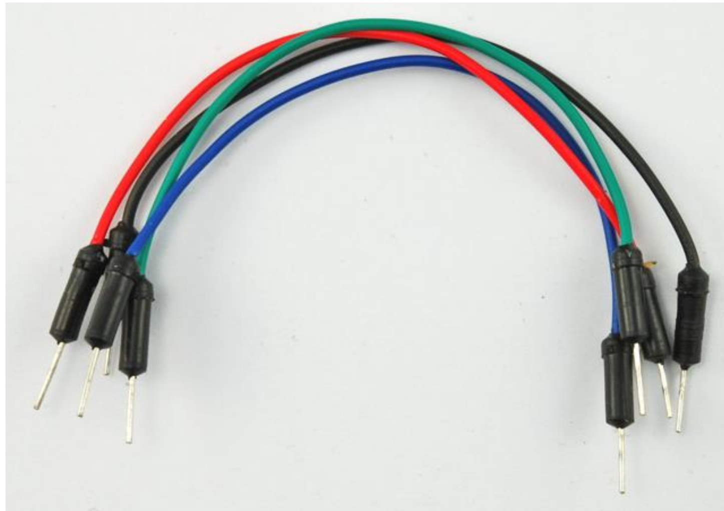


Figure 32 - Half-sized Breadboard

Figure 33 - Male to Male Jumpers



3-5-2-Wiring

Wire up the solderless breadboard using the header leads as shown below.

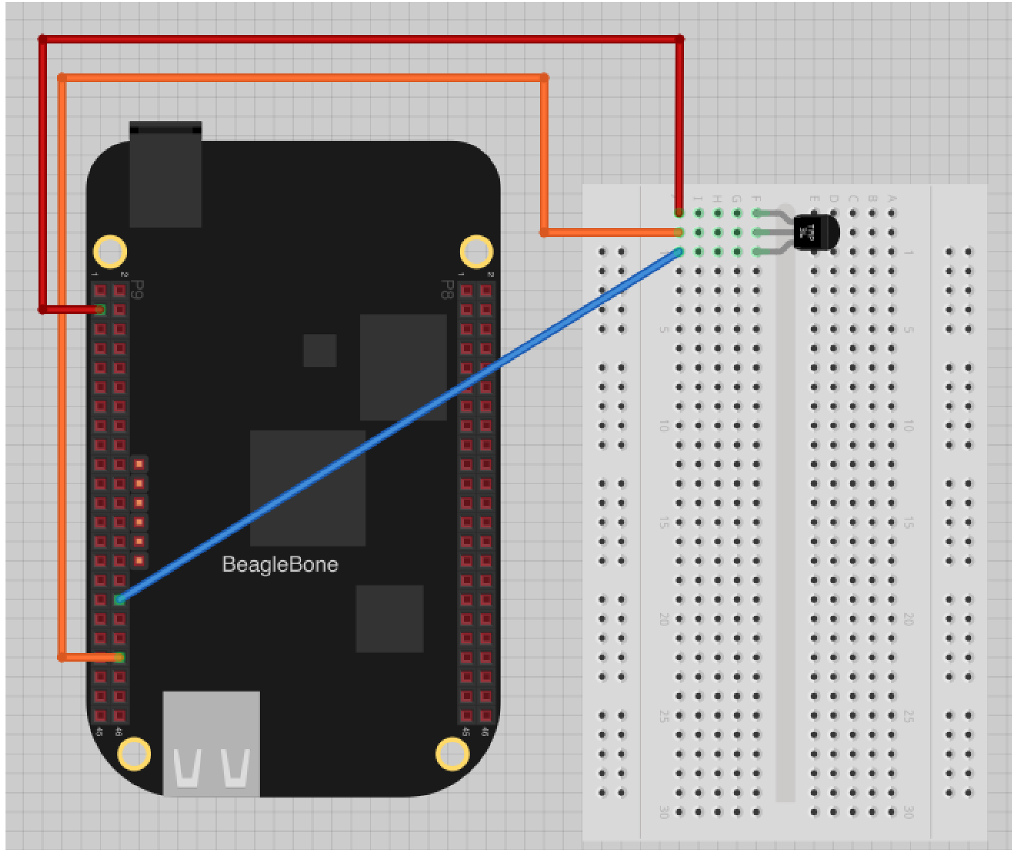


Figure 34 – Wiring Temp Sensor

The blue lead is connected from the GNDA_ADC connection to the GND pin of the TMP36 temperature sensor. The red lead is connected from pin 3 of the other connector (3.3V) to the positive supply pin of the TMP36 and the orange lead to pin P9.40.

3-5-3-Writing a program to read temperature

Enter the following command to create a new file called tmp36.py

#nano tmp36.py

We then write the following code:

```
import Adafruit_BBIO.ADC as ADC
import time

sensor_pin = 'P9_40'

ADC.setup()

while True:
    reading = ADC.read(sensor_pin)
    millivolts = reading * 1800  # 1.8V reference = 1800 mV
    temp_c = (millivolts - 500) / 10
    temp_f = (temp_c * 9/5) + 32
    print('mv=%d C=%d F=%d' % (millivolts, temp_c, temp_f))
    time.sleep(1)
```

Figure 35 – Read Temp Code

To start the program, enter the command:

mv=757 C=25 F=78

mv=760 C=26 F=78

mv=762 C=26 F=79

mv=765 C=26 F=79

mv=763 C=26 F=79

mv=763 C=26 F=79

mv=766 C=26 F=79

mv=768 C=26 F=80

3-6-Connecting the stepper motor

We will use the stepper motor to open and close a water pump nozzle, so we can irrigate the soil.

A stepper motor is an electromechanical device which converts electrical pulses into discrete mechanical movements. The shaft or spindle of a stepper motor rotates in discrete step increments when electrical command pulses are applied to it in the proper sequence. The motors rotation has several direct relationships to these applied input pulses. The sequence of the applied pulses is directly related to the direction of motor shafts rotation. The speed of the motor shafts rotation is directly related to the frequency of the input pulses and the length of rotation is directly related to the number of input pulses applied. One of the most significant advantages of a stepper motor is its ability to be accurately controlled in an open loop system. Open loop control means no feedback information about position is needed. This type of control eliminates the need for expensive sensing and feedback devices such as optical encoders. Your position is known simply by keeping track of the input step pulses.

The rotation angle of the motor is proportional to the input pulse. That's mean we can control the flow of water by controlling the aperture of the nozzle.

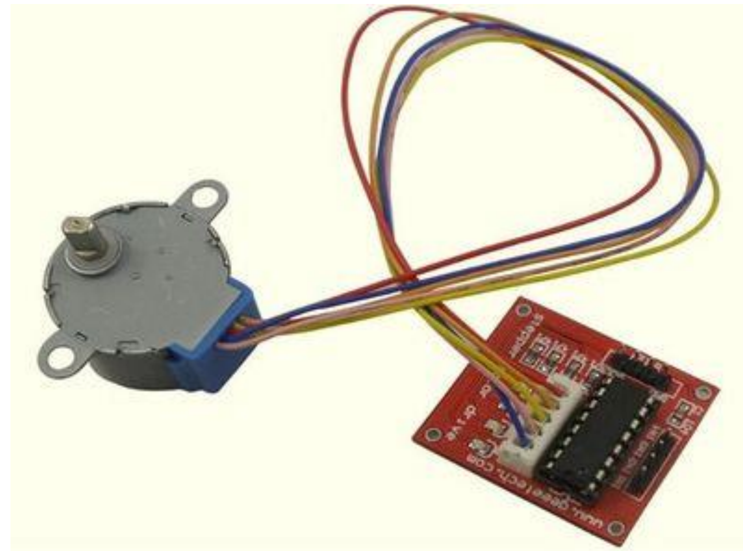


Figure 36 – Stepper Motor

3-6-1-Wiring Diagram

The simplest way of interfacing a stepper to BBB is to use a breakout for ULN2003 transistor array chip. The ULN2003 contains seven Darlington transistor drivers and is somewhat like having seven TIP120 transistors all in one package. The ULN2003 can pass up to 500 mA per channel and has an internal voltage drop of about 1V when on. It also contains internal clamp

diodes to dissipate voltage spikes when driving inductive loads. To control the stepper, apply voltage to each of the coils in a specific sequence.

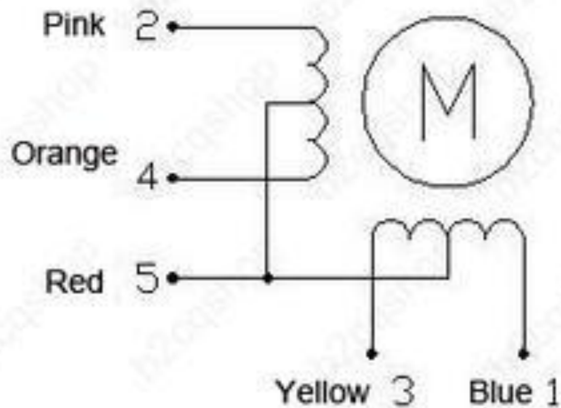


Figure 37 – Stepper Motor Wiring Diagram

From the BBB datasheet, we know that there are many pins for GPIO we can use

1. P8_14
2. P8_15
3. P8_16
4. P8_17

We will connect these four pins to the four input pins of ULN2003, Also, The + and - pins near "5-12V" need to be connected: - to BBB Ground, + to BBB +5 (for one motor test only) or (best) to a separate +5V 1A power supply.

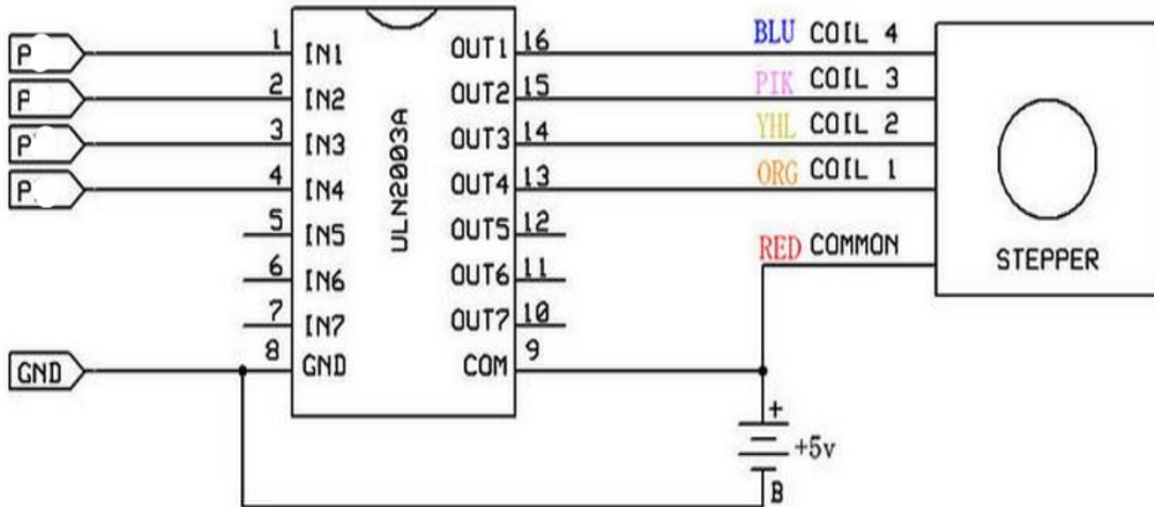


Figure 38 – Stepper Motor Control Board

3-6-2-Step sequences

The motor moves in response to the sequence in which the internal electromagnets are turned on. There are two possible sequences. In 4-step, there are always 2 of the 4 magnet coils turned on, and only one coil changes each step.

4 Step Sequence : AB-BC-CD-DA

The 8-step sequence uses only 1 coil on, then 2, then 1... etc

8 Step : A - AB - B - BC - C - CD - D - DA – A

3-6-3-Python code

We will write a python code to drive the stepper motor four steps, we can make more steps depending on the aperture we want for water flow from the nozzle.

Enter the following command to create a new files called step.py

#nano step.py

DRIVER LED LETTER	MOTOR CABLE#	MOTOR WIRE COLOR	step	step	step	step	step	step	step	step
4-STEP SEQUENCE				1		2		3		4
8-STEP SEQUENCE			1	2	3	4	5	6	7	8
	5	red	+	+	+	+	+	+	+	+
D	4	orange	0	0	0	0	0	1	1	1
C	3	yellow	0	0	0	1	1	1	0	0
B	2	pink	0	1	1	1	0	0	0	0
A	1	blue	1	1	0	0	0	0	0	1

Figure 39 – Stepper Motor Input Sequence

Write the following code in the editor; notice that any line start with # is a comment

```
import Adafruit_BBIO.GPIO as GPIO
# set the direction of pins as output
GPIO.setup("P8_14", GPIO.OUT)
GPIO.setup("P8_15", GPIO.OUT)
GPIO.setup("P8_16", GPIO.OUT)
GPIO.setup("P8_17", GPIO.OUT)
```

```
# we enter the sequence from here
GPIO.output("P8_14", GPIO.HIGH)
GPIO.output("P8_15", GPIO.HIGH)

GPIO.output("P8_14", GPIO.LOW)
GPIO.output("P8_16", GPIO.HIGH)

GPIO.output("P8_15", GPIO.LOW)
GPIO.output("P8_17", GPIO.HIGH)

GPIO.output("P8_16", GPIO.LOW)
GPIO.output("P8_14", GPIO.HIGH)

GPIO.cleanup()
```

To start the program, enter the command:

```
# python step.py
```

3-7- Connecting the Soil Moisture Sensor

It is a simple water sensor; it can be used to detect soil moisture. Module Output is high level when the soil moisture deficit or output is low. It can be used in module plant water device.

3-7-1-Interface description

VCC: 3.3v-5v

GND: GND

DO: Digital output interface (0 and 1)

AO: Analog output interface

Figure 40 – Soil moisture sensor

3-7-2- Instructions for use

- Soil moisture module is most sensitive to the ambient humidity is generally used to detect the moisture content of the soil.
- Module to reach the threshold value is set in the soil moisture, DO port output high, when the soil humidity exceeds a set threshold value, the module D0 output low;
- The digital output D0 can be connected directly with the microcontroller to detect high and low by the microcontroller to detect soil moisture;
- The digital outputs DO shop relay module can directly drive the buzzer module, which can form a soil moisture alarm equipment;
- Analog output AO and AD module connected through the AD converter, you can get more precise values of soil moisture;

We can use the analog or digital output from the module, analog is more accurate, but in our demonstration we will use the digital output.

We will connect P8_12 of the BBB to the DO port of the sensor.

We can calibrate the threshold point with the blue switch on the control board

Control threshold point

Figure 41 – Soil moisture Control threshold point

3-7-3-Python code

Enter the following command to create a new file called soil.py

#nano soil.py

Write the following code in the editor; notice that any line start with # is a comment

```
import Adafruit_BBIO.GPIO as GPIO

# set the direction of pins as output
 GPIO.setup("P8_13", GPIO.IN)

# we Detect the state here

if GPIO.input("P8_14"):
    print("HIGH the soil is full of water")
else:
```

```
    print("LOW the soil need irrigation")

GPIO.cleanup()
```

To start the program, enter the command:

python soil.py

3-8-The final scenario

the system now can drive the motor to control the water flow for irrigation, anyone can access the system remotely and give instructions to the computer, all you need to know is the IP address of your network.

We can even automate the process more. For example, we can run a scheduling task on the system to irrigate the farm every day for 20 minutes or we will not irrigate if the heat sensor detects a cold weather, it depends about our plants type and the environment.

Conclusion

By deploying a grid of soil moisture sensors in the soil we can detect the amount of water in the soil and using the temperature sensor, we can increase the amount of water in case of hot weather. The stepper motor is used to control the flow of water to irrigate the field. This is done by rotating the Valve of the water nozzle. All of this is controlled by the BeagleBoneBlack computer which runs the free operating system Linux, and the whole system costs us a fraction of the commercial systems.

Linux is one of the best operating systems nowadays, and because its openness, it is easy to configure it for any application. It has all the features offered by commercial expensive systems and it cares about security in the internet of things world. It is ready for the connected devices world of the future. As I demonstrated the configurations of the systems, it is clear that anyone can adapt Linux into their organization seamlessly.

It was thought that the field of IT is only important in the corporate and business world, but by demonstrating this application, it is clear that computer are continuing to invade our life and be part of it to help us find a better and more economical solutions for our great problems like water scarcity and by adopting them more in our life, we will save our resources.

References

1-Pro Linux Embedded Systems Copyright © 2010 by Gene Sally ISBN-13 (pbk): 978-1-4302-7227-4

2-Embedded Linux Primer, Second Edition A Practical, Real-World Approach Copyright © 2011 Pearson Education, Inc. ISBN-13: 978-0-137-01783-6

3-Getting Started With BeagleBone by Matt Richardson Copyright © 2014 Awesome Button Studios, LLC. All rights reserved. ISBN: 978-1-449-34537-2

4- http://arstechnica.com/information-technology/2013/04/for-your-robot-building-needs-the-45-beaglebone-linux-pc-goes-on-sale/

5- http://beagleboard.org/Products/BeagleBone+Black

6- http://makezine.com/magazine/how-to-choose-the-right-platform-raspberry-pi-or-beaglebone-black/

7- The Concise Handbook Of Linux for Embedded Real-Time Systems TimeSys Corporation ©2002 TimeSys Corporation Pittsburgh, PA

8- http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler

9- http://www.barrgroup.com/Embedded-Systems/Glossary-E Neutrino Technical Library. Retrieved 2007-04-21

10- Embedded Systems Design Second edition ISBN 0 7506 5546 1

11- Programming Embedded Systems By Michael Barr, Anthony Massa ISBN: 0-596-00983-6

12- HYDRA - the Kernel of a Multiprocessor Operating System, W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, Communications of the ACM, Vol. 17, No. 6, June, 1974, pp. 337-345; reprinted in Distributed Computing: Concepts and Implementations edited by P.L. McEntire, J.G. O'Reilly, and R.E. Larson, IEEE Press, 1984

13- http://www.minix3.org/

14- Dan Hildebrand (1992). "An Architectural Overview of QNX". Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures: 113–126. ISBN 1-880446-42-1

15- http://www.raspberrypi.org/

16- http://www.raspberrypi.org/about

17-http://www.ti.com/product/am3359

18- http://kernelnewbies.org/FAQ/Preemption

19- BeagleBoneBlack System Reference Manual Revision A5.2 April 11, 2013

20- https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions

21- Dedicated Systems Encyclopedia. Free downloadable reports, 2002.

http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations.

22- Out of Water: From Abundance to Scarcity and How to Solve the World's Water Problems

ISBN-10: 0-13-136726-9

23- Snyder, R. L.; Melo-Abreu, J. P. (2005). "Frost protection: fundamentals, practice, and economics – Volume 1" . Food and Agriculture Organization of the United Nations. ISSN 1684-8241.

24- Williams, J. F.; S. R. Roberts, J. E. Hill, S. C. Scardaci, and G. Tibbits. "Managing Water for Weed Control in Rice". UC Davis, Department of Plant Sciences.

25- http://agriwaterpedia.info/wiki/Water_use_efficiency

26- http://www.fao.org/docrep/s2022e/s2022e07.htm

27-http://learn.adafruit.com/setting-up-io-python-library-on-beaglebone-black/installation-on-ubuntu

List of figures

Figure 1 – BeagleBoneBlack layout ……………………………………………………………………VI

Figure 2 - ARM SOC Architecture ……………………………………………………………………3

Figure 3 - Kernel_Layout ………………………………………………………………………………4

Figure 4 –Monolithic Kernel Vs MicroKernel ………………………………………………………5

Figure 5 – Hybrid Kernel ………………………………………………………………………………6

Figure 6 – Raspberry PI ………………………………………………………………………………11

Figure 7 – BeagleBoneBlack …………………………………………………………………………12

Figure 8 – BeagleBoneBlack Vs Raspberry PI ……………………………………………………14

Figure 9 – Hard Vs Soft Real-time …………………………………………………………………16

Figure 10 – Interrupt to Process Latency …………………………………………………………17

Figure 11 – Anatomy of BBB …………………………………………………………………………20

Figure 12 – ARM Sitara Architecture ………………………………………………………………21

Figure 13 – Linux Lap setup …………………………………………………………………………22

Figure 14 – cloning Github Repository ……………………………………………………………23

Figure 15 – Checkout source …………………………………………………………………………23

Figure 16 – clone the RTKernel source ……………………………………………………………24

Figure 17 – Kernel Feature ……………………………………………………………………………25

Figure 18 – Preemption Model ………………………………………………………………………26

Figure 19 – Fully Preemptible Kernel ………………………………………………………………26

Figure 20 – Kernel Hacking ……………………………………………………………………………27

Figure 21 – Tracers ………………………………………………………………………………………28

Figure 22 – Tracers options …………………………………………………………………………29

Figure 23 – Save new Configuration ………………………………………………………………29

Figure 24 – Get root file system ……………………………………………………………………30

Figure 25 – Extract files ………………………………………………………………………………31

Figure 26 – Logging using SSH ……………………………………………………………………31

Figure 27 – Kernel Name ……………………………………………………………………..32

Figure 28 – System Layout ……………………………………………………………………36

Figure 29 – Temperature Sensor …………………………………………………………..38

Figure 30 – Adjust time ………………………………………………………………………39

Figure 31 – Voltage Vs Temperature ………………………………………………………40

Figure 32 - Half-sized Breadboard ………………………………………………………41

Figure 33 - Male to Male Jumpers …………………………………………………………41

Figure 34 – Wiring Temp Sensor ……………………………………………………………42

Figure 35 – Read Temp Code …………………………………………………………………43

Figure 36 – Stepper Motor ……………………………………………………………………44

Figure 37 – Stepper Motor Wiring Diagram ………………………………………………45

Figure 38 – Stepper Motor Control Board ………………………………………………..46

Figure 39 – Stepper Motor Input Sequence ……………………………………………..47

Figure 40 – Soil moisture sensor ……………………………………………………………..49

Figure 41 – Soil moisture Control threshold point ……………………………………..50