University Of Pisa

Department of information Engineering

Master Thesis In Embedded Computing System

UNIVERSITÀ DI PISA

PORTING OF FREERTOS ON A PYTHON VIRTUAL MACHINE FOR EMBEDDED AND IOT DEVICES

Supervisor : Prof. Enzo Mingozzi
Co-Supervisor : Prof. Carlo Valatti
Ext-Supervisor : Dr. Daniele Mazzei

Student: Nisar Ahmad

Academic year: 2014-2015

To my Family in Pakistan,
To you, Afsheen

Abstract

The fourth industrial revolution, *The Industry 4.0*, puts emphasis on the need of "*Smart*" and "*Connected*" objects through the use of services provided by Internet of Things, cyber-physical systems and cloud computing to optimize the cost, development time and remote connectivity. Development of highly scalable and flexible IoT applications is the need of time. These solutions require connectivity, less development time, time-to-market and at the same time offers a high performance and great reliability. Zerynth, a small company, provides its full stack for IoT solutions. Zerynth Virtual Machine is the core component among other components in stack which allow the programmers to code in python or hybrid C/Python coding with multithreaded Real Time OS with negligible memory footprint. The Python layer, *Application Layer,* is totally agnostic of underlying RTOS and hardware abstraction layer. This layered software architecture of Zerynth VM makes it totally compatible with new Industry 4.0 standard. The Hardware abstraction layer, *VHAL,* abstracts the hardware features of supported MCU and its peripherals while RTOS layer, VOSAL, uses the features of underlying Real Time OS. Zerynth VM can be ported with different Real Time OS and various hardware platforms depending upon the application's cost, features and other relevant parameters. Configuring Kinetis MCU (MK64FN1M0VDC12) with existing VM became the first objective of this thesis. This configuration covers from scratch the clock, boot loading and peripheral support. Since previous version of Zerynth VM had a support of only Chibi2 OS which has certain dependency on the hardware layer underneath so this became another objective to separate the Chibi2 OS from VHAL layer for total independence. Finally, Porting of FreeRTOS on Zerynth VM with Hexiwear MCU as target board could a make a room for another RTOS hence enhancing the features and support of currently available VM. This thesis report describes all porting steps, procedures and testing methodologies starting from configuring a new hardware platform Hexiwear to FreeRTOS porting on Zerynth VM.

Contents

1. INTRODUCTION 6

1.1 Embedded Software Architecture 6
1.2 Types of Software Architectures 6
1.2.1 Static Architecture 6
1.2.2 Dynamic Architecture 6
1.3 Real Time Systems 6
1.3.1 Need for Real Time System 7
1.3.2 Tasks and Task State Model 7
1.3.3 Context Switching 8
1.3.4 Multitasking 8
1.3.5 Preemptive and Non-Preemptive Scheduling 8
1.4 Layered Software Architecture 9
1.4.1 Microcontroller Abstraction layer 10
1.4.2 Hardware Abstraction Layer 11
1.4.3 RTOS Layer 11
1.4.4 Application Software Layer 11
1.5 Zerynth Software Architecture 11
1.6 Zerynth Stack 11
1.6.1 Zerynth Virtual Machine 12
1.6.2 Zerynth Studio 13
1.6.3 Zerynth Connector 14
1.6.4 Zerynth API 14
1.6.5 Zerynth App 14
1.6.6 Zerynth Toolchain 14
1.7 Zerynth and Python 14

2. THESIS OBJECTIVES 16

2.1 Zerynth Virtual Machine Current Configuration 16
2.1.1 Zerynth VM Platfrom Support 16
2.1.2 Zerynth VM VHAL layer 16
2.1.3 Zerynth VM VOSAL Layer 17
2.1.4 Zerynth VM Application Layer 18
2.2 Decoupling Chibi OS from VHAL layer 18
2.3 Porting of FreeRTOS to Kinetis MK64FN1M0VDC12 19
2.4 Main Approach 19

3. DECOUPLING CHIBIOS FROM VHAL 21

3.1 MK64FN1M0VDC12 (Hexiwear) as Porting Platform 21
3.1.1 Industrie 4.0 21
3.1.2 Over the Air Firmware Update 22
3.1.3 Haptic Feedback Support 22
3.1.4 Features and Sensors Support 22
3.1.5 DAPLink Firmware Update Support 22
3.2 Setup for Development Environment in Windows 7 22
3.2.1 Make for Windows 22
3.2.2 STM32 Toolchain for ARM GCC 23
3.2.3 Cygwin64 Tools 23
3.2.4 GITLAB support 23
3.2.4 Debug settings 24
3.2.5 OpenOCD Flash Programing 24
3.3 Configuration of Minimum Viable Zerynth Virtual Machine 25
3.4 BootUp Configuration for MK64FN1M0VDC12 25
3.4.1 Clock Configuration 25

3.4.2 Linker Script Configuration *26*
3.4.3 Vector Table Configuration *26*
3.4.4 MakeFile and Board Specific Configurations *27*
3.5 Implementation of VHAL Drivers 28
3.5.1 VHAL GPIO Drivers for K64 MCU *28*
3.5.2 VHAL Flash Drivers *29*
3.5.3 VHAL Serial Drivers *30*

4. PORTING FREERTOS ON ZERYNTH VIRTUAL MACHINE 32

4.1 FreeRTOS Basics 32
4.1.1 Why choose FreeRTOS? *32*
4.1.2 FreeRTOS Directory Structure and Data Types *32*
4.2 Tasks and Co-Routines 34
4.2.1 Tasks *34*
4.2.2 Task States *34*
4.2.3 Tasks Priorities *35*
4.2.4 Co-Routines *35*
4.3 Queues, Mutexes and Semaphores 36
4.3.1 Queues *36*
4.3.2 Mutexes *37*
4.3.3 Semaphores *38*
4.4 Software Timers 40
4.4.1 Software Timer Callback Functions *40*
4.4.2 Timer Period *40*
4.4.3 Software Timer States *41*
4.5 General Porting Steps 42
4.5.1 Porting of FreeRTOS Suggested Steps *42*
4.5.2 FreeRTOS Configuration *42*
4.5.3 Memory Management *45*
4.6 Porting of FreeRTOS to MK64FN1M0VDC12 (Hexiwear) 46
4.6.1 Cortex-M4 FreeRTOS port specific configuration *46*
4.6.2 FreeRTOSConfig.h and Port.c Configuration *47*
4.6.3 Static Memory Allocation by Garbage Collector *47*
4.6.4 Uplinker support *48*

5. TESTING AND RESULTS 49

5.1 Testing Methodology 49
5.1.1 Test Criteria Specification *49*
5.1.2 Test Case for Context Switch Time on Hexiwear with C *50*
5.1.2 Test Case for Context Switch Time on Hexiwear with Python *51*
5.1.4 Test Case for Interrupt Latency Testing Using C *51*
5.1.5 Test Case for Sleep Time Analysis *51*
5.2 Results 52
5.2.1 Performance Results of Context Switch Time with Both RTOS *52*
5.2.2 Interrupt Latency Time results *54*
5.2.3 Performance Results with Sleep Time results *55*
Appendix A 59

6. BIBLIOGRAPHY 63

Chapter 1

1. Introduction

1.1 Embedded Software Architecture

Code reusability has been much increased over the past few years. This has led engineers and scientists to develop new methodologies for software architectures. It's apparent that reusing code across software projects decreases project development time. Nonetheless, software engineers often resist developing reusable code because they're burdened by time-to-delivery commitments.

In an effort to reduce the development time of designing reusable software, adopting an architectural template that can be applied from project to project would be beneficial. The template would define hardware-independent reusable modules and an interface layer that is hardware dependent--changing when the hardware in the system changes. By applying the architecture template consistently across several program platforms, the goal would be to decrease the development time from one project to another while improving the maintainability of the software product.

1.2 Types of Software Architectures

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Software architecture involves the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability.

There in major two types for software architectures:

1.2.1 Static Architecture

This type of architecture describes which part of the software consists of. What are the externally visible features the software has? How the parts of software are connected to each other? This architecture is hierarchical with several levels. It covers the encapsulation reusability and quality of software.

1.2.2 Dynamic Architecture

This architecture covers the real time constraints and runtime efficiency. The structures of executable and allocation of run time to Tasks and processors. It also takes care of dynamic properties of tasks, priority and data dependencies.

1.3 Real Time Systems

A real-time system has to guarantee the externally generated stimuli within specified amount of time. System performance not only depends on the correct results but also the timeliness behavior of certain events are also has same significance. The Figure1-1 shows that Real time system mainly consists of Real Time kernel which has to manage the processes, multitasking, inter process communication, memory & resource management and interface between hardware and Application software. There are two types of Real-Time systems

- Soft-Real Time

In Soft real time systems if a process misses its deadline or completion within specified amount of time then it does not have any hazardous effects on all of the system performance except some delayed results will be obtained and in certain scenarios these effects are acceptable e.g. a video frame has some deadline of 30ms which means that video must have to be received within this amount of time and if video frame is not received within this specified time , the only effect will the degradation of video quality and video can be jerky. So this doesn't have any hazardous effect on overall system performance and the environment.

- Hard –Real Time Systems

In Hard-Real Time Systems if a process misses its deadline then system may undergo some changes which are catastrophic for overall system and its environment. For example, in aviation if engine thrust is not available within specified time or Navigation systems does not respond in time , then whole system may end up to very disastrous condition.

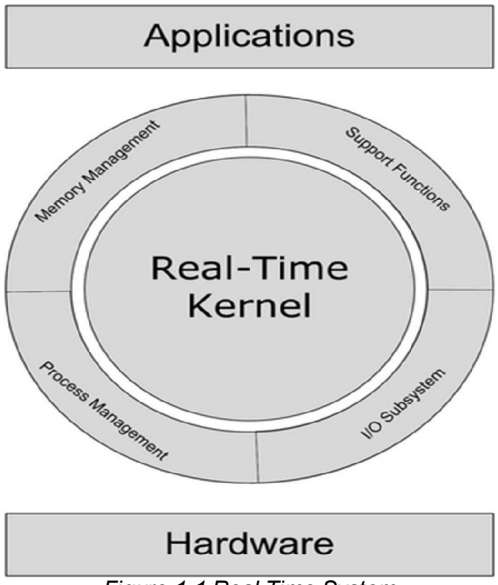


Figure 1-1 Real Time System

1.3.1 Need for Real Time System

Real-time operating systems (RTOSs) are often used to develop applications for systems with complex time and resource constraints. This situation often typifies laboratory automation where one or more computers must synchronize the activities among one or more instruments involving time, process, or precedent constraints. Time constraints might include actions such as "mix for at least x seconds" or "heat at 100 °C for 1 min". Process constraints condition activities, for example, "pick x and place at y" or "rotate 30°". In addition, precedent constraints such as "before", "during", "after", and their complements add further complexity to process control. Fortunately, RTOSs provide the necessary features to handle the demanding time, process, and precedent constraints often associated with such systems.

1.3.2 Tasks and Task State Model

Tasks are the basic functionalities that a real time system provides. Sometimes also knows as processes. Rea-time kernel is responsible to execute a task block or put in ready state. These states are determined based on different features, most importantly the priority of a task. The states of running tasks are shown in the following Figure 1-2 known as Task state Model. The tasks can be periodic, non-periodic or sporadic. Each task has its own memory stack, program counter and memory space to hold variables data.

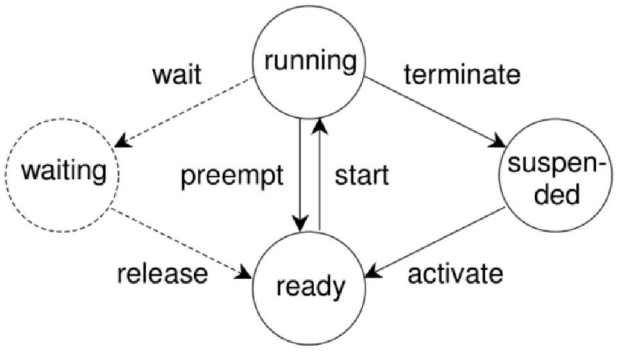


Figure 1-2 Task State Model

1.3.3 Context Switching

This is also a prime feature of Real-Time system. A running task can be suspended if a high priority task is ready to run. So real time kernel will put the current task into blocked queue by saving its current memory state and restore it when there are no higher priorities tasks are available to run. So a single CPU can be shared among many threads or Tasks. This phenomenon of saving and restoring the state of task or in other words putting a task into blocked state from running state or vice versa, is known as context switching.

1.3.4 Multitasking

The scheduler has a responsibility to select a particular task, based on algorithm, and put into the ready queue where the task has to wait until the current high priority task finishes its execution. In this fashion, real-time kernel can manage a number of tasks in pseudo parallel way so called Multitasking.

1.3.5 Preemptive and Non-Preemptive Scheduling

Preemptive as its name goes: interruption. Higher priority task can interrupt the running low priority task and context switch will happen in favor of higher priority task. If a scheduler allows this kind of feature then it is called *Preemptive Scheduler* otherwise in *Non-Preemptive Scheduling ,* the low priority task keep on running even if higher priority task is ready to run and context switch will only happen when the current thread will finish its execution. As shown in the following Figure 1-3 below;

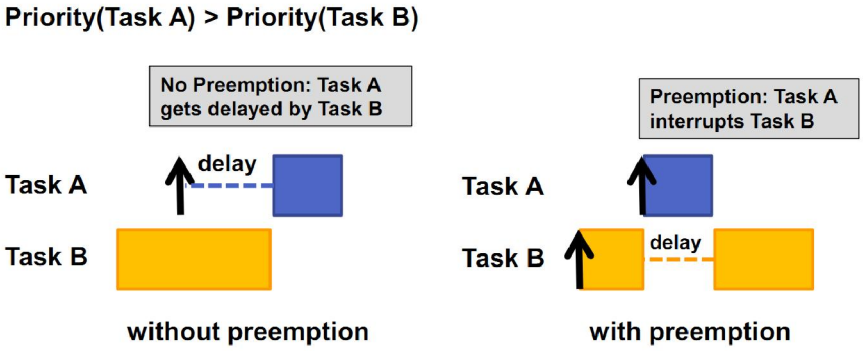


Figure 1-3 Preemptive and Non-Preemptive Scheduling

Task Synchronization and Resource Access

The tasks running in an RTOS may need to access the shared data, memory or files. Access to these resources can be given in a predefined way otherwise data corruption or false results can be obtained. Every RTOS provides some primitives to have this kind of synchronization. To illustrate this concept, let's take an example of a Bank account which is being accessed by two

persons simultaneously. One person want to withdraw some amount and other person wants to deposit some amount to the account. If these transactions are not done in a synchronized way then false updating of account may occur. Following Figure1-4 shows the correct transaction of money and account updating with true results. But if user 1 on ATM 1 has certain

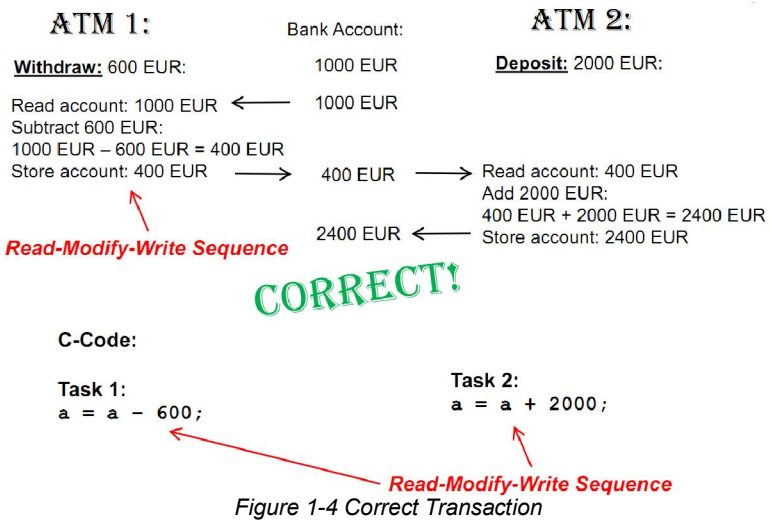


Figure 1-4 Correct Transaction

Delay after reading the account money and meanwhile user 2 on ATM 2 tries to read account for deposit it will read false money and final results will be incorrect. This situation is depicted in the following Figure 1-5.

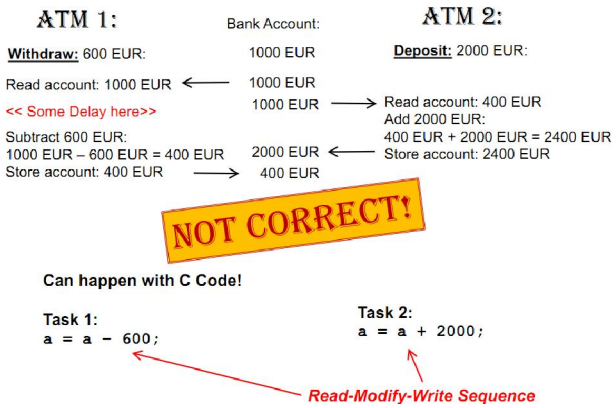


Figure 1-5 Incorrect Transaction

So RTOS provides this kind of security to access shared resource with certain *locking* mechanism. This mechanism is ensures that resource access is certain and avoids any kind of failure such as *Dead Lock and Priority Inversion*. A *Critical Section* is defined as a piece of code which cannot be interrupted when it's locked by one task. Other task may have access to that piece of code (shared data) only when it is unlocked by current task. These primitives include *Semaphores, Mutex and Mailboxes.*

1.4 Layered Software Architecture

The focus on software reusability and portability has enabled the engineers to define a new design architecture which is flexible, scalable, maintainable and easily testable. This architecture allows software partitioning into different layers based on principle of partitioning, which conforms to the following three rules:

- Abstraction

Abstraction is a view to look a particular side without going much into details of its implementation. This refers to look at problem with hiding all irrelevant details. At every step of software development phase it goes to different levels of abstraction. For example, at highest level just an application requirement are addressed and at lowest levels the source code is implemented on hardware to obtain the desired functionality. Abstraction inherently use *Information Hiding* concept which is very useful during testing and maintaining the software. Salient features of Information hiding are listed below;

- Low coupling
- Functional independence among components leads to fewer failures
- Higher quality Software is attained

- Modularity

Software is written such that bigger *functionality* split into smaller modules which can be tested and maintained individually. Each module has a specific data and function to perform dedicated functionality. Criteria for modularization is

- *Interfaces,* among modules should be minimal and well defined
- *Implementation,* details for each module is hidden from other modules in a software

- Reusability

Reusability of software helps a lot to reduce the development activity. For software to be reusable interfaces should be well defined and among different layers there should be least dependence and coupling. Modules can be written and tested independently and integrated with main software.

The layered software architecture has made it possible for developers to achieve functionality with different software components by just integrating them successfully with main software thus enabling same architecture portable to various platforms. This architecture consists of the following layers:

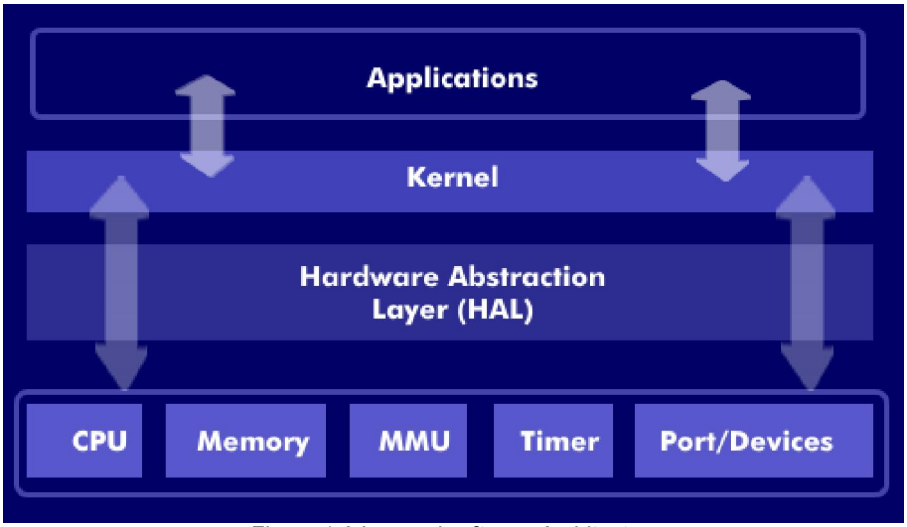


Figure 1-6 Layered software Architecture

1.4.1 Microcontroller Abstraction layer

MCAL is a software module used to access the physical hardware directly. The Hardware consists of MCU and some external devices. This layer implements the drivers for on-chip MCU peripherals and external devices. The Figure 1-6 shows that the lowest abstraction layer is MCAL. The peripherals consist of:

- Timers
- GPIO
- PWM
- Communication peripherals (SPI,I2C,USART,etc)
- ADC and DAC
- Memory

1.4.2 Hardware Abstraction Layer

Hardware abstraction layer is a division of code or providing an interface between physical hardware and software. It provides a direct access for OS to the hardware which is device independent.

Hardware abstraction layer is incorporated into numerous OSs to abstain from altering the OS which has an access to device independent hardware platform.

The HAL has the following benefits:

- It allows the applications to access the hardware with higher level of abstraction
- RTOS is able to access the hardware regardless of hardware architecture
- A generic approach for which makes the Software and Hardware independent of device architecture
- It is easy to port the main software to different MCU platforms.

1.4.3 RTOS Layer

An operating system abstraction layer (OSAL) or Real Time kernel provides an application programming interface (API) to an abstract operating system making it easier and quicker to develop code for multiple software or hardware platforms. It abstracts many hardware functions and provides them to applications in the form of services. Scheduling, files synchronization, and networking are the most common services provided by the OS.

OS abstraction layers deal with presenting an abstraction of the common system functionality that is offered by any Operating system by the means of providing meaningful and easy to use Wrapper functions that in turn encapsulate the system functions offered by the OS to which the code needs porting. A well designed OSAL provides implementations of an API for several real-time operating systems (such as vxWorks, FreeRTOS, Chibi, RTLinux etc.). Implementations may also be provided for non-real-time operating systems, allowing the abstracted software to be developed and tested in a developer friendly desktop environment.

In addition to the OS APIs, the OS Abstraction Layer project may also provide a hardware abstraction layer, designed to provide a portable interface to hardware devices such as memory, I/O ports, and non-volatile memory. To facilitate the use of these APIs, OSALs generally include a directory structure and set of makefiles that facilitate building a project for a particular OS and hardware platform.

Implementing projects using OSALs allows for development of portable embedded system software that is independent of a particular real-time operating system. It also allows for embedded system software to be developed and tested on desktop workstations, providing a shorter development and debug time.

1.4.4 Application Software Layer

This layer has the highest level of abstraction and it is located on the top of software hierarchy. It implements the system level functionality that stratifies the project goals. From the top view, all modules in this layer implement the system functionality. From a system perspective, each application is a separate OS process. Typically, applications run in the less-privileged processor mode and use the API system schedule provided by the OS to interact with the OS

1.5 Zerynth Software Architecture

Zerynth was born in 2015 to cater for the emerging needs in the area of *"Internet of Things"*. Zerynth provides the support to tailor existing objects to "*Smart Objects".* With multi-compatible cross platforms, cloud & mobile integration and high-level python script to embedded development, Zerynth enables the users to completely encompass the growing demands in the field of IoT and Machine-to-Machine (M2M) solutions.

The modular approach adopted in the development procedure will not only reduce the time-to-market of products and over all incurring expenses but also amplifies the potential for smart objects using python scripting.

1.6 Zerynth Stack

For complete IoT solution, *Zerynth* provides a full stack from firmware of multi-platforms of embedded devices to cloud integration and data visualization. The stack is flexible and modular giving a user complete freedom in selection of hardware platforms, Real Time systems and third

party cloud modules. The architecture described in the following Figure2-1 is the core of Zerynth stack.

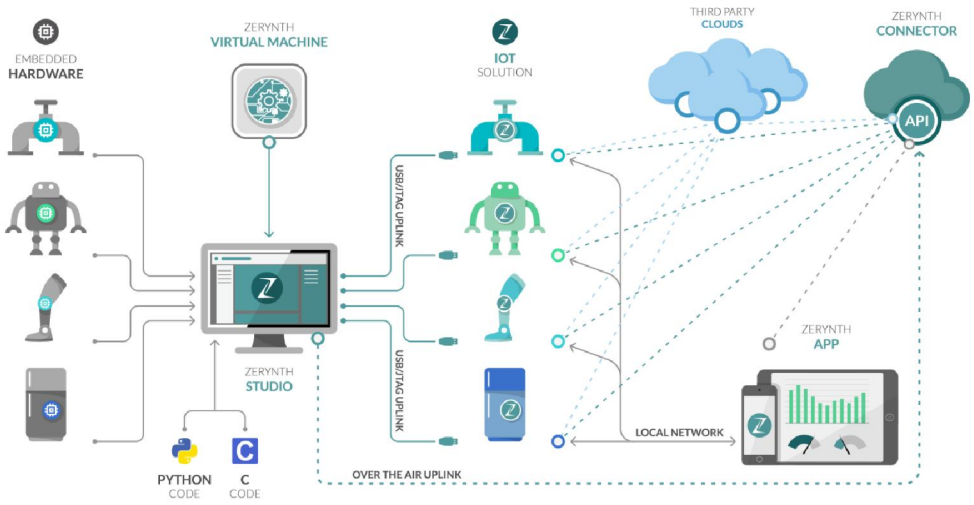


Figure 1-7 Zerynth Stack

Zerynth uses two terminologies for its supported devices:

- Internet Connected Devices

The device based on any MCU which has internet connection and implementing various communication protocols like MQTT, Coap etc., are called internet connected devices. These devices are generally equipped with sensors and actuators and can be programmed using high-level python script with Zerynth supported python libraries. They are used as gateways for non-internet connected devices.

- Non-internet Connected devices

These devices do not have internet connections but they use some other data connection protocols: ZigBee, Lora or Z-wave.

A brief description to every component in Zerynth stack is as follows:

1.6.1 Zerynth Virtual Machine

Zerynth VM comprises a set of Software and Hardware layers providing a quick design prototyping for any IoT solution. It contains a *Real Time Operating System* and Hardware platform to be accessed from higher layer based on python. Zerynth features multi-threaded environment. Software and hardware abstraction makes it easy for wide variety of applications. Zerynth VM is hardware agnostic, this feature enables the Zerynth VM hardware platform independent. Adding a python level application layer gives a comfort to the programmer to focus on functional requirements of project rather going deeper into details and debugging of lower level embedded development. It provides the support for network protocols like UDP and TCP and hence one can make the use of *http or https.* VM has separate cloud connector library written in python which act as unique communication interface for Zerynth enabled devices to connect any particular cloud. Hardware and RTOS independence makes Zerynth VM very flexible to any kind of IoT application with faster development time. Following Figure 2-2 shows block level view of Zerynth VM

The Zerynth VM can run Python scripts that are platform independent allowing a high reusability of code. Zerynth supports all the most used high-level features of Python like modules, classes, multithreading, callbacks, timers and exceptions, plus some hardware-related features like interrupts, PWM, digital I/O, etc.

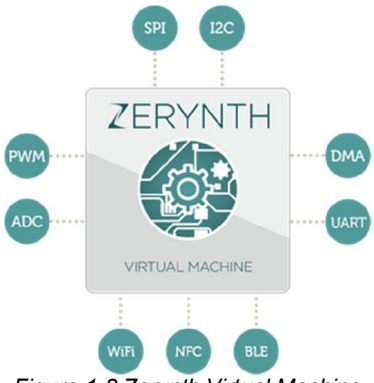


Figure 1-8 Zerynth Virtual Machine

The inner workings of the Zerynth VM are complex but can be reduced to a few components:

* **Bytecode Interpreter**: the Zerynth Compiler turns Python scripts to a set of bytecode objects, each one containing not only a sequence of instructions, but also enough information for memory management and error checking. Each bytecode instruction, called *opcode*, is exactly one byte in length with optional arguments going from 0 to 4 bytes. The bytecode interpreter simply scans the bytecode an opcode at time, executes the opcode in the current thread and continues to the next opcode until a stop is encountered. Zerynth bytecode closely resembles Python but introduces some embedded specific opcodes.

* **Global Interpreter Lock**: the GIL is an object shared by all Python threads; it coordinates the sequence of opcode execution between threads so that each opcode can be considered "atomic". This means that while thread-one is executing opcode "x", thread-one has the right to do so until the execution of "x" reaches the end. No other thread can stop it without compromising the interpreter integrity. When a Python thread goes to sleep, or its time quantum ends, the GIL is released so that another thread can take control of the bytecode interpreter.

* **Garbage Collector**: objects in Python have lifecycle. They are created and used by the programmer and must be removed when they are not needed anymore. While in low level languages the responsibility of freeing unused memory rests on the programmer, in Python it's the garbage collector (GC) duty. When necessary, a complete scan of the created object is performed in order to search the ones that can be removed safely. The VM GC algorithm is a mark-and-sweep-stop-the-world variant.

* **Interrupt Thread**: it is a very high priority thread that is woken up when an interrupt configured to run a Python function is fired. Python bytecode is executed outside of the ISR routine, but inside the Interrupt Thread. This way the Python function can allocate memory as a normal function. However, since the interrupt thread has the highest priority it is important to spend the least time possible inside it.

* **VOSAL**: it is the Zerynth VM operative system abstraction layer. It contains functions provided by the underlying RTOS to create threads, semaphores and other multithreading related objects. The VOSAL is linked into the VM, but many of its functions can be called from hybrid C/Python code.

* **VHAL**: it is the Zerynth VM hardware abstraction layer. It contains functions to control the microcontroller peripherals: serial ports, SPI, I2C, ADC, PWM and so on. Each family of microcontroller has its own VHAL implementation so that the programmer calling C from Python can have a uniform hardware API across different microcontrollers.

1.6.2 Zerynth Studio

Zerynth Studio is an IDE to manage Zerynth projects and virtualization of hardware platforms. This IDE is supported for Windows, Linux and Mac. User can develop the python projects in this IDE by selecting all supported boards and Operating Systems. Zerynth studio also has "*Package Manger*" for easy installation and management of libraries. All user applications developed in Zerynth studio can be uplinked to supported boards by JTAG or USB connections. It also gives a possibility to program "Over the air" using Zerynth Connector.

1.6.3 Zerynth Connector

Zerynth connector provides a frontend of a cloud for all connected devices to exchange data between a device or group of devices and cloud. Zerynth device manager supports bidirectional data transfer of TCP packets among multiple devices that are connected to the network. All of connected devices to local network can also be seen and also the devices which are connected remotely to cloud using Wi-Fi or mobile connection can also be explored.

1.6.4 Zerynth API

Zerynth API allows the end user to avail services provided by Zerynth connector. This API has a responsibility to ensure that only authorized requests are made to connected devices. Zerynth API can be used to access the other cloud services and makes the API independent of SAS (Software as Service) API or any other API. Zerynth API makes it possible to store the collected data for different devices on local database or selected cloud.

1.6.5 Zerynth App

Zerynth App provides an interface for all the connected Zerynth-enabled devices. This can be a smartphone or tablet App which can be downloaded and installed like all other mobile Apps. When App is running it automatically discovers all the objects that are connected to local network. When a particular device is selected this App becomes an interface for that and associates the current user to interact with Zerynth API. Zerynth App interface is written in HTML and that can be edited in IDE and can easily be integrated python script. So there is no separate need to write code for android or iOS.

1.6.6 Zerynth Toolchain

The Zerynth Toolchain (ZTC) is a command line tool that allows managing all the aspects of the typical Zerynth workflow.

Such workflow extends across different areas of the Zerynth programming experience:

- Managing projects
- Discovering, managing and virtualizing devices with virtual machines
- Compiling projects into executable byte code
- Uplinking byte code to virtualized devices
- Adding packages (e.g. libraries, drivers, device classes...) to the current installation
- Turn projects into libraries and publish them to the community repository

The workflow is made possible by the Zerynth backend that provides a set of REST API called by the ZTC. Therefore, most ZTC commands require an authentication token to act on the Zerynth backend on behalf of the user. Such token can be obtained by specific commands.

1.7 Zerynth and Python

The Zerynth VM has been developed with the goal of making Python usable in the IoT world. To do so some features of Python have been discarded because they were too resource intensive, while non-Python features have been introduced because they were more functional in the embedded setting. Here is a non-comprehensive list:

* Python Object size has been reduced as much as possible:

* Integers are signed and 31 bits wide, so that they can be represented with 4 bytes without additional overhead.

* Garbage collector overhead has been brought down to 8 bytes per object (and there is still space for optimization)

* Names are not saved as strings in the bytecode; they are converted to 16 bits integers to occupy much less space. This apparently minor change leads to a series of important consequences. First of all, Zerynth becomes a less "dynamic" language with respect to Python, since introspection is not allowed. However on the pro side, Zerynth scripts can be statically analyzed to remove unused bytecode greatly reducing memory usage. Another important consequence is that *setattr* and *getattr* cannot be used with non-constant arguments.

* Sequences and dictionaries can have at most 65536 elements.

* Exceptions have been transformed from full-fledged classes to a name organized in an inheritance tree. So an exception can't have methods, but it is faster to raise and to handle and it just takes 4 bytes of memory.

* Compilation has been moved outside the language; by removing the compile () and eval () builtins the VM shrink greatly in size.

* Not so often used Python features have been removed. Closures, generators and decorators will be added in future updates in a modular way.

* True multithreading with priorities has been introduced. CPython implementations use green-threads to emulate multithreaded environments without relying on any native OS capabilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support. In Zerynth each thread is a RTOS thread with its own memory and priority. Because of the GIL, only one Zerynth thread can execute bytecode in a time quantum, but it is possible to have more than one non-Python thread running in parallel. For example, a complex driver can be structured as a VOSAL thread written in C to control hardware, with any number of Zerynth threads running bytecode.

* New data structures have been introduced like shorts and short array to hold sequences of 16 bits integers. Big ints and fixed point math are in development.

Chapter 2

2. Thesis Objectives

2.1 Zerynth Virtual Machine Current Configuration

This chapter describes the current configuration of Zerynth Virtual machine, uncompliance with standard software archeitecture, proposed solution and development methodolgy to proceed. The Current configuration of Zerynth VM shown in Figure 2-1 consists of the three layers: the second and third layer are are shown at the same level which represents that *Virtual Software Abstraction Layer(VOSAL) and Virtual Hardware Abstraction Layer (VHAL)* has certain dependencies over each other. These dependecies will have to be removed and making the VOSAL and VHAL totally agnostic from Hardware and Real time Operation System.

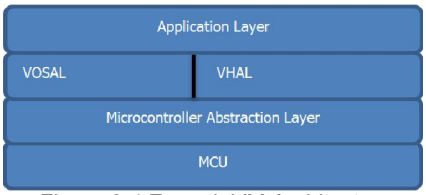


Figure 2-1 Zerynth VM Architecture

2.1.1 Zerynth VM Platfrom Support

Zerynth VM is hardware agnostic and it can support multiple hardware platforms. Currently supported devices are the following

- Arduino DUE
- Arduino/Genuino Zero
- Flip & Click Sam3X
- Particle Photon
- Quail Board

The *Microcontroller Abstraction Layer (MCAL)* is responsible to implement MCU specific drivers for each peripheral. This layer also contains Board support packages for selected hardware platform and has different implementation of each peripheral depending upon the MCU specific libraries provided by vendor.

2.1.2 Zerynth VM VHAL layer

The VHAL has generic implementation for all of supported MCUs. This layer is independent of underlying MCAL. It provides an interface between the higher software layer and the lower hardware layer. This layer provides following two features

- Pin Mapping

The VHAL introduces a distinction between **physical pins** and **virtual pins**. Physical pins are the actual pins available on the board and are defined in the file *port.c* for every supported board. A physical pin usually maps to a microcontroller register and offset, needed to drive the pin.

A virtual pin is just a name which refers to a particular configuration of a physical pin.

Therefore different virtual pins can map to the same physical pin. For example, imagine a board where the first physical pin (P0) can be used as a general purpose input output (GPIO) or as the clock line (SCL) of the first instance of the I2C bus. Such physical pin is always controlled by same microcontroller register, but in the first case it is configured as a GPIO and the

corresponding virtual pin name will be D0, while in the second case it is configured as I2C clock and the virtual pin name will be SCL0.

The internal representation of virtual pins is a 16 bit integer where the high byte represents the name of the peripheral (the “class” of the pin), while the low byte is the row number in the table of physical pins for that peripheral.

All these levels of indirection are hidden by the VHAL using macros to access the relevant information about pins. All VHAL functions requiring pin names expect virtual pin names.

The following table summarizes the virtual pin information:

Pin Class	Pin Offset	Pin Value	Pin Name
DIGITAL	0	0x0000	D0
DIGITAL	1	0x0001	D1
...	...	...	...
ANALOG	0	0x0100	A0
ANALOG	1	0x0101	A1
...	...	...	...
SPI	0	0x0200	MOSI0
SPI	1	0x0201	MISO0
SPI	2	0x0202	SCLK0
…	….	….	…
SER	0	0x700	RX0
SER	1	0x701	TX0
…	…	…	…
LED	0	0x900	LED0
LED	1	0x901	LED1
…	…	…	…

Table 2-1 Virtual Pin Information

Where *Pin Name* is a C Macro corresponding to *Pin Value*. For each string in *Pin Class* there exists a C macro with PINCLASS_ prepended, corresponding to the high byte of *Pin Value* (i.e. PINCLASS_DIGITAL is 0x00, PINCLASS_ANALOG is 0x01, etc...).

- Peripherals Mapping

Each microcontroller peripheral is mapped to a peripheral index in the board porting files. For each peripheral there exists a table mapping multiple peripheral instances of the same type to different indexes. The following example will clarify the mapping. Imagine a microcontroller with four different USART peripherals named USART1 to USART4. In the board porting each USART is mapped to a peripheral index by creating such table:

Index	Value
0	3
1	1
2	4
3	2

Table 2-2 VHAL Peripheral Mapping

When a VHAL function is called, expecting a peripheral index for a serial peripheral, the table is used to map the passed index (e.g. 0) to the corresponding mcu peripheral (e.g. USART3). Each board porting defines this kind of tables for each supported peripheral.

2.1.3 Zerynth VM VOSAL Layer

The Zerynth VM uses a common API to create and manage threads and synchronization objects. Such API is called VOSAL and abstracts the details of the underlying RTOS, so that it can be changed as needed for performance or licensing reasons.

Currently supported RTOS is chibi2.

VOSAL layer provides generic implementation for the following primitives of underlying RTOS:

- Threads
- Semaphores
- Mutexes
- Mailboxes

- System Timers

2.1.4 Zerynth VM Application Layer

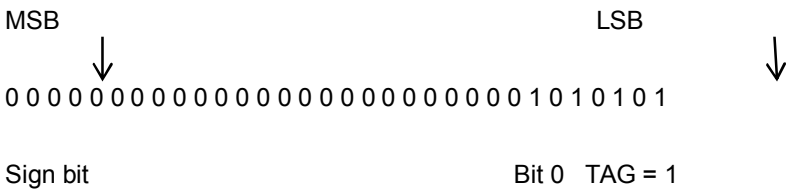
Application layer is written in python. User can write application code using Zerynth studio accessing supported python libraries. C functions called from Python can create and handle Python entities like lists, tuples, and dictionaries and so on. In the current version of Zerynth only a few selected Python data structures can be accessed from C.

Following is a brief introduction to supported functionalities:

- PObject

The VM treats every Python object as a pointer to a *PObject* structure. There exist two types of PObects: tagged and untagged. Tagged PObjects contains all the object information encoded in the 4 bytes of the pointer. Untagged objects are pointers to actual C structures. As a consequence, tagged PObjects are not allocated on the heap but reside on the stack of a frame of execution.

To better understand tagged PObjects imagine the case of integers: representing integers by allocating a PObject structure in memory is both a waste of ram and of computational power. Therefore small signed integers up to 31 bits are represented as tagged pointers. This “trick” is possible because a PObject pointer is 4 bytes (32 bits) and due to architecture constraints a valid PObject pointer is at least aligned to 2 or 4. In practical terms it means that the least significant bit of a valid PObject pointer is always 0: by “tagging” the PObject pointer, namely changing its lsb to 1, the VM is able to distinguish between concrete PObjects residing on the heap (untagged, lsb=0) and tagged PObjects (lsb=1). The representation of the number 42 as a tagged PObject follows:



Instead, an untagged PObject is a valid pointer to a C structure organized like this:

GCH: B0 B1 B2 B3 B4 B5 B6 B7

DATA: .....

Where GCH is an 8 byte header holding both garbage collection info and type/size info; DATA is whatever fields are needed to implement the PObject.

- Dictionaries and Sets

Some data structures in Python have functionalities similar to hash tables. In particular dictionaries are mappings from keys to values; set and frozen set are collections of items optimized to test the presence of a given item inside the set. Internally, the hash code of an item is calculated and used to find the item inside the structure in a fast way.

2.2 Decoupling Chibi OS from VHAL layer

It has been shown in the last sections that for Zerynth VM the VHAL and VOSAL layers, *shown at the same level*, are independent by design but by implementation there are certain dependencies. VOSAL layers abstract the features of underlying Chibi2 RTOS. So that layer by design and by implementation is correct. Since chibi2 RTOS provides itself some peripheral drivers, so VHAL is not totally independent from RTOS. Figure 2-2 clearly depicts the problem.

This task been assigned to remove this limitation and make the VM totally agnostic of hardware and Real Time Operating System.

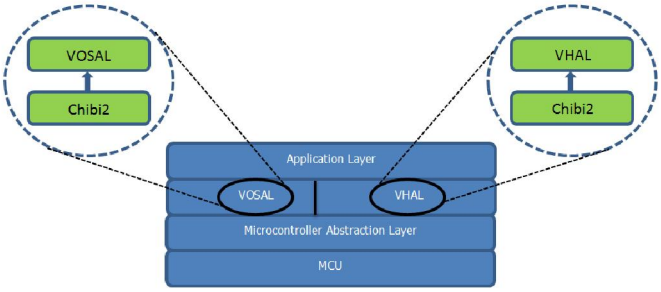


Figure 2-2 VOSAL and VHAL Dependencies

2.3 Porting of FreeRTOS to Kinetis MK64FN1M0VDC12

Hexiwear (based on Kinetic K64 MCU) and other on-chip sensors was the first board of its kind was chosen to extend the board support for Zerynth VM and to remove the first limitation of VHAL layer dependency on Real Time system for chibi2 OS.

It was also important to make Zerynth VM capable to support an RTOS other than chibi2 to enhance the scope of VM for various applications. Porting of FreeRTOS on Kinetis K64 MCU was second objective to be achieved. The description of each step that is taken to implement these two objectives will be described in detail in subsequent chapters, however, in the next section there is an explanation to the main approach and sequence of steps.

2.4 Main Approach

Porting of FreeRTOS on Hexiwear with Zerynth VM leads me to divide this thesis objectives in the following two parts:

Objective 1: Decoupling of ChibiOS from VHAL layer

Objective 2: Porting of FreeRTOS on Kinetis K64 MCU (Hexiwear)

- Objective 1: Decoupling of ChibiOS from VHAL layer

For accomplishment of objective 1 it is required to break down the approach into some sequence of steps. Following Figure 2-3 summarizes the flow of activities. The flow of activities for this objective are listed below:

i. Setup Development environment in Windows 7 by configuring all essential tools from Linux to windows
ii. Bootup Configuration of Hexiwear with existing RTOS i.e. chibi2. The bootup configuration includes all necessary changes to be made in Makefiles for VHAL layer, Clock configuration and Linker script configuration. The details of each setting will be covered in the chapter 3.
iii. Writing Code for minimum peripherals of K64 Platform required for *Minimum Viable Zerynth VM.* The peripherals include Serial, GPIO and Flash drivers.

- Objective 2: Porting of FreeRTOS on Kinetis K64 MCU (Hexiwear)

i. Downloading and understanding the kernel FreeRTOS 9.
ii. Configuring Systick and kernel priority
iii. Replacing the Chibi2 with FreeRTOS v9 by modifying the VOSAL layer

- Testing

In this step, when Zerynth VM has a support for both Operating Systems and with basic hardware peripherals needed, a devised test criteria for LED blinking time with each RTOS. Which makes sure that each RTOS working fine in VM accessing properly the underlying the hardware layer. Also to determine performance comparison in terms of time taken on RTOS primitives to synchronize the activities.

- Enabling Support Zerynth Studio

Finally enabling the support for Zerynth studio with FreeRTOS and Hexiwear. A user can write a code in python and choose hardware platform and RTOS.

Here is the flow of activities to fulfill above to objectives:

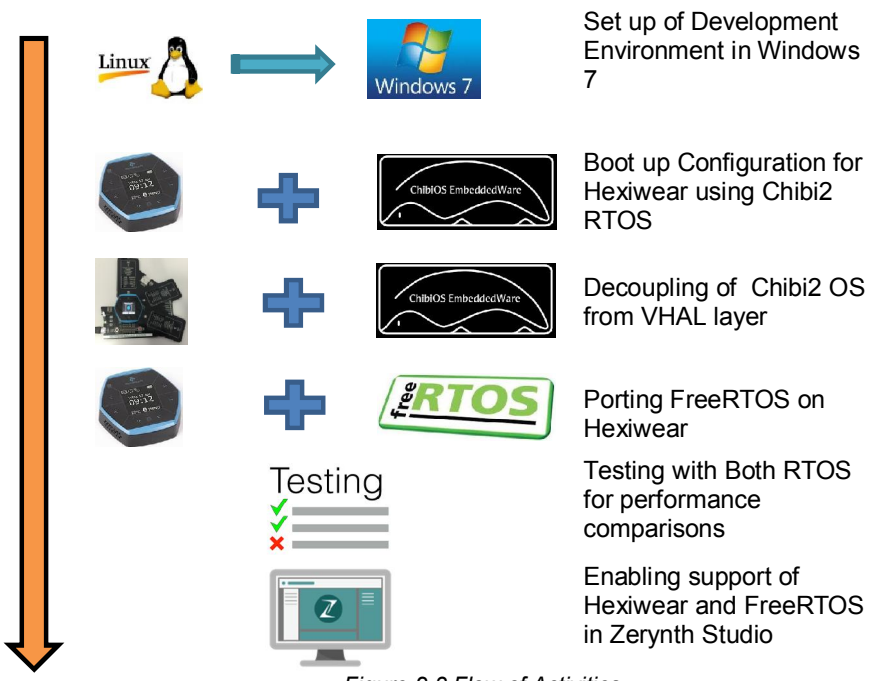


Figure 2-3 Flow of Activities

First three steps explains how to complete the Objective 1 and fourth step will be carried out to port FreeRTOS on K64 MCU. In the last step, enabling Zerynth support for Hexiwear and FreeRTOS.

Chapter 3

3. Decoupling ChibiOS from VHAL

3.1 MK64FN1M0VDC12 (Hexiwear) as Porting Platform

For porting purpose we have chosen the Hexiwear board based on Kinetis K64 MCU (Cortex M-4 core). Hexiwear provides complete IoT solution ranging from personal wearable watches to smart large scale industrial solution. The sleek and smart low power design equipped with sensors makes it an effective choice for IoT smart object. Following Figure 3-1 shows that Hexiwear can be used as wearable watch and its supported can be extended by mounting it on *Docking Station.*

Figure 3-1 Hexiwear

With three micro bus sockets, Micro-SD card, I2S interface extends the functionality of Hexiwear for large scale IoT applications. Following are few facts that describe the choice of Hexiwear for this Porting.

3.1.1 Industrie 4.0

Industry 4.0 is the current trend of automation and data exchange in manufacturing technologies. It includes cyber-physical systems, the Internet of things and cloud computing.

Industry 4.0 creates what has been called a "smart factory". Within the modular structured smart factories, cyber-physical systems monitor physical processes, create a virtual copy of the physical world and make decentralized decisions. Over the Internet of Things, cyber-physical systems communicate and cooperate with each other and with humans in real time, and via the Internet of Services, both internal and cross-organizational services are offered and used by participants of the value chain.

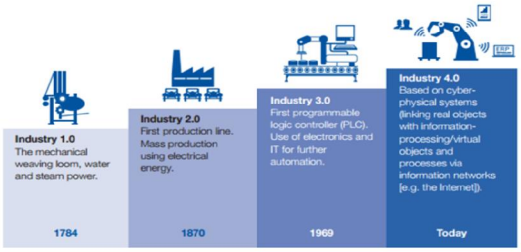


Figure 3-2 Industrie 4.0

Hexiwear is designed to make smart IoT solution which meets the need of Industrie 4.0.

3.1.2 Over the Air Firmware Update

Hexiwear support *Over the Air Programming* feature which is useful in IoT field when firmware update is required via internet. It supports two Microcontroller K64 (Main CPU) and KW40 (for BLE) interface.

3.1.3 Haptic Feedback Support

It has six capacitive touch buttons which can be used for haptics.

3.1.4 Features and Sensors Support

Featuring more than traditional MCU development platforms, the Hexiwear platform is ideal for connected applications, thanks to its power-efficient Kinetis K64F MCU featuring:

- ARM® Cortex®-M4 core;
- max frequency up to 120MHz;
- 1024KB Flash, 256KB RAM;
- many peripherals (16-bit ADCs, DAC, Timers);
- Interfaces (USB Device Crystal-less, UART, SPI, I2C, I2S, and SD-card).

Integration is also a key feature of Hexiwear, with a Bluetooth Low Energy (BLE) SoC (Kinetis KW40z) and 8 Sensors onboard:

- 6-axis Accelerometer and Magnetometer combo (FXOS8700CQ);
- 3-axis Gyroscope (FXAS21002CQ);
- Pressure sensor accurate up to Altitude sensing (MPL3115A2);
- Temperature and humidity combo (HTU21D);
- Ambient light sensor (TSL2561);
- Optical Heart rate sensor (Maxim MAX30101).

3.1.5 DAPLink Firmware Update Support

The Hexiwear docking station also integrates the openSDA adapter (with DAPLink firmware to support both MCUs), which enables virtual Serial, Flash-programming and several industry standard Debug interfaces, a JTAG debug connector for external probes.

3.2 Setup for Development Environment in Windows 7

It was important for developers to have both Linux and windows development environments. Sometimes it's more flexible to work in windows utilizing more graphical support for debugging and during other development activities so it was decided to work in windows. Zerynth VM compilation was totally done in Linux using ARM GCC compiler. It was an important task to have support of compilation equally in windows. So the following steps have been taken to configure Windows 7 for compiling code:

3.2.1 Make for Windows

Make is a tool to generate executable, bin files or non-source files from source files. It gets all the information of files from a file called MakeFile which has information of all non-source or output files from given source files.

GNU provides a support to have make files for windows. From the following link downloaded the make utility for windows.

http://gnuwin32.sourceforge.net/packages/make.htm

To use make command independent of any path in command window we have to add the path of make.exe and other related files in system environment variables.

Add make path in System environment variables for Windows 7 like

- From the desktop, right click the **Computer** icon.
- Choose **Properties** from the context menu.

- Click the Advanced system settings link.
- Click **Environment Variables**. In the section **System Variables**, find the PATH environment variable (*Shown highlighted in Figure 3-3)* and select it.
- Click on Path and edit the path. Every path is separated by semicolons (;)
- Go to end of all paths and insert a new path of make files, in this case it was the following path:

C:\Program Files (x86)\GnuWin32\bin;

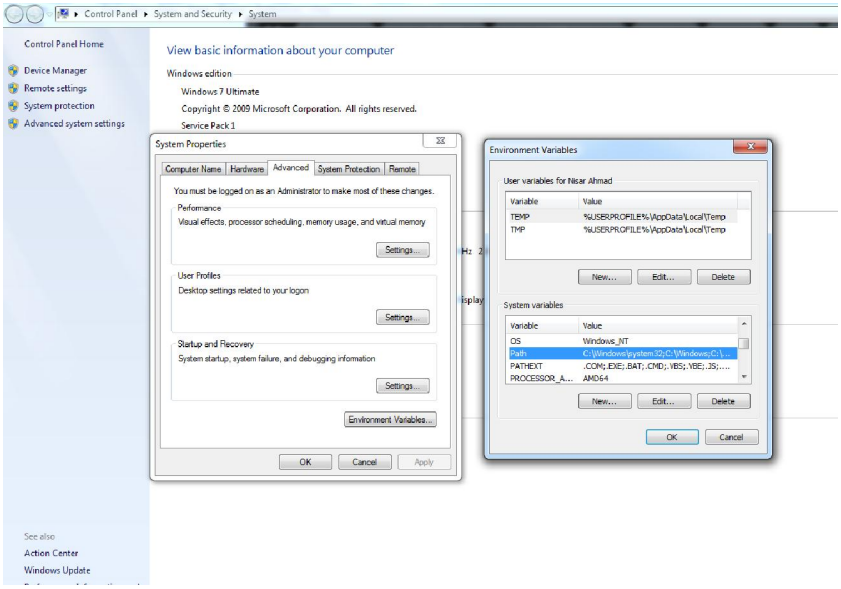


Figure 3-3 Setting up Environment Variables

3.2.2 STM32 Toolchain for ARM GCC

The next step involves to download the ARM Toolchain to support GCC which can be downloaded from the internet and add path in environment variables as described in the previous section. In this case it was the following path:

C:\STM32_Toolchain\gcc-arm-none-eabi-5_3-160412\bin;

3.2.3 Cygwin64 Tools

Although Windows has powerful command shell to use command prompt supporting a lot of features but still Linux has its terminal being used over the years and has more powerful shell. Cygwin provided Linux like environment for the users who want to work on windows but need to use the commands of Linux. This tool intermix both worlds so Linux developers may not feel strange while working on windows command prompt. We can use almost all commands that Linux shell supports in windows.

Cygwin64 was downloaded and installed for windows 7 (64-bit) and added the path in environment variables like:

C:\cygwin64\bin;

3.2.4 GITLAB support

Zerynth has its own Git repository to manage the source code and more often it is required to clone the projects or push the changes made during on-going development activities. Linux provides Git support in terminal once it's installed.

Fortunately, Git has also commands to complete operation for windows. You just need to download Git bash and install in windows. Then adding the path of bin file in system environment variables one can make use of Git features in windows.

The path was like:

C:\Program Files\Git\cmd;

3.2.4 Debug settings

Hexiwear has OpenSDA USB Debug and Programming adapter (available via the Docking Station)

Several industry standard Debug interfaces (PEmicro, CMSIS-DAP, JLink)

Drag-n-drop MSD Flash-programming

Virtual USB to Serial Port

OpenSDA used openOCD on its backend operation. Hexiwear developer team has provided openOCD along its release. So all debugging and programming of Flash memory is done using DAPLink interface and openOCD.

NXP has provided Kinetis design studio (KSD) based on Eclipse. KSD is used for debugging the Hexiwear. KSD can be downloaded from NXP website. Some eclipse updates are required to be done before start debugging. All information related to eclipse updates, installation of Python software and PyOCD plugin in KSD is available at:

https://docs.mikroe.com/Hexiwear_writing_first_program

Once all eclipse updates and PyOCD plugin is installed. Do the following

- Open KSD and go to debug configurations
- Right Click on GDB PyOCD Debugging and create new instance
- In Main tab, provide the path of .elf file in C/C++ Application field like

D:\Zerynth_Home\Viper-VM\build\viper.elf

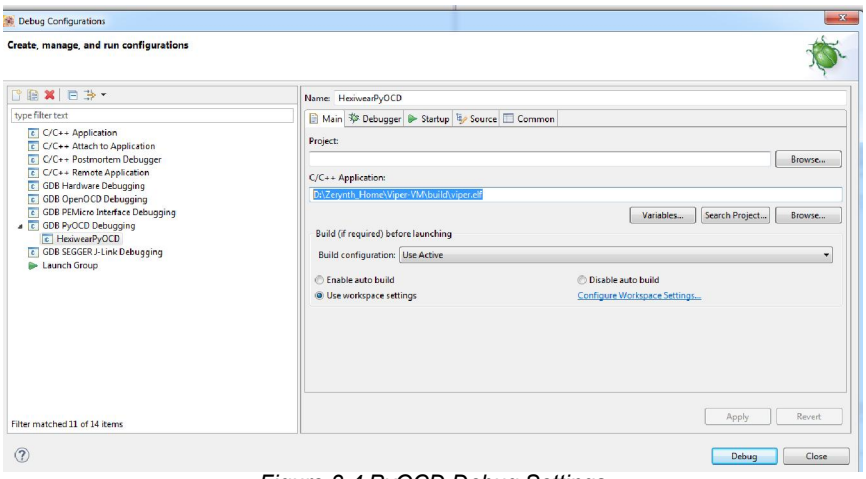


Figure 3-4 PyOCD Debug Settings

- Next click on Debugger Tab
- Check the box saying " *Start PyOCD Locally"*
- Provide a path of pyocd-gdbserver.exe in executable field (given by Hexiwear; from internet)
- GDB Port 3333; Semi hosting port 4444
- In other options put "*-t k64f*"
- For GDB Client setup provide the path of arm-none-eabi-gdb.exe in executable field. This executable is available in ARM Toolchain as shown in Fgure3-5.

3.2.5 OpenOCD Flash Programing

OpenOCD Flash programing is very useful if Drag-and-drop feature of DAPLink interface goes wrong during development activity. Telnet client features of windows make it possible to have to run openOCD for flash programming. Telnet should have to be enabled in windows features from

Control panel->Programs-> Turn on and off windows features ->check Telnet Client

After enabling the Telnet Client now follow the procedure to start programming

- Open command prompt
- Change the path to opnocd.exe executable (in this case it was the following)

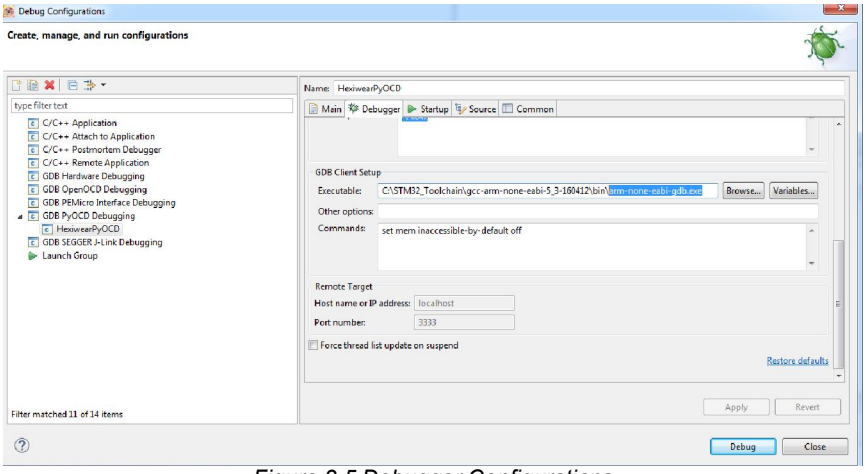


Figure 3-5 Debugger Configurations

C:\Freescale\KDS_v3\openocd\bin

- Type command: openocd.exe –f kinteis.cfg
- Now open another command prompt and type: *telnet localhost 4444*
- At this point you can give flash commands to program, read sectors, checking the current flash specifications.
- Type command : program “Path of bin File to be programmed”
- Other flash commands could be :
  - flash info 0 (For Flash bank 0 information)
  - halt (halt the CPU)
  - reset run
  - mdw, mdh,mdb
  - flash read_bank 0

3.3 Configuration of Minimum Viable Zerynth Virtual Machine

Zerynth virtual machine can support Hexiwear and FreeRTOS. For this, VM needs to be configured minimally to so that all its essential features can be implemented and tested. The minimal configurations needed are as follows:

2. To support Hexiwear as new platform
   - Clock configuration for MK64 CPU
   - Linker script setting for memory allocation
   - MakeFile and board specific settings
   - Serial, Flash and GPIO drivers are essential

The following section contains the description of how to configure VM for minimum viable operation.

3.4 BootUp Configuration for MK64FN1M0VDC12

First step to start with existing RTOS i.e. Chibi2, was to change some sections code that are related to boot up which includes clock configuration and linker script configuration. The startup code written in *crt0.c.* Whenever an MCU is powered up it starts the code from Reset Hander whose definition is written in *crt0.c and* this functions configures the clock and starts the *main ().*

3.4.1 Clock Configuration

Reset hander configures the clock in *__early_init ()* by calling a function *K64_clock_init ().*

K64 clock setup is bit complex to understand the Figure3-6 shows clock distribution and how to configure the clock.

The MCG module controls which clock source is used to derive the system clocks. The clock generation logic divides the selected clock source into a variety of clock domains, including the clocks for the system bus masters, system bus slaves, and flash memory. The clock generation logic also implements module-specific clock gating to allow granular shutoff of modules.

The primary clocks for the system are generated from the MCGOUTCLK clock. The clock generation circuitry provides several clock dividers that allow different portions of the device to be clocked at different frequencies. This allows for trade-offs between Performance and power dissipation.

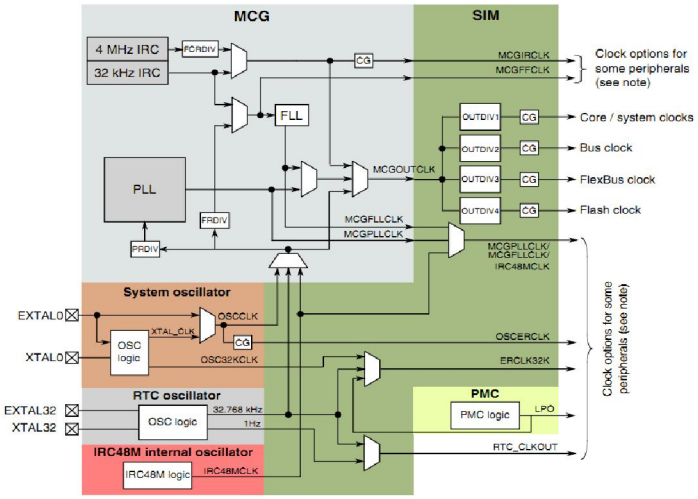


Figure 3-6 K64 Clock Distribution

Zerynth VM needs two clocks to be configured properly for its minimal working

1. System or Core Clock
2. Bus Clock

These two clocks are configured in function K64_clock_init ().

3.4.2 Linker Script Configuration

Linker script file named as *linker_script.ld* is also board specific file to configure all memories available on-chip. The memory map of each MCU provides the details of each memory region and related information. Linker script was configured for K64 MCU by following code

```
MEMORY
{
  m_interrupts        : org = 0x00000000, len = 0x00000400
  m_flash_config      : org = 0x00000400, len = 0x00000010
  m_interrupts_ram    : org = 0x1FFF0000, len = 0x00000410
  flash               : org = 0x00000410, len = 0x000FFBF0
  ram1                : org = 0x1FFF0410, len = 0x0000FC00
  ram2                : org = 0x20000000, len = 0x00030000
}
```

Memory is divided into ram and flash. The sections are written to specify where to place each block of code e.g. text, data, bss, and heap, interrupt vectors etc.

3.4.3 Vector Table Configuration

ARM- Cortex series has its standard 16 core interrupts as shown in Table3-1 below.

The interrupt source assignments are defined in the following table.

- Vector number — the value stored on the stack when an interrupt is serviced.
- IRQ number — non-core interrupt source count, which is the vector number minus 16.

The IRQ number is used within ARM's NVIC documentation.

Address	Vector	IRQ[1]	NVIC non-IPR register number [2]	NVIC IPR register number [3]	Source module	Source description
ARM Core System Handler Vectors						
0x0000_0000	0	–	–	–	ARM core	Initial Stack Pointer
0x0000_0004	1	–	–	–	ARM core	Initial Program Counter
0x0000_0008	2	–	–	–	ARM core	Non-maskable Interrupt (NMI)
0x0000_000C	3	–	–	–	ARM core	Hard Fault
0x0000_0010	4	–	–	–	ARM core	MemManage Fault
0x0000_0014	5	–	–	–	ARM core	Bus Fault
0x0000_0018	6	–	–	–	ARM core	Usage Fault
0x0000_001C	7	–	–	–	—	—
0x0000_0020	8	–	–	–	—	—
0x0000_0024	9	–	–	–	—	—
0x0000_0028	10	–	–	–	—	—
0x0000_002C	11	–	–	–	ARM core	Supervisor call (SVCall)
0x0000_0030	12	–	–	–	ARM core	Debug Monitor
0x0000_0034	13	–	–	–	—	—
0x0000_0038	14	–	–	–	ARM core	Pendable request for system service (PendableSrvReq)
0x0000_003C	15	–	–	–	ARM core	System tick timer (SysTick)

Table 3-1 ARM-Cortex Core Interrupts

K64 MCU has 86 non-core interrupts. All of the core or non-core interrupts need to be configured properly before startup. VM has generic board specific file named *vectors.c.* This file contains all information of interrupts and their handlers. Core interrupts are configured here while non-core interrupts can be configured run time. At startup all vector table is copied from flash to ram and this action allows a programmer to install handler of specific IRQ at run time. The second interrupt vector shown above “Initial Program Counter” is actually a vector for Reset Handler. The section “vectors” in linker script correspond to vector table in flash memory and .ram_vectors corresponds to ram address space for vector table.

3.4.4 MakeFile and Board Specific Configurations

In porting process the first step was to identify the sections of Virtual Machine where changes to be made. VM was saved in D: drive and named it Viper-VM. Since it was going to integrate a new platform with Zerynth Virtual machine so some new folders were created on specific path locations as described below step by step:

1. created a board named folder Hexiwear at the following location

D:\Zerynth_Home\Viper-VM\boards\hexiwear

This folder contains board specific changes and important files needed to be changed were platform.mk, port.c and port.def. The complete details will be covered in the subsequent section.

2. Another folder named K64 was created in vhal layer to make VHAL specific changes. The path of folder needs to be at following place

D:\Zerynth_Home\Viper-VM\common\vhal\ARMCMx\K64

3. A folder named HEXIWEAR (in caps) created at the following address which contains board.mk and board.c files where configuration of clock for Hexiwear was done:

D:\Zerynth_Home\Viper-VM\common\vos\chibi2\os\boards\HEXIWEAR

4. Another folder named *K64* created at the following address:

D:\Zerynth_Home\Viper-VM\common\vos\chibi2\os\os\hal\platforms\K64

All the header files and c files for peripheral drivers that were already available from manufacturer. Like MK64F12.h and other free scale pre-compiled files.

5. Another folder named *K64* created at the following location which contains port.mk to specify which files need to be included for current porting. Like vectors.c is included here for Hexiwear and all GCC and CMSIS paths are included :

D:\Zerynth_Home\Viper-VM\common\vos\chibi2\os\os\ports\GCC\ARMCMx\K64

After creating the necessary folders and copying files some changes have to be made to support Hexiwear hardware platform.

1. ***Changes in platform.mk for Hexiwear***

   The board name folder should be replaced everywhere in makefiles with K64. Following changes are made in platform.mk file

```
MCU = cortex-m4
#####
VHAL_PLATFORM=ARMCMx/K64
BOARD_CONFIG_DIR=boards/$(BOARD)/config
RTOS_AVAILABLE:= NO
BOARD_OS_BOARD:=HEXIWEAR
```

   Or some other changes if a new RTOS is available.

2. ***Changes in Port.c and Port.def***

   Port.c and Port.def needs to be changed a lot because the first file contains the details of all types and number of peripherals supported by current board. It has generic implementation to assign virtual names for each instance of a peripheral. For example Hexiwear has three UARTS that a user can access using docking station. VM has given them virtual names as ser 0, 1 and 3. The following code in port.c will map the first physical serial port to ser 0 and so on.

```
/* PERIPHERAL MAPS */


BEGIN_PERIPHERAL_MAP(serial) \
PERIPHERAL_ID(1), \
PERIPHERAL_ID(3), \
PERIPHERAL_ID(4), \
END_PERIPHERAL_MAP(serial);
```

   This file also contains c functions for byte code storage on flash. Byte code is application code be to run on virtual machine. Up linker has a responsibility to program specific section of flash memory with byte code. That section is addressed by the codemem pointer in linker script.

   Port.def, on the other hand, contains macros to define virtual pins from physical pins. A pin can be used for an alternate functions and this file describes and make virtual pin from physical attributes.

3.5 Implementation of VHAL Drivers

After properly configuring the linker script, clock, interrupt vectors, make files, Systick and paths the board will be able to boot onto main() and initialize chibi2 OS. At this point we can create threads and semaphores to test the existing board with chibi2 RTOS. However in order to be sure whether Chibi2 thread is working on K64 MCU properly or not, we have to make some visual checks i.e. blinking LED will suffice the need for operation. First it is started to write VHAL drivers for GPIO first so that it may become possible to verify the Chibi OS thread is working to toggle the LED.

3.5.1 VHAL GPIO Drivers for K64 MCU

The general-purpose input and output (GPIO) module communicates to the processor core via a zero wait state interface for maximum pin performance. The GPIO registers support 8-bit, 16-bit or 32-bit accesses.

The GPIO data direction and output data registers control the direction and output data of each pin when the pin is configured for the GPIO function. The GPIO input data register displays the logic value on each pin when the pin is configured for any digital function, provided the corresponding Port Control and Interrupt module for that pin is enabled. Efficient bit manipulation of the general-purpose outputs is supported through the addition of set, clear, and toggle write-only registers for each port output data register.

- FEATURES

  Features of the GPIO module include:

  Port Data Input register visible in all digital pin-multiplexing modes

Port Data Input register with corresponding set/clear/toggle registers Port Data Direction register Zero wait state access to GPIO registers through IOPORT

- Signal Multiplexing

To optimize functionality in small packages, pins have several functions available via signal multiplexing. The Port Control block controls which signal is present on the external pin.

- Clock Gating

The clock to the port control module can be gated on and off using the SCGC5 [PORTx] bits in the SIM module. These bits are cleared after any reset, which disables the clock to the corresponding module to conserve power. Prior to initializing the corresponding module, set SCGC5 [PORTx] in the SIM module to enable the clock. Before turning off the clock, make sure to disable the module.

- VHAL Layer GPIO functions

Zerynth hardware abstraction layer provides general functions to access the GPIO features of any underlying MCU. The brief details of some of these functions are as follows:

- vhalPinSetMode

Set the digital mode of *vpin* to *mode*. Valid values for *mode* are the digital input and output PINMODE macros. Return 0 in case of success.

- vhalPinToggle

Invert the digital value of *vpin*. If *vpin* is high it is set to low, if *vpin* is low it is set to high. Return 0 in case of success.

- vhalPinSetToPeripheral

Transfer the control of *vpin* to a peripheral identified by *prph*. The configuration parameters for *vpin* are passed via *prms* in a format depending on the microcontroller porting. Return 0 in case of success. The parameter *prph* is ignored in the current version of the VHAL.

- vhalPinFastSet

Bypass the virtual pin indirection by operating on the microcontroller register *port* with offset *pad*. Set the corresponding pin to high.

These functions have direct access to lower Microcontroller abstraction layer which implements the board specific GPIO drivers for K64 MCU.

3.5.2 VHAL Flash Drivers

Hexiwear has two (512 KB each) blocks of program flash consisting of 4 KB sectors. The FTFE module includes a memory controller that executes commands to modify flash memory contents. An erased bit reads '1' and a programmed bit reads '0'. The programming operation is unidirectional; it can only move bits from the '1' state (erased) to the '0' state (programmed). Only the erase operation restores bits from '0' to '1'; bits cannot be programmed from a '0' to a '1'.

Vhal flash drivers are needed to program and retrieve the byte code from flash.

VHAL layer provides the following functions to access the on-chip flash:

- vhalFlashErase

Erase sector starting at *addr* for *size* bytes. If *size* is greater than the sector length, following sectors are erased. It will erase one or more sectors. The address should be 128-bit aligned i.e. FlashAdrees [3-0] = 0000. After clearing CCIF to launch the Erase Flash Sector command, the FTFE erases the selected program flash or data flash sector and then verifies that it is erased. The Erase Flash Sector command aborts if the selected sector is protected. If the swap system is enabled, the swap indicator address in each program flash block is implicitly protected from sector erase unless the swap system is in the UPDATE or UPDATE-ERASED state and the program flash sector containing the swap indicator address being erased is in the non-active block. If the erase-verify fails the FSTAT [MGSTAT0] bit is set.

- vhalFlashWrite

Write *data* starting at *addr* for *len* bytes. In much architecture, for vhalFlashWrite to work, the sectors must be erased first. It will program one phrase (8bytes) at a time. The address

should be 64-bit aligned i.e. FlashAdress [2-0] = 000; Provide an address in second half of Flash i.e. from 0x80000-1FFFFFF (512KB)

- vhalFlashAlignToSector

If *addr* points to the start of a sector, return *addr*. Otherwise the start of the next sector is returned. Return NULL on error.

- vhalInitFLASH

To initialize the flash on start up with known configuration.

3.5.3 VHAL Serial Drivers

Hexiwear has 5 UART modules with the following features:

1. Standard features of all UARTs:

- RS-485 support
- Hardware flow control (RTS/CTS)
- 9-bit UART to support address mark with parity
- MSB/LSB configuration on data

2. UART0 and UART1 are clocked from the core clock, the remaining UARTs are clocked on the bus clock. The maximum baud rate is 1/16 of related source clock frequency.

3. IrDA is available on all UARTs

4. UART0 contains the standard features plus ISO7816

5. UART0 and UART1 contain 8-entry transmit and 8-entry receive FIFOs

6. All other UARTs contain a 1-entry transmit and receive FIFOs

Zerynth VM needs UART interface to upload the byte code from computer serial port to VM running on target platform (K64 MCU). The UART can also be used to interface some other sensors communicating via UART serial protocol with Hexiwear docking station.

Vhal layer provides the following functions to access the UART serial port on target platform:

- vhalSerialInit

  Initialize the serial peripheral identified by the peripheral index *ser*. Baud rate is set to *baud* and configuration parameters are taken from *cfg* encoded with: macro: `SERIAL_CFG`. *rxpin* and *txpin* are configured accordingly. Return 0 on success

- vhalSerialRead

  Read *len* bytes from *ser* into *buf* blocking the current thread until all bytes are read. Return the actual number of bytes read.

- vhalSerialWrite

  Write *len* bytes from *buf* to *ser*. Depending on the implementation, the function may return before all bytes are actually written to *ser*. Return the number of bytes written to *ser* or to an internal buffer. This function has several implementations depending upon the requirement. In this case it is used to access the VOSAL layer for synchronization of threads. Multiple threads may want send data over the same serial port then access to that serial port is given to only one thread. This can be done by exploiting the RTOS features of semaphore. A thread will block on binary semaphore until signal by ISR that all data has been transmitted. After that some other thread can the control of serial port for write access.

```
int vhalSerialWrite(uint32_t ser, uint8_t *buf, uint32_t len) {
int serid = GET_PERIPHERAL_ID(serial, ser);
UART_Type * uartbase = g_uartBase[serid];
uart_state_t * uartState = (uart_state_t *)g_uartStatePtr[serid];
uart_status_t retVal = kStatus_UART_Success;
uint32_t result;
/* Indicates current transaction is blocking.*/
uartState->isTxBlocking = true;

 /* Start the transmission process */
 retVal = UART_DRV_StartSendData (serid, buf, len);
```

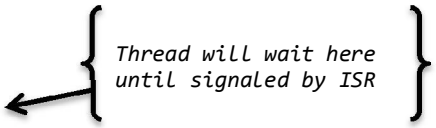
Decoding the serial ID to physical UART instance base address and accessing the UART state structure

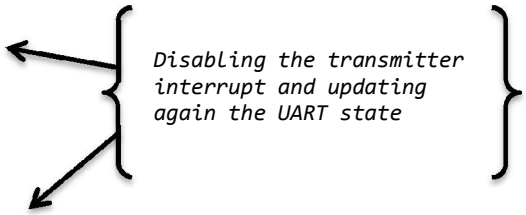
Updating the uart state with available data to be transmitted and enabling interrupt

```
if (retVal == kStatus_UART_Success)
{
    result = vosSemWait (uartState->txIrqSync);
    if (result == VRES_OK)
    {
     UART_BWR_C2_TIE (uartbase, 0U);
    }
    else
    {
     UART_BWR_C2_TIE (uartbase, 0U);
    }

    /* Disable the transmitter data register empty interrupt */
    UART_BWR_C2_TIE(uartbase, 0U);

    /* Update the information of the module driver state */
    uartState->isTxBusy = false;
```





There are some functions available to initialize the UART instance with given details the sequence of these functions should be like this:

1. CLOCK_SYS_EnableUartClock
2. vhalPinSetToPeripheral
3. UART_HAL_Init
4. UART_HAL_SetBaudRate
5. UART_HAL_SetBitCountPerChar
6. UART_HAL_SetParityMode
7. UART_HAL_SetStopBitCount
8. UART_HAL_GetTxFifoSize
9. UART_HAL_SetRxFifoWatermark
10. UART_HAL_SetTxFifoCmd
11. UART_HAL_SetRxFifoCmd
12. UART_HAL_FlushTxFifo
13. UART_HAL_FlushRxFifo
14. vosInstallHandler
15. vhalIrqEnable

/* finally, enable the UART transmitter and receiver*/

16. UART_HAL_EnableTransmitter(uartbase);
17. UART_HAL_EnableReceiver(uartbase);

These functions will completely configure each UART instance after that we can call VHAL read write functions to access the serial port. In Windows the data can be monitored and logged using *Tera Term* software.

At this point Zerynth VM is fully configured to support the Hexiwear for its minimum viable configuration using chibi2 RTOS. The Next chapter will be about replacing the Chibi2 OS with FreeRTOS v9.

Chapter 4

4. Porting FreeRTOS on Zerynth Virtual Machine

4.1 FreeRTOS Basics

FreeRTOS™ is a market leading RTOS from Real Time Engineers Ltd. which supports around 35 architectures. It is professionally developed, strictly quality controlled, robust, supported, and free to embed in commercial products without any requirement to expose your proprietary source code. In this chapter, on its first part, contains introduction to the FreeRTOS basics including directory structure to implementation guide. In second half of this chapter contains description of the porting of FreeRTOS to MK64FN1M0VDC12 MCU.

4.1.1 Why choose FreeRTOS?

There are certain unique features and support available for porting FreeRTOS to microcontroller for commercial applications which makes it better choice for among other RTOS available. Following are some features that FreeRTOS provides:

- FreeRTOS is not only strict in high quality in its C code but also in its implementation.
- FreeRTOS never performs a non-deterministic operation, such as walking a linked list, from inside a critical section or interrupt.
- FreeRTOS has efficient software timer implementation that does not use any CPU time unless a timer actually needs servicing. It means software timers do not contain variables that need to be counted down to zero.
- Likewise, lists of Blocked (pended) tasks do not require time consuming periodic servicing.
- Direct to task notifications allow fast task signaling, with practically no RAM overhead, and can be used in the majority of inter-task and interrupt to task signaling scenarios.
- The FreeRTOS queue usage model manages to combine simplicity with flexibility (in a tiny code size) - attributes that are normally mutually exclusive.
- FreeRTOS queues are base primitives on top of which other communication and synchronization primitives are built. The code re-use obtained dramatically reduced overall code size, which in turn assists testing and helps ensure robustness.
- Pre-configured example projects for all supported ports Free support, quoted as better than some commercial alternatives

The core FreeRTOS source files (those that are common to all ports) conform to the MISRA coding standard guidelines.

4.1.2 FreeRTOS Directory Structure and Data Types

The FreeRTOS download includes source code for every processor port, and every demo application. Placing all the ports in a single download greatly simplifies distribution, but the number of files may seem daunting. The directory structure is however very simple, and the FreeRTOS real time kernel is contained in just 3 files (additional files are required if software timer, event group or co-routine functionality is required).

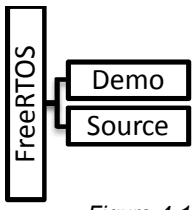


Figure 4-1 FreeRTOS Directory Structure

The core RTOS code is contained in three files, which are called tasks.c, queue.c and list.c. These three files are in the FreeRTOS/Source directory. The same directory contains two optional files called timers.c and croutine.c which implement software timer and co-routine functionality respectively.

Each supported processor architecture requires a small amount of architecture specific RTOS code. This is the RTOS portable layer, and it is located in the FreeRTOS/Source/Portable/ [compiler]/ [architecture] sub-directories, where [compiler] and [architecture] are the compiler used to create the port, and the architecture on which the port runs, respectively.

The structure of the `FreeRTOS/Source` directory is shown below.

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

Figure 4-2 FreeRTOS Essential Files

Source files specific to a FreeRTOS port are contained within the FreeRTOS/Source/portable directory. The portable directory is arranged as a hierarchy, first by compiler, then by Processor architecture. This is shown in Figure 4-3.

```
FreeRTOS
  └─Source
    └─portable Directory containing all port specific source files
      ├─MemMang Directory containing the 5 alternative heap allocation source files
      ├─[compiler 1] Directory containing port files specific to compiler 1
      │  ├─[architecture 1] Contains files for the compiler 1 architecture 1 port
      │  ├─[architecture 2] Contains files for the compiler 1 architecture 2 port
      │  └─[architecture 3] Contains files for the compiler 1 architecture 3 port
      └─[compiler 2] Directory containing port files specific to compiler 2
         ├─[architecture 1] Contains files for the compiler 2 architecture 1 port
         ├─[architecture 2] Contains files for the compiler 2 architecture 2 port
         └─[etc.]
```

Figure 4-3 Port Specific FreeRTOS Architecture

Examples of portable layer directories:

- If using the TriCore 1782 port with the GCC compiler:
- The TriCore port specific file *Port.c is in the FreeRTOS/Source/Portable/GCC/TriCore* directory. So all other directories can be deleted or ignored.
- Similarly if the compiler is IAR then port.c can be located under the following path for TriCore 1782 *FreeRTOS/Source/Portable/IAR/TriCore*

There are four Data types that are defined for each port. These are:

- TickType_t

If configUSE_16_BIT_TICKS is set to non-zero (true), then TickType_t is defined to be an unsigned 16-bit type. If configUSE_16_BIT_TICKS is set to zero (false), then TickType_t is defined to be an unsigned 32-bit type. In 32-bit architectures should always set configUSE_16_BIT_TICKS to 0.

- BaseType_t

This is defined to be the most efficient, natural, type for the architecture. For example, on a 32-bit architecture BaseType_t will be defined to be a 32-bit type. On a 16-bit architecture BaseType_t will be defined to be a 16-bit type. If BaseType_t is defined to char then particular care must be taken to ensure signed chars are used for function return values that can be negative to indicate an error.

- UBaseType_t

This is an unsigned BaseType_t.

- StackType_t

Defined to the type used by the architecture for items stored on the stack. Normally this would be a 16-bit type on 16-bit architectures and a 32-bit type on 32-bit architectures, although there are some exceptions. Used internally be FreeRTOS.

4.2 Tasks and Co-Routines

An application can be designed using just tasks, just co-routines, or a mixture of both - however tasks and co-routines use different API functions and therefore a queue (or semaphore) cannot be used to pass data from a task to a co-routine or vice versa.

4.2.1 Tasks

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler, may therefore, repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real time RTOS scheduler to ensure that the processor context (register values, stack contents, etc.) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in.

4.2.2 Task States

A task can exist in one of the following states:

1. Running

When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

2. Ready

Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

3. Blocked

A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay () it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.

Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

4. Suspended

Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend () and xTaskResume () API calls respectively.

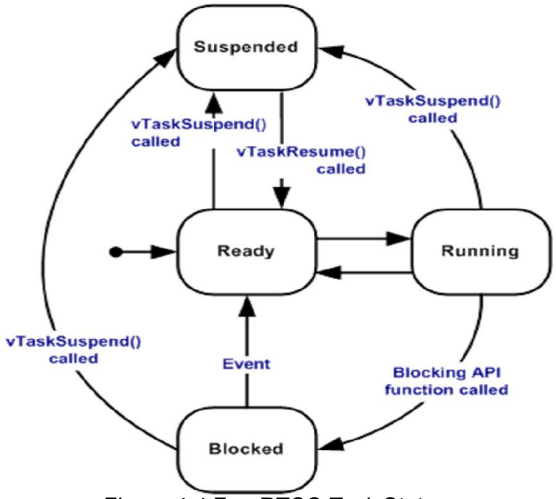


Figure 4-4 FreeRTOS Task States

4.2.3 Tasks Priorities

Each task is assigned a priority from 0 to (configMAX_PRIORITIES - 1), where configMAX_PRIORITIES is defined within FreeRTOSConfig.h.

Low priority numbers denote low priority tasks. The idle task has priority zero (tskIDLE_PRIORITY).The FreeRTOS scheduler ensures that tasks in the Ready or Running state will always be given processor (CPU) time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed into the Running state is always the highest priority task that is able to run.

Any number of tasks can share the same priority. If configUSE_TIME_SLICING is not defined, or if configUSE_TIME_SLICING is set to 1, then Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme

4.2.4 Co-Routines

Co-routines were implemented for use on very small devices, but are very rarely used in the field these days. For that reason, while there are no plans to remove co-routines from the code, there are also no plans to develop them further.

Co-routines are conceptually similar to tasks but have the following fundamental differences

1. Stack usage

   All the co-routines within an application share a single stack. This greatly reduces the amount of RAM required compared to a similar application written using tasks.

2. Scheduling and priorities

   Co-routines use prioritized cooperative scheduling with respect to other co-routines, but can be included in an application that uses preemptive tasks.

3. Macro implementation

   The co-routine implementation is provided through a set of macros.

4. Restrictions on use

   The reduction in RAM usage comes at the cost of some stringent restrictions in how co-routines can be structured.

4.3 Queues, Mutexes and Semaphores

4.3.1 Queues

Queues are the primary form of inter task communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

Queues are normally used as First in First out (FIFO) buffers, where data is written to the end (Tail) of the queue and removed from the front (head) of the queue. Figure4-5 demonstrates data being written to and read from a queue that is being used as a FIFO. It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue.

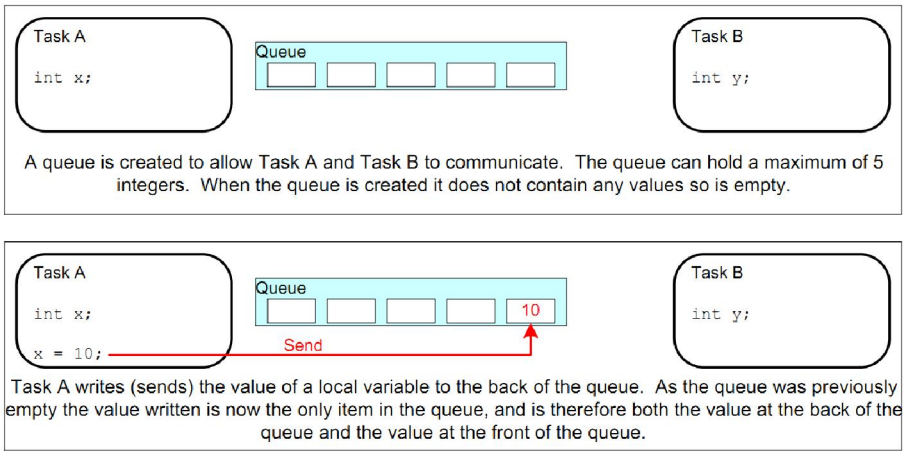


Figure 4-5 FreeRTOS Queue

There are two ways in which queue behavior could have been implemented:

a. Queue by copy

Queuing by copy means the data sent to the queue is copied byte for byte into the queue.

b. Queue by reference

Queuing by reference means the queue only holds pointers to the data sent to the queue, not the data itself.

FreeRTOS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queuing by reference because:

- Stack variable can be sent directly to a queue, even though the variable will not exist after the function in which it is declared has exited.
- Data can be sent to a queue without first allocating a buffer to hold the data, and then copying the data into the allocated buffer.
- The sending task can immediately re-use the variable or buffer that was sent to the queue.
- The sending task and the receiving task are completely de-coupled—the application designer does not need to concern themselves with which task 'owns' the data, or which task is responsible for releasing the data.
- Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue instead.
- The RTOS takes complete responsibility for allocating the memory used to store data.
- In a memory protected system, the RAM that a task can access will be restricted. In that case queuing by reference could only be used if the sending and receiving task could both access the RAM in which the data was stored. Queuing by copy does not

impose that restriction; the kernel always runs with full privileges, allowing a queue to be used to pass data across memory protection boundaries.

Access by Multiple Tasks

Queues are objects in their own right that can be accessed by any task or ISR that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

Blocking on Queue Reads

When a task attempts to read from a queue, it can optionally specify a 'block' time. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty. A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue. The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data. If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked.

Blocking on Queue Writes

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task that is waiting for space. If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked.

Blocking on Multiple Queues

Queues can be grouped into sets, allowing a task to enter the Blocked state to wait for data to become available on any of the queues in the set.

There are several queue API functions available in FreeRTOS to create a queue and perform operations on queue.

Receiving Data from Multiple Sources

It is common in FreeRTOS designs for a task to receive data from more than one source. The receiving task needs to know where the data came from to determine how the data should be processed. An easy design solution is to use a single queue to transfer structures with both the value of the data and the source of the data contained in the structure's fields. This scheme is demonstrated in Figure 4-6.

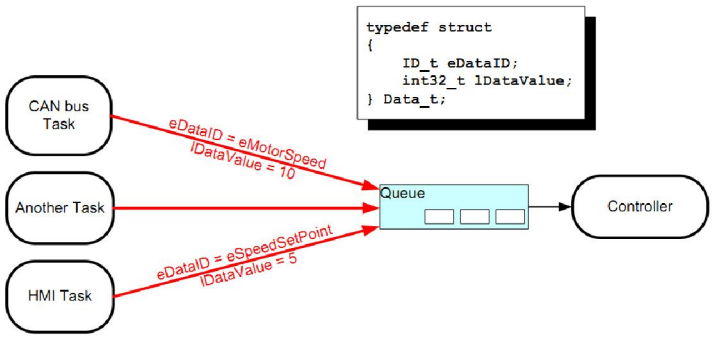


Figure 4-6 Receiving Data from Multiple Sources

4.3.2 Mutexes

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique. The goal is to ensure that, once a task starts to access a shared resource that is not

re-entrant and not thread-safe, the same task has exclusive access to the resource until the resource has been returned to a consistent state.

FreeRTOS provides several features that can be used to implement mutual exclusion, but the best mutual exclusion method is to (whenever possible, as it is often not practical) design the application in such a way that resources are not shared, and each resource is accessed only from a single task.

Basic critical sections are regions of code that are surrounded by calls to the macros taskENTER_CRITICAL () and taskEXIT_CRITICAL (), respectively. Critical sections are also known as critical region taskENTER_CRITICAL () and taskEXIT_CRITICAL () do not take any parameters, or return a value.

A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from 'MUTual EXclusion'. ConfigUSE_MUTEXES must be set to 1 in FreeRTOSConfig.h for Mutexes to be available. When used in a mutual exclusion scenario, the Mutex can be thought of as a token that is associated with the resource being shared. For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back. Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

Even though Mutexes and binary semaphores share many characteristics, the primary difference is what happens to the semaphore after it has been obtained:

- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.

Priority inversion in Figure 4-7 shows one of the potential pitfalls of using a Mutex to provide mutual exclusion. The sequence of execution depicted shows the higher priority Task 2 having to wait for the lower priority Task 1 to give up control of the Mutex. A higher priority task being delayed by a lower priority task in this manner is called 'priority inversion'. This undesirable behavior would be exaggerated further if a medium priority task started to execute while the high priority task was waiting for the semaphore—the result would be a high priority task waiting for a low priority task—without the low priority task even being able to execute.

FreeRTOS mutexes and binary semaphores are very similar—the difference being that mutexes include a basic 'priority inheritance' mechanism, whereas binary semaphores do not. Priority inheritance is a scheme that minimizes the negative effects of priority inversion. It does not 'fix' priority inversion, but merely lessens its impact by ensuring that the inversion is always time bounded. However, priority inheritance complicates system timing analysis, and it is not good practice to rely on it for correct system operation.

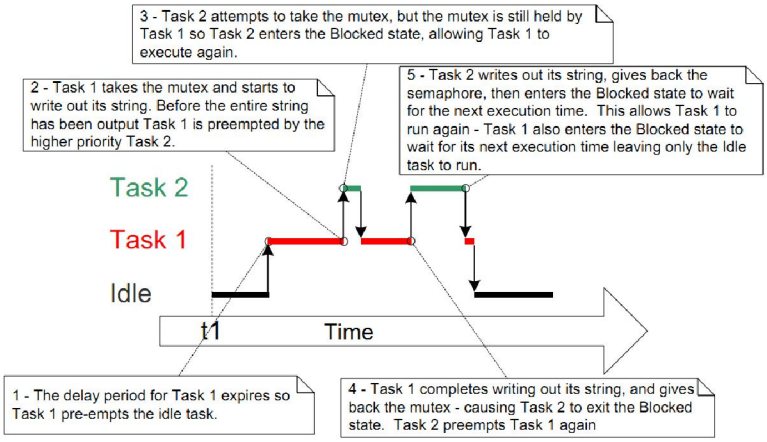


Figure 4-7 Priority Inversion

4.3.3 Semaphores

Binary Semaphore

An interrupt service routine must record the cause of the interrupt, and clear the interrupt. Any other processing necessitated by the interrupt can often be performed in a task, allowing the

interrupt service routine to exit as quickly as is practical. This is called ‘deferred interrupt processing’, because the processing necessitated by the interrupt is ‘deferred’ from the ISR to a task. If the priority of the task to which interrupt processing is deferred is above the priority of any other task, then the processing will be performed immediately, just as if the processing had been performed in the ISR itself.

The interrupt safe version of the Binary Semaphore API can be used to unblock a task each time a particular interrupt occurs, effectively synchronizing the task with the interrupt. This allows the majority of the interrupt event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR. If the interrupt processing is particularly time critical, then the priority of the deferred processing task can be set to ensure the task always pre-empts the other tasks in the system. The ISR can then be implemented to include a call to portYIELD_FROM_ISR (), ensuring the ISR returns directly to the task to which interrupt processing is being deferred. This has the effect of ensuring the entire event processing executes contiguously (without a break) in time, just as if it had all been implemented within the ISR itself. Figure 4-8 describes how the execution of the deferred processing task can be controlled using a semaphore.

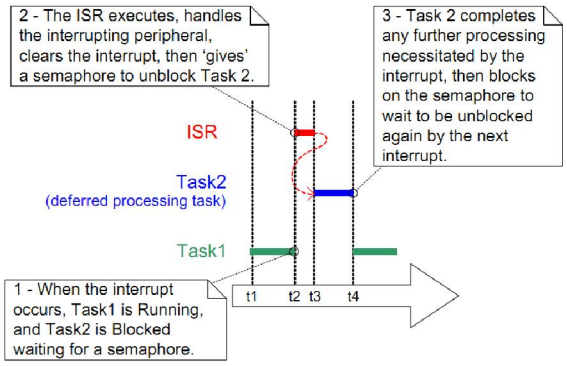


Figure 4-8 Deferred Interrupt Processing Using Semaphores

The deferred processing task uses a blocking ‘take’ call to a semaphore as a means of entering the Blocked state to wait for the event to occur. When the event occurs, the ISR uses a ‘give’ operation on the same semaphore to unblock the task so that the required event processing can proceed.

‘Taking a semaphore’ and ‘giving a semaphore’ are concepts that have different meanings depending on their usage scenario. In this interrupt synchronization scenario, the binary semaphore can be considered conceptually as a queue with a length of one. The queue can contain a maximum of one item at any time, so is always either empty or full (hence, binary). By calling xSemaphoreTake (), the task to which interrupt processing is deferred effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty. When the event occurs, the ISR uses the xSemaphoreGiveFromISR () function to place a token (the semaphore) into the queue, making the queue full. This causes the task to exit the Blocked state and remove the token, leaving the queue empty once more. When the task has completed its processing, it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event.

The deferred interrupt handling task had one weakness; it did not use a time out when it called xSemaphoreTake().Instead, the task passed portMAX_DELAY as the xSemaphoreTake() xTicksToWait parameter, which results in the task waiting indefinitely (without a time out) for the semaphore to be available. Indefinite timeouts are often used in example code because their use simplifies the structure of the example, and therefore makes the example easier to understand. However, indefinite timeouts are normally bad practice in real applications, because they make it difficult to recover from an error. As an example, consider the scenario where a task is waiting for an interrupt to give a semaphore, but an error state in the hardware is preventing the interrupt from being generated:

- If the task is waiting without a time out, it will not know about the error state, and will wait forever.

- If the task is waiting with a time out, then xSemaphoreTake() will return pdFAIL when the time out expires, and the task can then detect and clear the error the next time it executes.

Counting Semaphores

Just as binary semaphores can be thought of as queues that have a length of one, counting semaphores can be thought of as queues that have a length of more than one. Tasks are not interested in the data that is stored in the queue—just the number of items in the queue configUSE_COUNTING_SEMAPHORES must be set to 1 in FreeRTOSConfig.h for counting semaphores to be available.

Each time a counting semaphore is 'given', another space in its queue is used. The number of items in the queue is the semaphore's 'count' value.

Counting semaphores are typically used for two things:

1. **Counting events**

   In this scenario, an event handler will 'give' a semaphore each time an event occurs— causing the semaphore's count value to be incremented on each 'give'. A task will 'take' a semaphore each time it processes an event—causing the semaphore's count value to be decremented on each 'take'. The count value is the difference between the number of events that have occurred and the number that have been processed.

   Counting semaphores that are used to count events are created with an initial count value of zero.

2. **Resource management**

   In this scenario, the count value indicates the number of resources available. To obtain control of a resource, a task must first obtain a semaphore—decrementing the semaphore's count value. When the count value reaches zero, there are no free resources. When a task finishes with the resource, it 'gives' the semaphore back—incrementing the semaphore's count value.

4.4 Software Timers

Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency. The function executed by the software timer is called the software timer's callback function.

Software timers are implemented by, and are under the control of, the FreeRTOS kernel. They do not require hardware support, and are not related to hardware timers or hardware counters. Note that, in line with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is actually executing.

Software timer functionality is optional. To include software timer functionality:

1. Build the FreeRTOS source file FreeRTOS/Source/timers.c as part of your project.
2. Set configUSE_TIMERS to 1 in FreeRTOSConfig.h.

4.4.1 Software Timer Callback Functions

Software timer callback functions are implemented as C functions. The only thing special about them is their prototype, which must return void, and take a handle to a software timer as its only parameter. The Callback function has prototype as following:

```
void ATimerCallback( TimerHandle_t xTimer );
```

Software timer callback functions execute from start to finish, and exit in the normal way. They should be kept short, and must not enter the Blocked state. Software timer callback functions execute in the context of a task that is created automatically when the FreeRTOS scheduler is started. Therefore, it is essential that software timer callback functions never call FreeRTOS API functions that will result in the calling task entering the Blocked state. It is ok to call functions such as xQueueReceive (), but only if the function's xTicksToWait parameter which specifies the function's block time) is set to 0. It is not ok to call functions such as vTaskDelay (), as calling vTaskDelay () will always place the calling task into the Blocked state.

4.4.2 Timer Period

A software timer's 'period' is the time between the software timer being started, and the software timer's callback function executing.

There are two types of Software Timers:

1. ***One-shot timers***

   Once started, a one-shot timer will execute its callback function once only. A one-shot timer can be restarted manually, but will not restart itself.

2. ***Auto-reload timers***

   Once started, an auto-reload timer will re-start itself each time it expires, resulting in periodic execution of its callback function. Figure 4-9 shows the difference in behavior between a one-shot timer and an auto-reload timer.

   The dashed vertical lines mark the times at which a tick interrupt occurs.

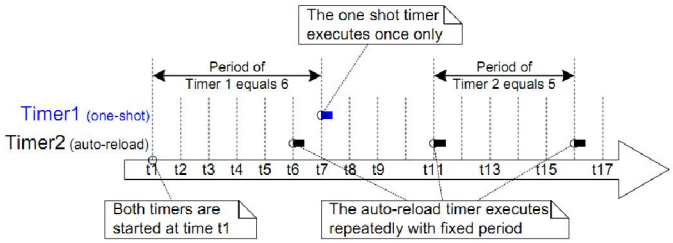


Figure 4-9 One-shot and Auto-reload Timer

4.4.3 Software Timer States

A software timer can be in one of the following two states:

1 ***Dormant***

A Dormant software timer exists, and can be referenced by its handle, but is not running, so its callback functions will not execute.

2 ***Running***

A Running software timer will execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state, or since the software timer was last reset.

Figure 4-10 and Figure 4-11 show the possible transitions between the Dormant and Running states for an auto-reload timer and a one-shot timer respectively. The key difference between the two diagrams is the state entered after the timer has expired; the auto-reload timer executes its callback function then re-enters the Running state, the one-shot timer executes its callback function then enters the dormant state.

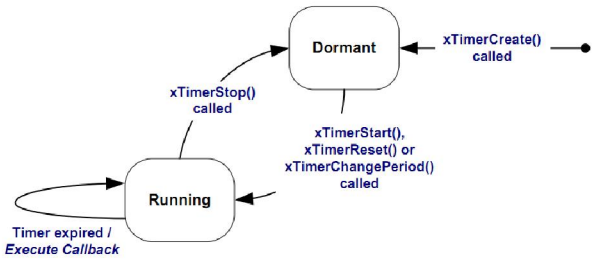


Figure 4-10 Auto-Reload Timer States and Transitions

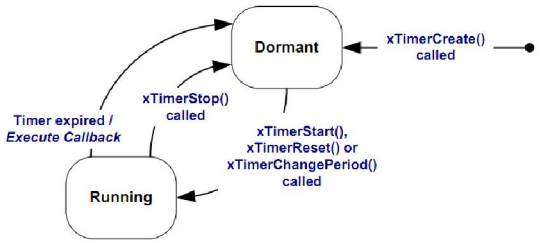


Figure 4-11 One-Shot Timer States and Transitions

4.5 General Porting Steps

For every port of FreeRTOS there are some general steps to be taken. The FreeRTOS kernel source code is generally contained within 3 source files (4 if co-routines are used) that are common to all ports, and one or two 'port' files that tailor the RTOS kernel to a particular architecture.

4.5.1 Porting of FreeRTOS Suggested Steps

Suggested steps are:

1. Download the latest version of the FreeRTOS source code.
2. Unzip the files into a convenient location, taking care to maintain the directory structure.
3. Familiarize yourself with the source code organization and directory structure.
4. Create a directory that will contain the 'port' files for the [architecture] port. Following the convention outlined in the link, the directory should be of the form: FreeRTOS/Source/portable/ [compiler name]/ [processor name]. For example, if you are using the GCC compiler you can create a [architecture] directory off the existing FreeRTOS/Source/portable/GCC directory.
5. Copy empty port.c and portmacro.h files into the directory you have just created. These files should just contain the stubs of the functions and macro's that require implementing. See existing port.c and portmacro.h files for a list of such functions and macros. You can create a stub file from one of these existing files by simply deleting the function and macro bodies.
6. If the stack on the microcontroller being ported to grows downward from high memory to low memory then set portSTACK_GROWTH in portmacro.h to -1, otherwise set portSTACK_GROWTH to 1.
7. Create a directory that will contain the demo application files for the [architecture] port. Following the convention again this should be of the form FreeRTOS/Demo/ [architecture_compiler], or something similar.
8. Copy existing FreeRTOSConfig.h and main.c files into the directory just created. Again these should be edited to be just stub files.
9. Take a look at the FreeRTOSConfig.h file. It contains some macro's that will need setting for your chosen hardware.
10. Create a directory off the directory just created and call it ParTest (probably FreeRTOS/Demo/ [architecture compiler]/ParTest). Copy into this directory a ParTest.c stub file.
11. ParTest.c contains three simple functions to:

1. Setup some GPIO that can flash a few LEDs,
2. Set or clear a specific LED, and
3. Toggle the state of an LED.

4.5.2 FreeRTOS Configuration

FreeRTOS is customized using a configuration file called FreeRTOSConfig.h. Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path. FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.

Each demo application included in the RTOS source code download has its own FreeRTOSConfig.h file. Some of the demos are quite old and do not contain all the available

configuration options. Configuration options that are omitted are set to a default value within an RTOS source file.

Here is a typical FreeRTOSConfig.h definition, followed by an explanation of each parameter:

```
#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0

/* Run time and task stats gathering related definitions. */
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                0
#define configUSE_STATS_FORMATTING_FUNCTIONS    0
```

Figure 4-12 FreeRTOSConfig.h Parameters

Some useful Parameters details are as follows:

configCPU_CLOCK_HZ

Enter the frequency in Hz at which the internal clock that driver the peripheral used to generate the tick interrupt will be executing - this is normally the same clock that drives the internal CPU clock. This value is required in order to correctly configure timer peripherals.

configTICK_RATE_HZ

The frequency of the RTOS tick interrupt. The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the RTOS kernel and is higher than would normally be required.

More than one task can share the same priority. The RTOS scheduler will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

configMAX_PRIORITIES

The number of priorities available to the application tasks. Any number of tasks can share the same priority. Co-routines are prioritized separately. Each available priority consumes RAM within the RTOS kernel so this value should not be set any higher than actually required by your application.

configSUPPORT_STATIC_ALLOCATION

If configSUPPORT_STATIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM provided by the application writer. If configSUPPORT_STATIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM allocated from the FreeRTOS heap. If configSUPPORT_STATIC_ALLOCATION is left undefined it will default to 0.

configSUPPORT_DYNAMIC_ALLOCATION

If configSUPPORT_DYNAMIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM that is automatically allocated from the FreeRTOS heap. If configSUPPORT_DYNAMIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM provided by the application writer. If configSUPPORT_DYNAMIC_ALLOCATION is left undefined it will default to 1.

configAPPLICATION_ALLOCATED_HEAP

By default the FreeRTOS heap is declared by FreeRTOS and placed in memory by the linker. Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to instead be declared by the application writer, which allows the application writer to place the heap wherever they like in memory.

If heap_1.c, heap_2.c or heap_4.c is used, and configAPPLICATION_ALLOCATED_HEAP is set to 1, then the application writer must provide a uint8_t array with the exact name and dimension as shown below. The array will be used as the FreeRTOS heap. How the array is placed at a specific memory location is dependent on the compiler being used - refer to your compiler's documentation.

Uint8_t ucHeap [configTOTAL_HEAP_SIZE];

configKERNEL_INTERRUPT_PRIORITY configMAX_SYSCALL_INTERRUPT_PRIORITY

For ports that only implement configKERNEL_INTERRUPT_PRIORITY

ConfigKERNEL_INTERRUPT_PRIORITY sets the interrupt priority used by the RTOS kernel itself. Interrupts that call API functions must also execute at this priority. Interrupts that do not call API functions can execute at higher priorities and therefore never have their execution delayed by the RTOS kernel activity (within the limits of the hardware itself).

For ports that implement both configKERNEL_INTERRUPT_PRIORITY and ConfigMAX_SYSCALL_INTERRUPT_PRIORITY:

ConfigKERNEL_INTERRUPT_PRIORITY sets the interrupt priority used by the RTOS kernel itself. ConfigMAX_SYSCALL_INTERRUPT_PRIORITY sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.

A full interrupt nesting model is achieved by setting ConfigMAX_SYSCALL_INTERRUPT_PRIORITY above (that is, at a higher priority level) than configKERNEL_INTERRUPT_PRIORITY. This means the FreeRTOS kernel does not completely disable interrupts, even inside critical sections. Further, this is achieved without the disadvantages of a segmented kernel architecture. Note however, certain microcontroller architectures will (in hardware) disable interrupts when a new interrupt is accepted - meaning interrupts are unavoidably disabled for the short period between the hardware accepting the interrupt, and the FreeRTOS code re-enabling interrupts.

Interrupts that do not call API functions can execute at priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY and therefore never be delayed by the RTOS kernel execution.

For example, imagine a hypothetical microcontroller that has 8 interrupt priority levels - 0 being the lowest and 7 being the highest. The Figure 4-13 below describes what can and cannot be done at each priority level should the two configuration constants be set to 4 and 0 as shown:

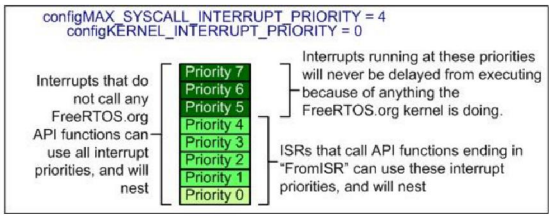


Figure 4-13 FreeRTOS Priority Configuration

These configuration parameters allow very flexible interrupt handling:

Interrupt handling 'tasks' can be written and prioritized as per any other task in the system. These are tasks that are woken by an interrupt. The interrupt service routine (ISR) itself should be written to be as short as it possibly can be - it just grabs the data then wakes the high priority handler task. The ISR then returns directly into the woken handler task - so interrupt processing is contiguous in time just as if it were all done in the ISR itself. The benefit of this is that all interrupts remain enabled while the handler task executes.

Ports that implement configMAX_SYSCALL_INTERRUPT_PRIORITY take this further - permitting a fully nested model where interrupts between the RTOS kernel interrupt priority and configMAX_SYSCALL_INTERRUPT_PRIORITY can nest and make applicable API calls.

Interrupts with priority above configMAX_SYSCALL_INTERRUPT_PRIORITY are never delayed by the RTOS kernel activity.

ISR's running above the maximum syscall priority are never masked out by the RTOS kernel itself, so their responsiveness is not effected by the RTOS kernel functionality. This is ideal for interrupts that require very high temporal accuracy - for example interrupts that perform motor commutation. However, such ISR's cannot use the FreeRTOS API functions.

4.5.3 Memory Management

The RTOS kernel needs RAM each time a task, queue, Mutex, software timer, semaphore or event group is created. The RAM can be automatically dynamically allocated from the RTOS heap within the RTOS API object creation functions, or it can be provided by the application writer.

If RTOS objects are created dynamically then the standard C library malloc () and free () functions can sometimes be used for the purpose, but...

1. they are not always available on embedded systems,
2. they take up valuable code space,
3. they are not thread safe, and
4. they are not deterministic (the amount of time taken to execute the function will differ from call to call)

... So more often than not an alternative memory allocation implementation is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, allowing an application specific implementation appropriate for the real time system being developed to be provided. When the RTOS kernel requires RAM, instead of calling malloc (), it instead calls pvPortMalloc (). When RAM is being freed, instead of calling free (), the RTOS kernel calls vPortFree ().

FreeRTOS offers several heap management schemes that range in complexity and features. It is also possible to provide your own heap implementation, and even to use two heap implementations simultaneously. Using two heap implementations simultaneously permits task stacks and other RTOS objects to be placed in fast internal RAM, and application data to be placed in slower external RAM.

Memory Allocation Implementation

The FreeRTOS download includes five sample memory allocation implementations, each of which are described in the following subsections. The subsections also include information on when each of the provided implementations might be the most appropriate to select.

Each provided implementation is contained in a separate source file (heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c respectively) which are located in the `Source/Portable/MemMang` directory of the main RTOS source code download. Other implementations can be added as needed. Exactly one of these source files should be included in a project at a time [the heap defined by these portable layer functions will be used by the RTOS kernel even if the application that is using the RTOS opts to use its own heap implementation].

Following below:

heap_1 - the very simplest, does not permit memory to be freed

heap_2 - permits memory to be freed, but not does coalescence adjacent free blocks.

heap_3 - simply wraps the standard malloc () and free () for thread safety

heap_4 - coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option

heap_5 - as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas

4.6 Porting of FreeRTOS to MK64FN1M0VDC12 (Hexiwear)

The sections 4-1 to 4-5 covers all the basics of FreeRTOS. Now it is possible to switch to second thesis objective that is Porting of FreeRTOS to MK64FN1M0VDC12. Since This MCU uses Cortex-M4 ARM Core so some board specific implementations are required. We use GCC compiler for compilation of Zerynth VM. In the following sections there is complete description of the steps and necessary configurations of FreeRTOS on K64 MCU.

4.6.1 Cortex-M4 FreeRTOS port specific configuration

Interrupt configuration on Cortex-M processors is confusing, and prone to error. To assist your development, the FreeRTOS Cortex-M ports automatically check the interrupt configuration, but only if configASSERT () is defined. ConfigASSERT ()

The ARM Cortex cores, and ARM Generic Interrupt Controllers (GICs), use numerically low priority numbers to represent logically high priority interrupts. This can seem counter-intuitive, and is easy to forget. If you wish to assign an interrupt a logically low priority, then it must be assigned a numerically high value. If you wish to assign an interrupt a logically high priority, then it must be assigned a numerically low value.

The Cortex-M interrupt controller allows a maximum of eight bits to be used to specify each interrupt priority, making 255 the lowest possible priority. Zero is the highest priority. However, Cortex-M microcontrollers normally only implement a subset of the eight possible bits. The number of bits actually implemented is dependent on the microcontroller family.

For MK64FN1M0VDC12 4-bits of the eight possible bits has been implemented, it is only the most significant bits of the byte that can be used—leaving the least significant bits unimplemented. Unimplemented bits can take any value, but it is normal to set them to 1. This is demonstrated by Figure 4-14, which shows how a priority of binary 101 is stored in a MK64FN1M0VDC12 microcontroller that implements four priority bits.

Figure 4-14 Cortex-M 4 priority bits Implementation

In Figure 4-14 the binary value 101 has been shifted into the most significant four bits because the least significant four bits are not implemented. The unimplemented bits have been set to 1.

Some library functions expect priority values to be specified after they have been shifted up into the implemented (most significant) bits. When using such a function the priority shown in Figure 4-14 can be specified as decimal 95. Decimal 95 is binary 101 shifted up by four to make binary 101nnnn (where ‘n’ is an unimplemented bit), and with the unimplemented bits set to 1 to make binary 1011111.

Some library functions expect priority values to be specified before they have been shifted up into the implemented (most significant) bits. When using such a function the priority shown in Figure 4-14 must be specified as decimal 5. Decimal 5 is binary 101 without any shift.

ConfigMAX_SYSCALL_INTERRUPT_PRIORITY-configKERNEL_INTERRUPT_PRIORITY must be specified in a way that allows them to be written directly to the Cortex-M registers, so after the priority values have been shifted up into the implemented bits. ConfigKERNEL_INTERRUPT_PRIORITY must always be set to the lowest possible interrupt priority. Unimplemented priority bits can be set to 1, so the constant can always be set to 255, no matter how many priority bits are actually implemented. Cortex-M interrupts will default to a priority of zero—the highest possible priority. The implementation of the Cortex-M hardware does not permit configMAX_SYSCALL_INTERRUPT_PRIORITY to be set to 0, so the priority of an interrupt that uses the FreeRTOS API must never be left at its default value.

4.6.2 FreeRTOSConfig.h and Port.c Configuration

FreeRTOSConfig.h

The very first step was to configure FreeRTOSConfig.h with MK64FN1M0VDC12 specific parameters. Following are the changes made for in this file for target platform:

```
#define configTICK_RATE_HZ   1000  /** frequency of tick interrupt */
#define configCPU_CLOCK_HZ  SystemCoreClock  /** CPU core clock frequency (in Hz) */
#define configBUS_CLOCK_HZ  SystemBusClock  /** Bus clock frequency (in Hz) */
#define configSYSTICK_CLOCK_HZ              ((configCPU_CLOCK_HZ)/configSYSTICK_CLOCK_DIVIDER)  /** frequency of system
tick counter */
#define configMINIMAL_STACK_SIZE            200  /** stack size in addressable stack units */
#define configMAX_PRIORITIES                255
#define configCOMPILER                  configCOMPILER_ARM_GCC
#define configCPU_FAMILY                configCPU_FAMILY_ARM_M4F
#define CORTEX_PRIORITY_SVCALL        (CORTEX_MAXIMUM_PRIORITY + 1)
#define CORTEX_MAX_KERNEL_PRIORITY     (CORTEX_PRIORITY_SVCALL + 1)
#define CORTEX_PRIORITY_PENDSV        CORTEX_MAX_KERNEL_PRIORITY
* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY                                 0x7


/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY            5


/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY                 CORTEX_MAX_KERNEL_PRIORITY //(
configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!! */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY          ( CORTEX_PRIORITY_SVCALL ) //0x10
```

ARM Cortex-M3, ARM Cortex-M4 and ARM Cortex-M4F ports need FreeRTOS handlers to be installed on the SysTick, PendSV and SVCCall interrupt vectors. The vector table can be populated directly with the FreeRTOS defined xPortSysTickHandler(), xPortPendSVHandler() and vPortSVCHandler() functions respectively, or, if the interrupt vector table is CMSIS compliant, the following three lines can be added to FreeRTOSConfig.h to map the FreeRTOS function names to their CMSIS equivalents:

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

Using #defines in this way will only work if the default handlers provided by your development tools are defined as weak symbols. If the default handlers are not defined as week symbols they must be either commented out or deleted.

4.6.3 Static Memory Allocation by Garbage Collector

Zerynth VM uses Static memory allocation scheme to assign memory for underlying RTOS primitives like Tasks, semaphores and Mutexes etc. it uses a garbage collector peace of code to assign memory for Task Control Block (TCB) and Task Stack.

******Garbage Collector**:***

Objects in Python have lifecycle. They are created and used by the programmer and must be removed when they are not needed anymore. While in low level languages the responsibility of freeing unused memory rests on the programmer, in Python it's the garbage collector (GC) duty. When necessary, a complete scan of the created object is performed in order to search

the ones that can be removed safely. The VM GC algorithm is a mark-and-sweep-stop-the-world variant.

When there isn't enough memory available to satisfy an allocation request from the pool of allocated heap blocks, the runtime system invokes the garbage collector (GC). A Python program can't explicitly free a value when it is done with it. Instead, the GC regularly determines which values are *live* and which values are *dead*, i.e., no longer in use. Dead values are collected and their memory made available for reuse by the application.

The GC doesn't keep constant track of values as they are allocated and used. Instead, it regularly scans them by starting from a set of *root* values that the application always has access to (such as the stack). The GC maintains a directed graph in which heap blocks are nodes, and there is an edge from heap block `b1` to heap block `b2` if some field of `b1` is a pointer to `b2`.

All blocks reachable from the roots by following edges in the graph must be retained, and unreachable blocks can be reused by the application. The algorithm used by Zerynth GC to perform this heap traversal is commonly known as *mark and sweep* garbage collection.

4.6.4 Uplinker support

Once Zerynth virtual machine is installed on target platform, a user can program its application code written in Zerynth studio using python. The byte code can then be programmed to using uplink support of Zerynth studio. Normally a serial, whether USB or UART, interface is dedicated on target board for uploading the byte code.

Chapter 5

5. Testing and Results

5.1 Testing Methodology

FreeRTOS porting on Hexiwear creates a possibility for user to use both Real time OS on Zerynth Virtual machine. However it is important to test and compare some real time aspects of both operating systems which can become significant in certain scenarios. For Testing the following methodogly is adopted shown in Figure 5-1, starting from Test Case implementation, running those tests on hardware platform, acquiring data and analyze data. Data acquisition was done using Saleae logic analyzer which can capture input signals @ up to 24 MS/s which is quite sufficient for this kind of testing. Finally, the data is analyzed in MATLAB for results and conclusions.

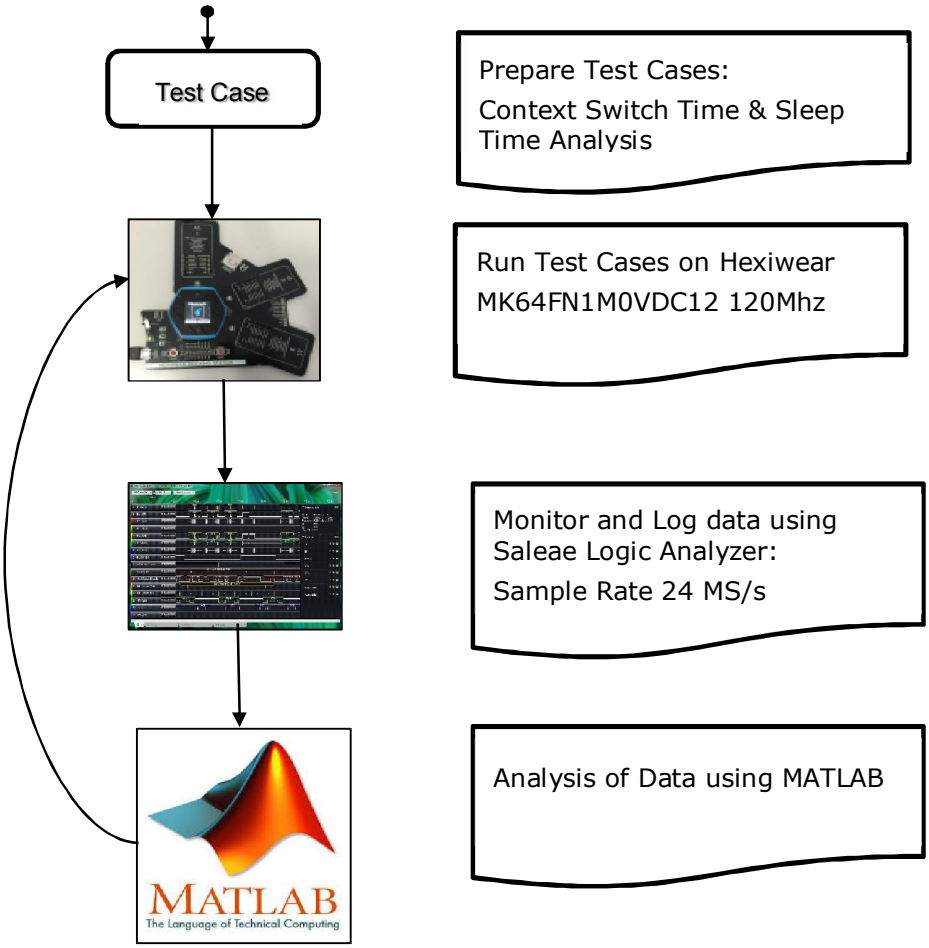


Figure 5-1 Testing Methodology

5.1.1 Test Criteria Specification

Testing all aspects of a real time operating system is difficult however some aspects can be tested comfortably and these aspects can play important role in selection of real time OS for

certain applications. Following are some features tested on Hexiwear with both operating systems

- Context Switch Time using C
- Context switch time using Python
- Interrupt Latency time
- Thread Sleep Time

For each of the above testing scenarios there is a test case to run on target board.

5.1.2 Test Case for Context Switch Time on Hexiwear with C

Context switch in a real time OS can be an overhead in certain safety critical applications. The Figure 5.2 shows the test case for context switching. Two tasks are selected of equal priority with two binary semaphores with initial value set to 0. So Task 1 will block on semaphore 1 and wait for signal from Task 2. Similarly Task 2 will block on semaphore 2 and wait for signal command from Task 1. Blocking on semaphore cause the scheduler to cause context switch from current task to other. A GPIO is toggled in such a way that its HIGH time measures the *Context Switch Time* from Task1 to Task 2 and its LOW time measures for vice versa.

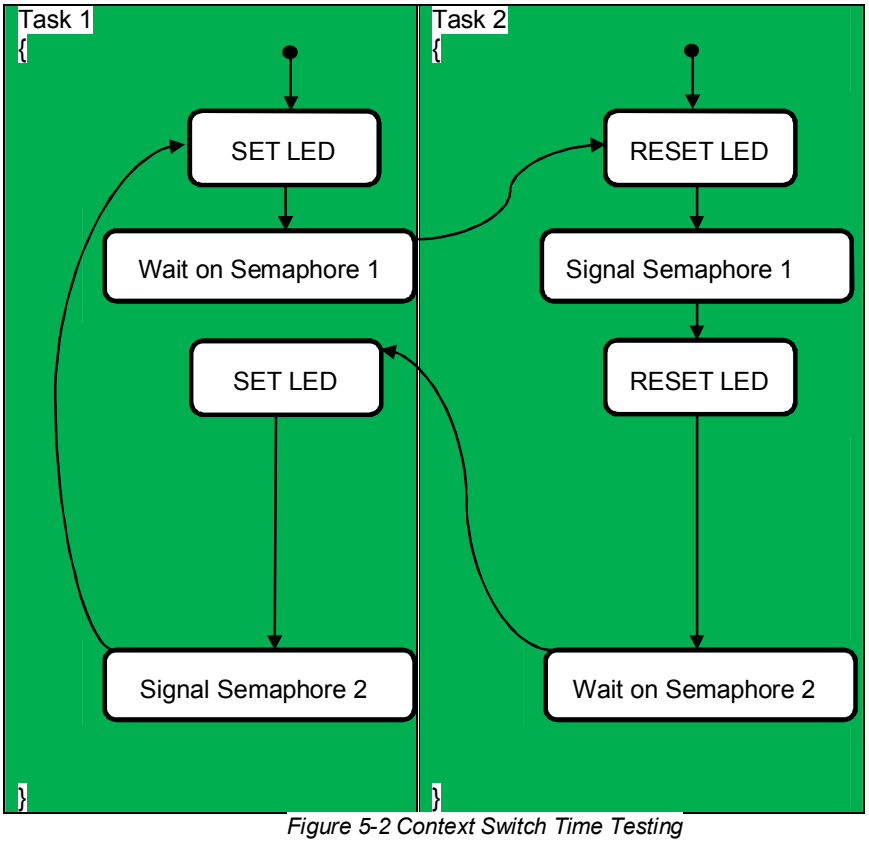


Figure 5-2 Context Switch Time Testing

Following Figure 5-3 shows the GPIO status obtained from Saleae logic analyzer.

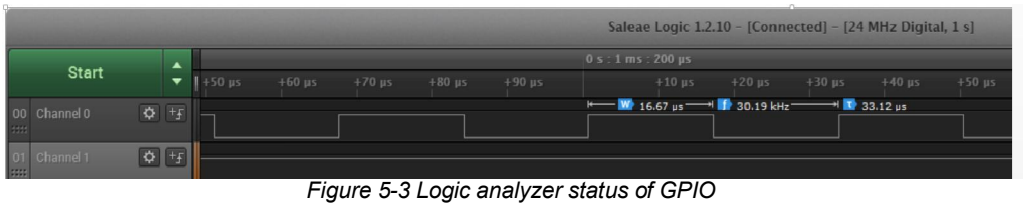


Figure 5-3 Logic analyzer status of GPIO

This test is executed using both RTOS on Hexiwear for comparison.

5.1.2 Test Case for Context Switch Time on Hexiwear with Python

Python code is written in application layer which uses the services of RTOS and VHAL layer. Simply toggling LED will take more time than in C because it calls some other functions in between toggling the LED from python and actual physical toggling occurs. Since application user has to write code in this layer so context switching delay can be more significant for certain applications. The same scenario is repeated for this testing as did in C.

The Appendix A shows python code for Context switch.

5.1.4 Test Case for Interrupt Latency Testing Using C

Interrupt latency is "*the time between actual interrupt occurs and when it's serviced"* is another important parameter to test in real time embedded systems. Test case for this scenario was prepared using another board *Particle Photon* which simply outputs a square wave of

- 10ms period and 50% duty cycle.
- 4 uS period and 50% duty cycle
- 2uS period and 50% duty cycle

This signal was fed into Hexiwear docking station at virtual pin D24 (PC16 on docking station) which serves as interrupt pin. This pin was configured as to serve the rising edge interrupts. Another pin on Hexiwear D23 (PC17 on docking station) was configured as output. This pin was toggled in ISR function related to interrupt. The time is measured between rising edge of pin D24 and any edge of D23 as shown Figure 5-4. This is interrupt latency time.

The Appendix A shows python code for Particle photon to generate PWM wave with 50% duty cycle.

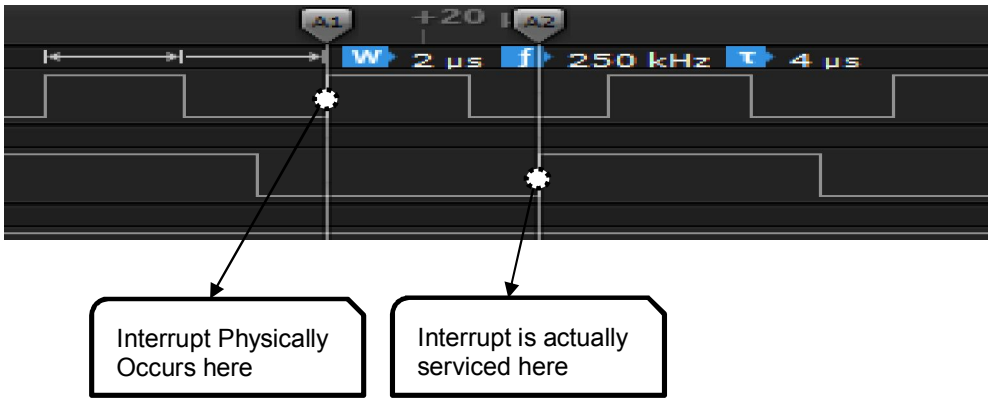


Figure 5-4 Interrupt Latency

5.1.5 Test Case for Sleep Time Analysis

Tasks in real time OS can be blocked to sleep state for certain specific time. Sleep Time is analysis is also important to measure and compare the exact sleep time that task is blocked and when it is awaked. It gives another performance measure to select an operating system for specific application. This test was carried out on both RTOSs FreeRTOS and Chibi2 with following test cased

- Single task is blocked for 1000ms and toggles a GPIO when awaked
- Single task is blocked for 500ms and toggles a GPIO when awaked
- Single task is blocked for 100ms and toggles a GPIO when awaked
- Single task is blocked for 5ms and toggles a GPIO when awaked
- Single task is blocked for 1ms and toggles a GPIO when awaked

The results are obtained using logic analyzer and later on analyzed using MATLAB for comparison.

Following Figure 5-5 shows Sleep Time for 1ms on chibi2 OS.

Figure 5-5 Sleep Time for 1ms

5.2 Results

5.2.1 Performance Results of Context Switch Time with Both RTOS

Context switch was experimented using both C and Python each RTOS. Following Figure 5-6 shows the context switch of FreeRTOS using C language. FreeRTOS has one Parameter in its FreeRTOSConfig.h which is stack overflow. It is likely that the stack will reach its greatest (deepest) value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.

This method is quick but not guaranteed to catch all stack overflows. Set configCHECK_FOR_STACK_OVERFLOW to 1 to use this method only. Stack overflow checking introduces a context switch overhead so its use is only recommended during the development or testing phases.

The Test code is available in Appendix B.

Using C

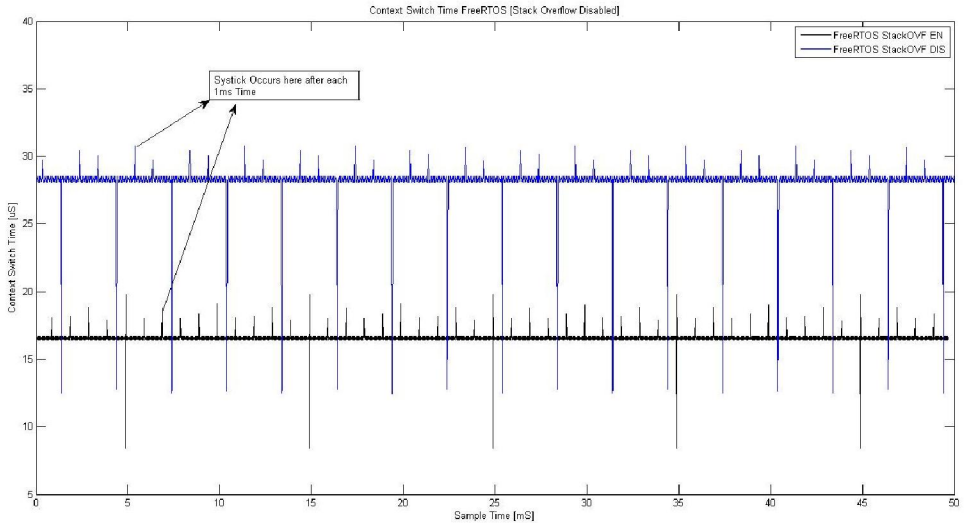


Figure 5-6 Context Switch FreeRTOS using C

The results shows that with stack overflow checking is enabled the context switch overhead in FreeRTOS reaches to 28us and it is around 16.5us when stack over flow is disabled. The small peaks at every 1ms shows the trigger of systick timer. The following Figure5-7 shows the context switching within two tasks using chibi2 OS and in C language.

The high time of GPIO says:

High Time = CST Time (1→2) + Setting GPIO twice

Setting GPIO from C takes 0.4375us so total 0.875us we have to subtract from

CST Time (1→2) = High Time - Setting GPIO twice Time

CST Time (1→2) = 28us – 0.875us = ***27.125us*** *(Stack overflow Enabled)*

Resetting Time of GPIO from C takes also 0.4375us

CST Time (2→1) = 28us – 0.875us = ***27.125us*** *(Stack overflow Enabled)*

When stack overflow disabled the CST is

CST Time = 16.5us – 0.875us = ***15.625us*** *(Stack overflow Disabled)*

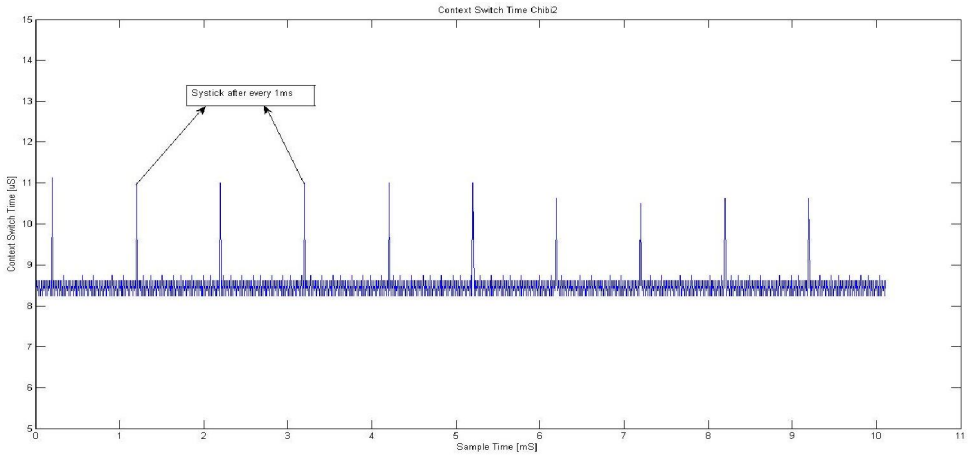


Figure 5-7 Context Switch using Chibi2 [C Language]

Context switch using chibi2 and C language is around 8.5us which is almost two times faster than in FreeRTOS. Systick behavior can be noticed in this figure also at every 1ms the context switch is little longer.

CST Time = 8.5us – 0.875us = ***7.625us***

Using Python

The following Figure 5-8 shows the context switch using python with both RTOS. Since python layer adds some extra delay before the actual context is switched which is reflected in figure. The CST for FreeRTOS is around 180us while for chibi2 is around 135us. However the CST from Task1-to-2 and from Task2-to-1 is different. This was not the case using C language. The fact is that Task 1 is setting up GPIO while Task 2 resetting the same GPIO. In python, there are some extra functionality to be called when setting the GPIO than resetting it. The amount of time recorded for setting GPIO is 12us from python while resetting it takes 8.125us. Since in each task setting or resetting takes place twice before the actual context switch i.e.

High Time = CST Time (1→2) + Setting GPIO twice

High Time = CST Time (1→2) + 24us, similarly

Low Time = CST Time (2→1) + Resetting GPIO twice

Low Time= CST Time (2→1) + 16.25 us

Difference = 24 – 16.25 = ***7.75us***

This difference can be seen in the figure for context switching in both FreeRTOS and chibi2 OS.

So for FreeRTOS

CST Time (1→2) = High Time - Setting GPIO twice Time

CST Time (1→2) = 184us – 24us = ***160us***

CST Time (2→1) = Low Time - Resetting GPIO twice Time

CST Time (2→1) = 177us – 16.25us = ***160.75us***

For chibi2

CST Time (1→2) = High Time - Setting GPIO twice Time

CST Time (1→2) = 141us – 24us = ***117us***

CST Time (2→1) = Low Time - Resetting GPIO twice Time

CST Time (2→1) = 134us – 16.25us = ***117.75us***

So clearly it is evdient that chibi2 is 26% better in context switching for user application than Freertos9.

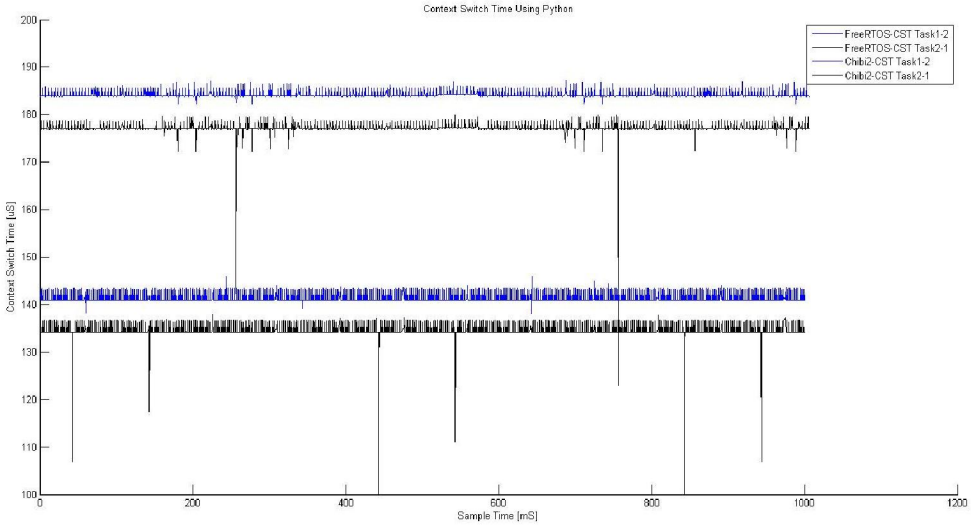


Figure 5-8 CST Using Python

5.2.2 Interrupt Latency Time results

Interrupt latency time is recorded and analyzed in this test for both OS using C and Python.

Using C

Following Figure5-9 shows the interrupt latency for both RTOS.

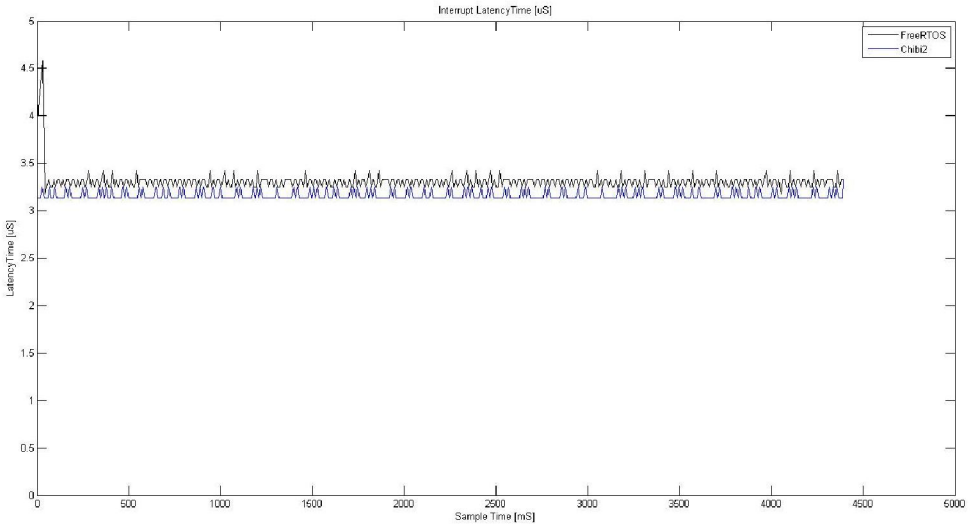


Figure 5-9 Interrupt Latency Using C

This shows that interrupt latency is almost equal in both RTOS using C.

INTERRUPT LATECY (FreeRTOS9) = ***3.3us***

INTERRUPT LATECY (Chibi2) = ***3.25us***

So interrupts that are occurring at frequency higher than 303 KHz will not be serviced properly like in the following figure5-10 a square wave of 500 KHz frequency with 2us period was used as an interrupt source the ISR behavior was totally disturbed and it was not following the positive edge of interrupt source.

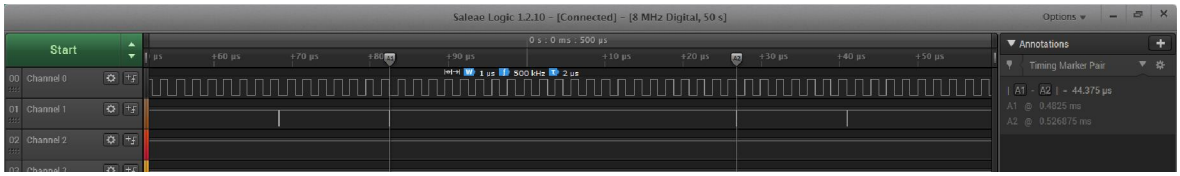


Figure 5-10 500 KHz interrupt source

Using Python

The Figure5-11 shows that there is great difference in interrupt latency when measures using python.

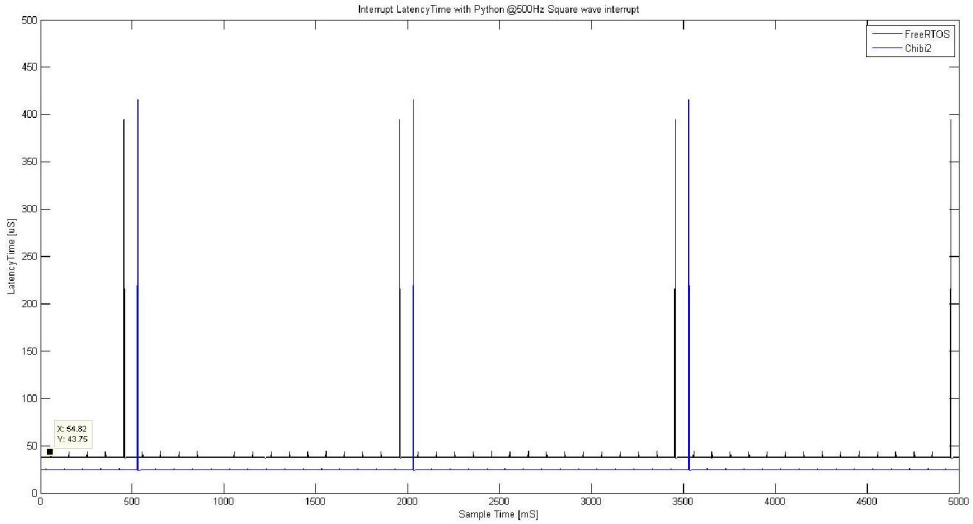


Figure 5-11 Interrupt Latency with Python @500Hz

INTERRUPT LATECY (FreeRTOS9) = ***37.75us***

INTERRUPT LATECY (Chibi2) = ***24.88us***

The small peaks after every 100ms in the Figure 5-11 shows that python scheduler is called after 100ms and latency time is takes longer than usual and high peaks after every 1.5 seconds shows that garbage collector in python is called at this rate and it is used to free the unused memory. At these points latency time increased to a considerably very large values.

For FreeRTOS the maximum interrupt source should not exceed 26.5 KHz while for chibi2 the interrupt source should not exceed 40.2 KHz.

5.2.3 Performance Results with Sleep Time results

Sleep Time of task in FreeRTOS and chibi2 are recorded and plotted as shown in the Figure5-12 and Figure 5-13.

This test was carried out for around 50 seconds with different sleep time of a task and it was observed that both Oss are following exact sleep time.

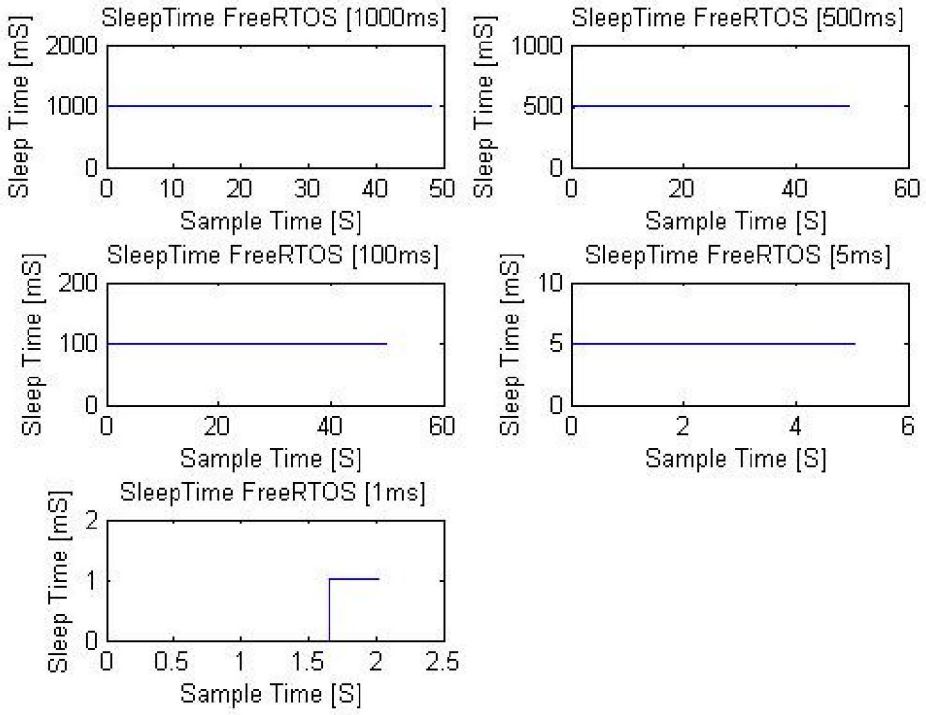


Figure 5-12 Sleep Time FreeRTOS

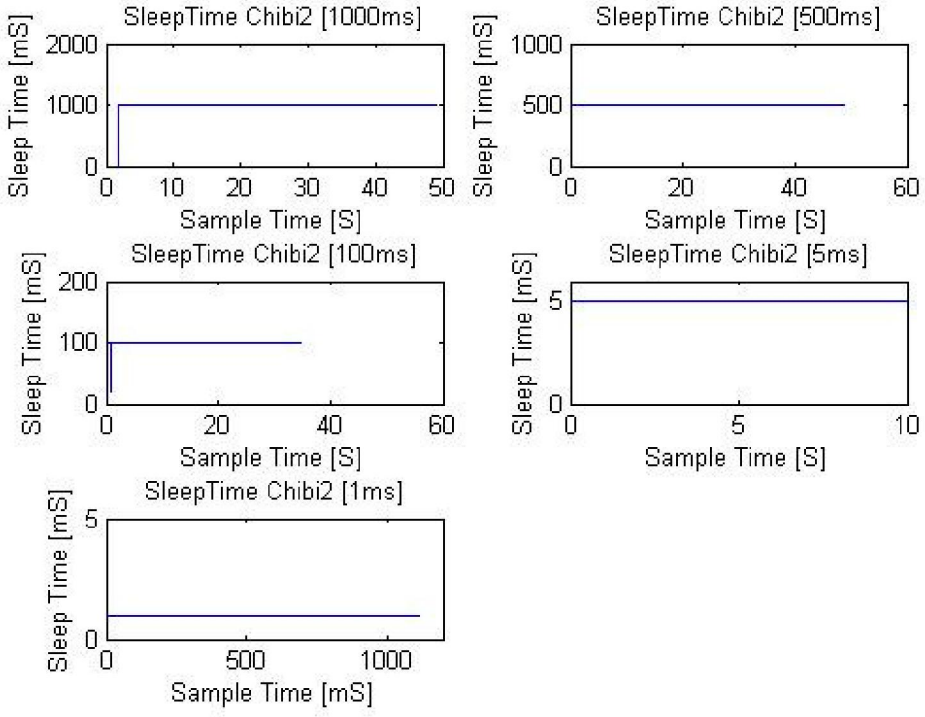


Figure 5-13 Sleep Time Chibi2

Appendix A

Python code for Context Switch

```
import streams  # import the streams module
import threading

# create a stream linked to the default serial port
streams.serial()

pinMode(D23,OUTPUT)
pinMode(D24,OUTPUT)

digitalWrite(D23,0)
digitalWrite(D24,0)

sem0 = threading.Semaphore(0)
sem1 = threading.Semaphore(0)

def th(semA,semB,pin,n):
if n==0:
while True:
        digitalWrite(pin,0)
        semA.release()
        digitalWrite(pin,0)
        semB.acquire()

else:
while True:
        digitalWrite(pin,1)
        semA.acquire()
        digitalWrite(pin,1)
        semB.release()


thread(th,sem0,sem1,D23,1)
thread(th,sem0,sem1,D23,0)

while True:
        sleep(10000)
```

Python Code for Particle Photon to generate PWM

```
import streams
import pwm

#create a serial port stream with default parameters
streams.serial()

# the pin where the buzzer is attached to
buzzerpin = D2.PWM

pinMode(buzzerpin,OUTPUT) #set buzzerpin to output mode
frequency=30000         #define a variable to hold the played tone frequency

while True:
```

```
        period=1000000//frequency #we are using MICROS so every sec is 1000000 of micros. // is the int division, pwm.write period doesn't accept floats
        print("frequency is", frequency,"Hz")

  #set the period of the buzzer and the duty to 50% of the period
  pwm.write(buzzerpin,period,period//2,MICROS)

  sleep(10000)
```

Python Code for Hexiwear for Interrupt Latency

```
import streams
# define where the button and the LED are connected
# in this case BTN0 will be automatically configured according to the selected board button
# change this definition to connect external buttons on any other digital pin
buttonPin=D24
ledPin=D23  # LED0 will be configured to the selected board led

# configure the pin behaviour to drive the LED and to read from the button
pinMode(buttonPin,INPUT)
pinMode(ledPin,OUTPUT_PUSHPULL)

# define the function to be called when the button is pressed
def pressed():
    pinToggle(ledPin)

# attach an interrupt on the button pin and call the pressed function when it falls
# being BTN0 configured as pullup, when the button is pressed the signal goes to from HIGH to LOW.
# opposite behaviour can be obtained with the equivalent "rise" interrupt function: onPinRise (pin,fun)
# hint: onPinFall and onPinRise can be used together on the same pin, even with different functions
onPinRise(buttonPin,pressed)
```

Appendix B

Context Switch Time Analysis MATLAB Code

```
clc;
clear;
close all;
%=========================CST FreeRTOS StackOVF Enabled=========
CST = csvread('FreeRTOS-ContextSwitch-stackoverflow-Enabled.csv',1,10,[1 10
3000 10]);
Time = (csvread('FreeRTOS-ContextSwitch-stackoverflow-Enabled.csv',1,0,[1 0
3000 0])) * 1000.0;
figure;
plot(Time,CST,'color','k');hold on;
xlim([0 50]);
ylim ([5 30]);
xlabel('Sample Time [mS]');
ylabel('Context Switch Time [uS]');
title('Context Switch Time FreeRTOS [Stack Overflow Enabled]');

%%===============CST FreeRTOS StackOVF Disabled====================
CST = csvread('FreeRTOS-ContextSwitch-stackoverflow-Disabled.csv',1,10,[1 10
3000 10]);
Time = (csvread('FreeRTOS-ContextSwitch-stackoverflow-Disabled.csv',1,0,[1 0
3000 0])) * 1000.0;
plot(Time,CST);
legend('FreeRTOS StackOVF EN','FreeRTOS StackOVF DIS');
xlim([0 50]);
ylim ([5 40]);
xlabel('Sample Time [mS]');
ylabel('Context Switch Time [uS]');
title('Context Switch Time FreeRTOS [Stack Overflow Disabled]');

%%=================CST Chibi2=========================================
CST = (csvread('Chibi2-ContextSwitch-Time.csv',1,1,[1 1 1193 1])) *1000000.0;
Time = (csvread('Chibi2-ContextSwitch-Time.csv',1,0,[1 0 1193 0])) * 1000.0;
figure;
plot(Time,CST);
xlim([0 11]);
ylim ([5 15]);
xlabel ('Sample Time [mS]');
ylabel ('Context Switch Time [uS]');
title ('Context Switch Time Chibi2');
%%====================CST FreeRTOS using python=====================
clear all;
M = (csvread('ContextSwitch-TimeUsingPython.csv',3,0,[3 0 7122 3]));
Time = M(1:5452,1)* 1000.0;
Time2 = M(1:7119,3)*1000.0;
r=1;
c=1;
CST_freertos = M(1:5452,2) * 1000000.0;
CST_chibi2 = M(1:7119,4) * 1000000.0;
for i=1:5452
    if M(i,2) < 180.0e-06
       CSTLow(r,1) = CST_freertos(i,1);
       TimeL(r,1) =  Time(i,1);
       r=r+1;
    else if  M(i,2) > 180.0e-06 &&  M(i,2) < 190.0e-06
           CSThigh(c,1) = CST_freertos(i,1);
           TimeH(c,1) =  Time(i,1);
           c=c+1;
```

```
        end
    end
end
figure; hold on;
plot(TimeH, CSThigh,'b');
plot(TimeL,CSTLow,'k');

r=1;
c=1;
for i=1:7119
    if M(i,4) < 138.0e-06
       CSTLow(r,1) = CST_chibi2(i,1);
       TimeL(r,1) =  Time2(i,1);
       r=r+1;
    else if  M(i,4) > 138.0e-06 &&  M(i,4) < 150.0e-06
           CSThigh(c,1) = CST_chibi2(i,1);
           TimeH(c,1) =  Time2(i,1);
           c=c+1;
        end
    end
end
plot(TimeH, CSThigh,'b');
plot(TimeL,CSTLow,'k');
legend('FreeRTOS-CST Task1-2','FreeRTOS-CST Task2-1','Chibi2-CST Task1-
2','Chibi2-CST Task2-1');
xlim([0 1200]);
ylim ([100 200]);
xlabel('Sample Time [mS]');
ylabel('Context Switch Time [uS]');
title('Context Switch Time Using Python');
```

Interrupt Latency Time Analysis MATLAB Code

```
%==================INTERRUPT LATENCY TESTING=============
%%%%%%%%%%%%%%%%%%%%%%%%USING C%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;

M = csvread('InterruptLatencyUsingC.csv',3,0,[3 0 1321 3]);
r=1;
for i=1:1318
    if M(i,2) < 6.0e-06
       LatencyTime(r,1) = M(i,2);
       Time(r,1) =  M(i,1);
       r=r+1;
    end
end
figure;
plot(Time*1000.0,LatencyTime* 1000000.0,'k');
hold on;
xlim([0 5000]);
ylim ([0 5]);
xlabel('Sample Time [mS]');
ylabel('LatencyTime [uS]');
title('Interrupt LatencyTime FreeRTOS [uS]');
%=========================Latency Time Chibi2======================
r=1;
for i=1:1318
    if M(i,4) < 6.0e-06
       LatencyTime(r,1) = M(i,4);
       Time(r,1) =  M(i,3);
       r=r+1;
    end
end
```

```
plot(Time*1000.0,LatencyTime* 1000000.0);
legend('FreeRTOS','Chibi2');
xlim([0 5000]);
ylim ([0 5]);
xlabel('Sample Time [mS]');
ylabel('LatencyTime [uS]');
title('Interrupt LatencyTime [uS]');
clear all;
%==================INTERRUPT LATENCY TESTING=============
%%%%%%%%%%%%%%%%%%%%%%%%%%%PYTHON%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;

M = csvread('InterruptLatencyUsingPython.csv',2,0,[2 0 7522 3]);
r=1;
for i=1:7520
    if M(i,2)==1
        if M(i+1,2) == 1
           LatencyTime(r,1) = M(i+1,1) - M(i,1);
           Time(r,1) =  M(i,1);
           r=r+1;
        end
    end
end
figure;
plot(Time*1000.0,LatencyTime* 1000000.0,'k');
hold on;
xlim([0 5000]);
ylim ([0 500]);
xlabel('Sample Time [mS]');
ylabel('LatencyTime [mS]');
title('Interrupt LatencyTime FreeRTOS with Python [mS]');
%=========================Latency Time Chibi2======================
r=1;
for i=1:7520
    if M(i,4)==1
        if M(i+1,4) == 1
           LatencyTime(r,1) = M(i+1,3) - M(i,3);
           Time(r,1) =  M(i,3);
           r=r+1;
        end
    end
end
plot(Time*1000.0,LatencyTime* 1000000.0);
legend('FreeRTOS','Chibi2');
xlim([0 5000]);
ylim ([0 500]);
xlabel('Sample Time [mS]');
ylabel('LatencyTime [uS]');
title('Interrupt LatencyTime with Python @500Hz Square wave interrupt');
clear all;
```

Sleep Time Analysis MATLAB Code

```
%%==================Chibi2 Sleep Time Testing=============
%=========================================================
clear all;
M = csvread('Chibi2-SleepTimeTesting.csv',3,0,[3 0 1711 9]);
Time = M(1:49,1);
SleepTime = M(1:49,2) *1000.0;
figure;
subplot(3,2,1);
plot(Time,SleepTime);
xlim([0 50]);
ylim ([0 2000]);
```

```
xlabel('Sample Time [S]');
ylabel('Sleep Time [mS]');
title('SleepTime Chibi2 [1000ms]');
%==================500ms=========================
Time = M(1:98,3);
SleepTime = M(1:98,4) *1000.0;
subplot(3,2,2);
plot(Time,SleepTime);
xlim([0 60]);
ylim ([0 1000]);
xlabel('Sample Time [S]');
ylabel('Sleep Time [mS]');
title('SleepTime Chibi2 [500ms]');
```

6. Bibliography

1. **L.Bass, P.Clements, R.Kazmann.** *Software Architecture in Process.* 2003.

2. **Madau, Dinu P.** An Architecture for designing resuable embedded systems software. *Embedded.* [Online] 01 06, 2017. http://www.embedded.com/design/prototyping-and-development/4007567/.

3. *An Overview of Real-Time Operating Systems.* **Walter Cedeño, Ph.D., Phillip A. Laplante, Ph.D.** Malvern PA : s.n., 2007.

4. **Margull, Dr Ulrich.** Development Methodologies for Automotive Systems. Ingolstadt : s.n., 2016.

5. **Wikipedia.** Operating system abstraction layer. *https://en.wikipedia.org.* [Online] October 14, 2016. https://en.wikipedia.org/wiki/Operating_system_abstraction_layer.

6. *A Full Stack for Quick Prototyping of IoT Solutions.* **Daniele Mazzei, Giacomo Baldi,Gabriele Montelisciani, Gualtiero Fantoni.** Paris : s.n., 2016.

7. **Team, Zerynth.** documents . *doc.zerynth.com.* [Online] [Cited: 01 09, 2017.] http://doc.zerynth.com/.

8. —. zerynth.com. *docs.zerynth.com.* [Online] zerynth. [Cited: 01 09, 2016.] https://docs.zerynth.com/latest/official/core.zerynth.toolchain/docs/index.html.

9. **Wikipedia.** industrie 4.0. *en.wikipedia.org.* [Online] Wikipedia, December 31, 2016. [Cited: Januaray 12, 2017.] https://en.wikipedia.org/wiki/Industry_4.0.

10. **mbed.** Hexiwear. *developer.mbed.org.* [Online] August 19, 2016. [Cited: Januray 12, 2017.] https://developer.mbed.org/platforms/Hexiwear/.

11. **Mikroe.** Hexiwear. *docs.mikroe.com.* [Online] Mikroe, October 20, 2016. [Cited: January 12, 2017.] http://docs.mikroe.com/Hexiwear.

12. **Barry, Richard.** Mastering the FreeRTOS™ Real Time Kernel. 161204 Pre-Release, 2016.

13. **FreeRTOS.** About FreeRTOS. *www.freertos.org.* [Online] Real Time Emgineers Ltd. [Cited: 01 18, 2016.] http://www.freertos.org/RTOS.html.

14. **Jason Hickey, Anil Madhavapeddy and Yaron Minsky.** Understanding the Garbage Collector. *https://realworldocaml.org.* [Online] Real World OCaml, 2012. [Cited: 01 23, 2017.] https://realworldocaml.org/v1/en/html/understanding-the-garbage-collector.html.