**Research Report**

Real Time System

**Installing and using ChibiOS on STM32F4Discovery Kit**

Author: Ta Duy Hung – 20101682

Bui Tri Trung – 20102365

Lecturer: Dr. Ngo Lam Trung

**TABLE OF CONTENTS**

1 INTRODUCTION 4

1.1 PREFACE 4

1.2 TASK LIST 4

2 CONTENTS 5

2.1 Overview of RTOS 5

2.1.1 RTOS Definition 5

2.1.2 Kernel Components of a RTOS 5

2.1.3 RTOS Key characteristic 6

2.2 STM32F4 Discovery 6

2.2.1 Overview 6

2.2.2 Hardware System 7

2.2.3 Programming on STM32F4 8

2.3 ChibiOS 10

2.3.1 Feature 10

2.3.2 Performance Data 10

2.3.3 Supported Ports 11

2.3.4 ChibiOS General Architecture 11

2.3.5 Thread Management in ChibiOS 13

2.4 REQUIRED TOOLS 15

2.4.1 Tools 15

2.4.2 Setup 15

2.5 INSTALL CHIBIOS ON STM32F4 17

2.5.1 ChibiOS Source Code 17

2.5.2 Load firmware onto STM32F4 17

2.5.3 Debugging 18

2.6 BUILDING A DEMO PROGRAM RUNNING ON STM32F4 19

2.6.1 LED blinking demo program 19

2.6.2 Program Implementation 19

3 CONCLUSION 21

**LIST OF FIGURE**

Figure 1: RTOS kernel components 5

Figure 2: STM32Discovery Board 7

Figure 3: Hardware block diagram 8

Figure 4: An example of IAR debug session in progress 9

Figure 5: ChibiOS general architectures 12

Figure 6: Thread state changes 13

Figure 7: ChibiOS thread list 14

Figure 8: GCC Command 15

Figure 9: Running *st-util* utility 16

Figure 10: Using GDB to debug program 18

# INTRODUCTION

## PREFACE

In the past recent year, Real time Operating System (RTOS) is becoming more and more popular as electronic industry grows rapidly. Electronic devices are not restricted to just some particular jobs. They can now be considered some mini computer with significantly processing power.

A lot of important applications using RTOS have been implemented in various field including automobiles, robotics, communications etc. To give people who’d like to work in this area, some electronic companies have produced some micro controller unit for the purpose of giving beginner an overview about this field and an environment to simulate how these kinds of devices works as well as some fun to build application.

## TASK LIST

This report focuses on:

* Overview of STM32F4
* ChibiOS and its scheduling mechanism
* Installing ChibiOS on STM32F4
* Building an LED Blinking demo application using ChibiOS running on STM32F4

# CONTENTS

## Overview of RTOS

### RTOS Definition

A RTOS is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing embedded application code.

### Kernel Components of a RTOS

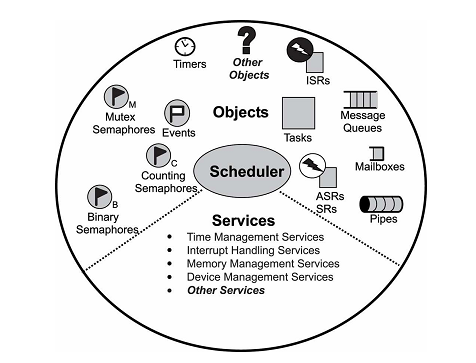


Figure 1: RTOS kernel components

* Scheduler: provides a set of scheduling algorithms to determine which task executes when. Two of the most approaches are preemption and round robin.
* Object: special kernel constructs to help developers create embedded applications. Some example: task, semaphores, messages queues.
* Service: kernel operations on objects such as Timing, Interrupt, Handling and Resource management.

### RTOS Key characteristic

* Reliability: some embedded systems need to work in a long periods without human intervention. If something bad were happened, it could take severe damage to the system. Thus, they require a high degree of reliability.
* Predicability: meeting time requirement is a key in any real time system to ensure proper operation.
* Performance: performance is important in everything. At least a system must be fast enough to fulfill time requirement.
* Compactness: In embedded system where hardware real estate is limited due to size and costs, the design of the system has to take that in mind so that the complete system is small in size but still efficient.
* Scalability: an RTOS can be used in a wide variety of embedded system. Therefore, they must be able to scale or down flexiblely to meet application specific requirement.

## STM32F4 Discovery

### Overview

STM32 is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. The STM32 chips are grouped into related series that are based around the same 32-bit ARM processor core, such as the Cortex-M7, Cortex-M4F, Cortex-M3, Cortex-M0+, or Cortex-M0. Internally, each microcontroller consists of the processor core, static RAM memory, flash memory, debugging interface, and various peripherals.

The STM32F4Discovery Board is a member of STM32 F4 series. It includes everything required for beginners and experienced users to get started quickly.

Based on the STM32F407VGT6, it includes an ST-LINK/V2 embedded debug tool, two ST MEMS, digital accelerometer and digital microphone, one audio DAC with integrated class D speaker driver, LEDs and push buttons and an USB OTG micro-AB connector.

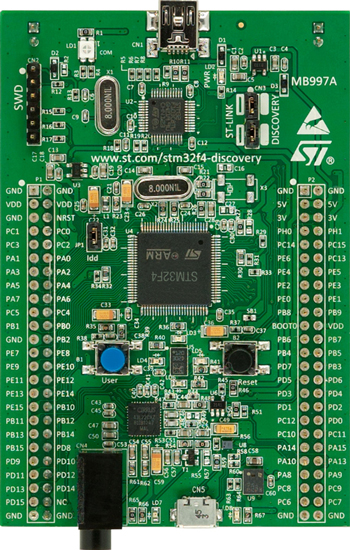


Figure 2: STM32Discovery Board

### Hardware System

* STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex-M4F core, 1 MB Flash, 192 KB RAM in an LQFP100 package
* On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
* Board power supply: through USB bus or from an external 5 V supply voltage
* External application power supply: 3 V and 5 V
* LIS302DL or LIS3DSH ST MEMS 3-axis accelerometer
* MP45DT02, ST MEMS audio sensor, omni-directional digital microphone
* CS43L22, audio Digital to Analog Converter (DAC) with integrated class D speaker driver
* Eight LEDs:
  + LD1 (red/green) for USB communication
  + LD2 (red) for 3.3 V power on
  + Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
  + USB OTG LEDs LD7 (green) VBus and LD8 (red) over-current
* Two push buttons (user and reset)
* USB OTG FS with micro-AB connector
* Extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing

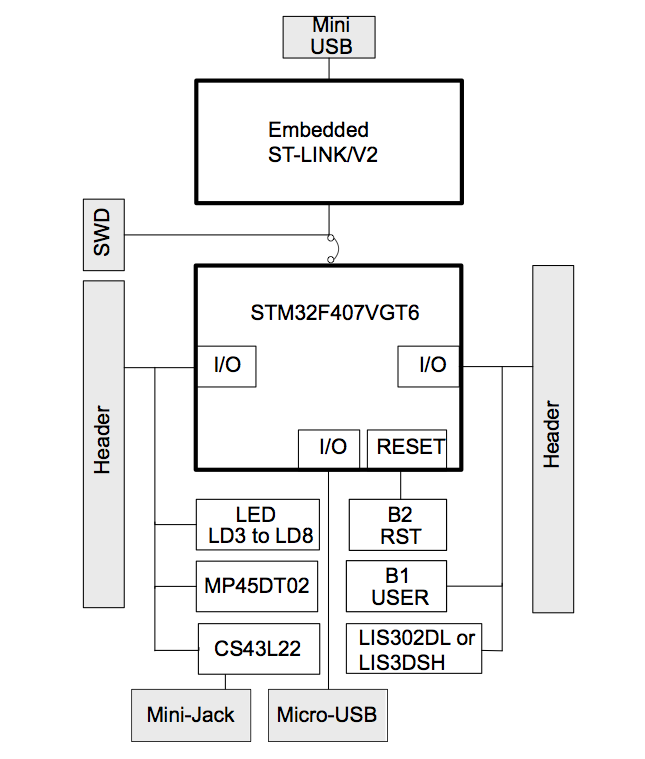


Figure 3: Hardware block diagram

### Programming on STM32F4

A program for a microcontroller is prepared on a personal computer (PC) using a suitable set of programs. First the source code of the program is written, than this code is translated into the machine code for a particular microcontroller, the STM32F407VG in our case. The developer then loads the code into the memory of the microcontroller. Several methods for loading are known. Most universal programmers (devices that fit between a PC and the target microcontroller, and allow the transfer of the code into the microcontroller) support JTAG method. The company ST introduced a method called serial wire debug (SWD), which uses fewer wires than JTAG method. The method SWD is supported by STM32F4DISCOVERY board, and the programmer that comes between a USB connector and the SWD pins of the microcontroller is available on the Discovery board.

The STM32F4DISCOVERY board hosts additional connector for SWD signals and can also be used to program external free standing microcontroller of the same family.

There are two popular IDE used for programming on STM32F4: Keil and IAR. They require at least these following files to create the machine code for the microcontroller:

* The user program, a text file written in “C” language.
* The file “startup\_stm32f4xx.s”, a text file written in assembly language; this file contains  instructions for the initial set-up of the microcontroller (stack, program counter, interrupt  vector table, initial system clock etc.; the description of the file is given in its header).
* The file “system\_stm32f4xx.c”, a text file written in “C”; this file contains functions for the detailed microcontroller set-up (system clock, clock distribution etc.; the description of this file is  given in its header).

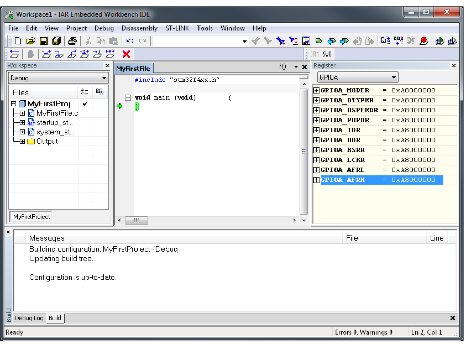


Figure : An example of IAR debug session in progress

## ChibiOS

ChibiOS is a complete, portable, open source, compact and extremely fast RTOS. ChibiOS is designed for deeply embedded real time applications where execution efficiency and compact code are important requirements. This RTOS is characterized by its high portability, compact size and, mainly, by its architecture optimized for extremely efficient context switching.

### Feature

* Efficient and portable preemptive kernel.
* Best in class context switch performance.
* Many supported architectures and platforms.
* Static architecture, everything is statically allocated at compile time.
* Dynamic extensions, dynamic objects are supported by an optional layer built on top of the static core.
* Rich set of primitives: threads, virtual timers, semaphores, mutexes, condition variables, messages, mailboxes, event flags, queues.
* Support for priority inheritance algorithm on mutexes.
* HAL component supporting a variety of abstract device drivers: Port, Serial, ADC, CAN, EXT, GPT, I2C, ICU, MAC, MMC, PWM, RTC, SDC, SPI, UART, USB, USB-CDC.
* Extensive test suite with benchmarks.

### Performance Data

Table : Test result from version 2.4.0

|  |  |  |
| --- | --- | --- |
| **Platform-Frequency-Compiler** | **Context Switch** | **Kernel Size (bytes)** |
| ARM7(ARM)/LPC2148-48-GCC4.6.2 | 2.03µS | 9224 |
| ARM7(THUMB)/LPC2148-48-GCC4.6.2 | 2.43µS | 6052 |
| ARMCM0/LPC1114-48-GCC4.6.2 | 2.60µS | 5500 |
| ARMCM0/LPC1343-72-GCC4.6.2 | 1.02µS | 6172 |
| ARMCM3/STM32F1xx-72-GCC4.6.2 | 1.03µS | 6172 |
| ARMCM3/STM32F4xx-168-GCC4.6.2 | 0.40µS | 6172 |
| MSP430F1611-8-GCC3.2.3 | 16.93µS | 6108 |
| AVR/ATMega128-16-GCC4.3.0 | 11.24µS | N/A ~7500 |
| STM8L152-16-Cosmic | N/A | N/A ~6500 |
| STM8L152-16-Raisonance | 9.03µS | N/A ~7000 |
| STM8S105-16-Cosmic | N/A | N/A ~6500 |
| STM8S105-16-Raisonance | 9.03µS | N/A ~7000 |
| PA/SPC563M64-80-GCC4.4.1 | 1.11µS | 11944 |

### Supported Ports

Table : Offically supported ports

|  |  |  |
| --- | --- | --- |
| **Core Architecture** | **Compiler** | **Supported Platforms** |
| ARM Cortex-M0 (ARMv6-M) | GCC | LPC11xx, LPC11Uxx, STM32F0xx |
| ARM Cortex-M0 (ARMv6-M) | RVCT | LPC11xx, LPC11Uxx, STM32F0xx |
| ARM Cortex-M3 (ARMv7-M) | GCC | LPC13xx, STM32F1xx, STM32F2xx, STM32L1xx |
| ARM Cortex-M3 (ARMv7-M) | IAR | LPC13xx, STM32F1xx, STM32F2xx, STM32L1xx |
| ARM Cortex-M3 (ARMv7-M) | RVCT | LPC13xx, STM32F1xx, STM32F2xx, STM32L1xx |
| ARM Cortex-M4 (ARMv7-ME) | GCC | STM32F3xx, STM32F4xx |
| ARM Cortex-M4 (ARMv7-ME) | IAR | STM32F3xx, STM32F4xx |
| ARM Cortex-M4 (ARMv7-ME) | RVCT | STM32F3xx, STM32F4xx |
| ARM7 | GCC | AT91SAM7x, LPC214x |
| MegaAVR | GCC | ATmega128, AT90CAN128, ATmega328p, ATmega1280 |
| MSP430 | GCC | MSP430F1611 |
| Power Architecture e200z | GCC/HighTec | SPC56x (all) |
| STM8 | Cosmic | STM8L, STM8S |
| STM8 | Raisonance | STM8L, STM8S |

### ChibiOS General Architecture

ChibiOS is a very modular design, it is internally divided in several major independent components. The components themselves are divided in multiple subsystems.

**Kernel**

This is the platform independent part of the OS kernel. Note that the kernel together with its port layer is a totally stand alone system and can work even without the HAL or any other component.

**Port Layer**

This is the architecture/compiler dependent part of the OS kernel. This component is responsible of the system startup, interrupts abstraction, lock/unlock primitives, context switch related structures and code.

This component usually contains very little code because most of the OS is very portable but the quality of the implementation of the Port component can affect heavily the performance of the ported OS. This is probably the most critical part of the whole OS.

**Hardware Abstraction Layer (HAL)**

This component contains a set of abstract device drivers that offer a common I/O API to the application across all the support platforms. The HAL code is totally portable across the various architectures and compilers. Drivers are classified in several classes:

**Platform Layer**

This layer contains a set of device drivers implementations. Device driver implementations are usually dedicated to a whole products family rather than a specific MCU model.

**Board Initialization**

Board files are used by the system startup in order to initialize the target board, the HAL and the Kernel before launching the application.

**Various**

This is a library of various extra utilities that do not belong to any particular component but can make life easier while developing an embedded application.

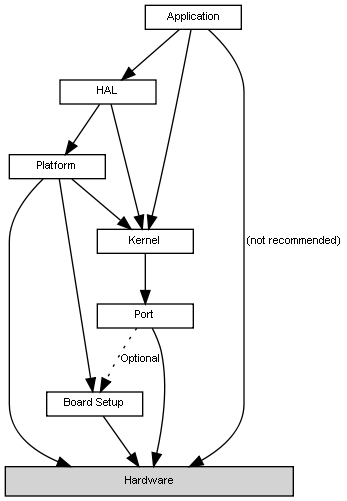


Figure : ChibiOS general architectures

### Thread Management in ChibiOS

#### Thread States

There are two classes of threads in ChibiOS/RT:

* **Static Threads**: This class of thread is statically allocated in memory at compile time.
* **Dynamic Threads**: Threads created by allocating memory at run time from a memory heap or a memory pool.

Usually, **Static Threads** are used because they are ideal for safety applications because there is no risk of a memory allocation failure because progressive heap fragmentation.

A couple of functions to deal with threads are shown in the following Figure:

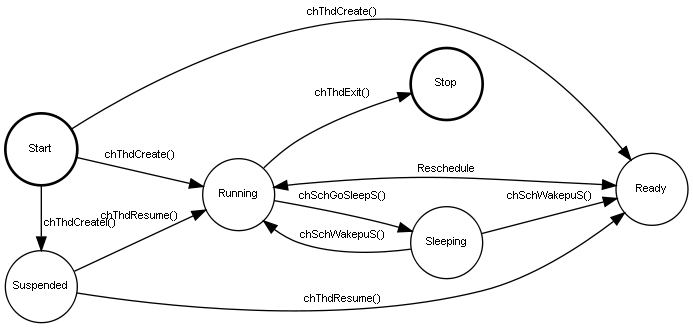


Figure : Thread state changes

#### Scheduling

ChibiOS utilizes two type of scheduling mechanism:

* Preemptive priority: this is the default mechanism used in ChibiOS. Highest priority thread is executed first.
* Round robin: multiple threads with the same priority are executed in a specific time slices before stepping in the background and let the others replace them in foreground.

The strategy is very simple the currently ready thread with the highest priority is executed. If more than one thread with equal priority are eligible for execution then they are executed in a round-robin way, the CPU time slice constant is configurable. The ready list is a double linked list of threads ordered by priority.

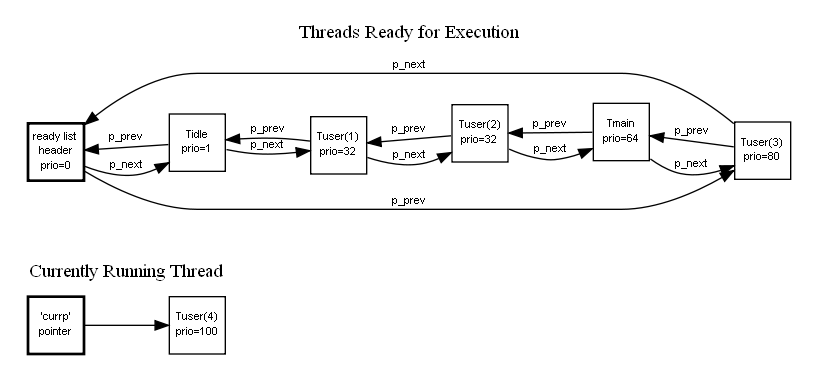


Figure : ChibiOS thread list

The currently running thread is not in the ready list, the list only contains the threads ready to be executed but still actually waiting.

Priorities in ChibiOS are a contiguous numerical range but the initial and final values are not enforced:

* *IDLEPRIO*: this is the lowest priority level and is reserved for the idle thread, no other threads should share this priority level. This is the lowest numerical value of the priorities space.
* *LOWPRIO*: the lowest priority level that can be assigned to an user thread.
* *NORMALPRIO*: this is the central priority level for user threads. It is advisable to assign priorities to threads as values relative to NORMALPRIO, as example NORMALPRIO-1 or NORMALPRIO+4, this ensures the portability of code should the numerical range change in future implementations.
* *HIGHPRIO*: the highest priority level that can be assigned to an user thread.
* *ABSPRO*: absolute maximum software priority level, it can be higher than HIGHPRIO but the numerical values above *HIGHPRIO* up to *ABSPRIO* (inclusive) are reserved. This is the highest numerical value of the priorities space.

## REQUIRED TOOLS

### Tools

Development Enviroment: Mac OS X 10.10

Required tools:

* **XCode Command line Tools**: just install the Apple free XCode command line tools or the full XCode IDE from the Mac App Store. This is essential for compiling and building code.
* **ST-LINK Utility:** It is used for programming and debugging different micro-controllers.
* **Crosscompiler for Mac**: a GNU Tools for ARM Embedded Processor used for compiling and debugging code for ARM Embedded Processor.

### Setup

**XCode Command Line Tools**: With XCode command line tools: you can go to <http://railsapps.github.io/xcode-command-line-tools.html> to get a full instruction over how to install XCode command line tools on Mac OS X 10.10, if you can pull off a *gcc* command, then the tools are ready to go.



Figure : GCC Command

**STLINK-Utility**: clone the repo from this link <https://github.com/texane/stlink>. Next, go to *stlink* folder and execute these commands:

$ ./autogen.sh

$ ./configure

$ make

A couple of binaries are generated where the most interesting for use is *st-util* – it is used to connect STM32F4 board with debugging program.



Figure 9: Running *st-util* utility

**GCC ARM Crosscompiler**: download binaries file from this link:

<https://launchpad.net/gcc-arm-embedded/4.8/4.8-2014-q1-update/+download/gcc-arm-none-eabi-4_8-2014q1-20140314-mac.tar.bz2>

Extract everything then export the *arm-gcc-none-eabi* utility to $PATH so it becomes an environment variable. This will make the *makefile* easier to write.

Enter the following command in Terminal App to open *.bash\_profile*:

touch ~/.bash\_profile; open ~/.bash\_profile

Add the following line to the end of the file adding whatever additional directory you want in your path:

export PATH="$HOME/arm-none-eabi/bin:$PATH"

Save the *.bash\_profile* and force it to execute with the command:

source ~/.bash\_profile

## INSTALL CHIBIOS ON STM32F4

### ChibiOS Source Code

ChibiOS already supports source code for use with STM32 board in general. We can find the source code for the STM32F407 Discovery at the folder ChibiOS/demos/STM32/RT-STM32F407-DISCOVERY. It also contains a *makefile* for building convention. There is not much for the demo program to be built. Further application can be developed using this demo source code as based.

Just clone the source code from <https://github.com/ChibiOS/ChibiOS> and we are good to go.

### Load firmware onto STM32F4

To load firmware onto STM32F4 board, first we have to connect it to the computer using a USB type A and type B mini. After that, we turn on *st-util* utility tool to connect the board with the debugging and loading tool as described above in the Setup section.

Next, go to folder ***ChibiOS/demos/STM32/RT-STM32F407-DISCOVERY***. This folder contains demo code for building an application using ChibiOS and run on STM32F4 board. Just use the *make* command to build all the source code. It then generates a sub folder ***build*** containing an *.elf* file, which is used to load onto STM32F4 board.

To load this file onto the board, we open the Terminal app, and use *arm-gdb-none-eabi* command to debug the *.elf* file.

$ arm-none-eabi-gdb build/ch.elf

This brings on the popular *gdb* command line interface. We then use these following commands to load the binary file onto the board and run the firmware:

(gdb) tar ext :4242 This is to tell the debugger to connect to the target at port 4242 (default connecting port for STM32F4 board)

(gdb) monitor reset halt  command resets and halts the target to make it ready to receive the image.

(gdb) load h loads the image file onto the board.

(gdb) continue The board can now excute the program which has been loaded on.

### Debugging

Similar to how we load the firmware onto the board but at loading step, we add the target destination as an argument so that we can have important debugging symbols afterward.

(gdb) load build/ch.elf Load the ch.elf file onto the board.

We can then using usual *gdb* command to debug our program.



Figure 10: Using GDB to debug program

## BUILDING A DEMO PROGRAM RUNNING ON STM32F4

### LED blinking demo program

LEDs are one the most important and basic thing in a microcontroller board. Most of the demo programs always start with LED blinking.

In this demo program, we will build a LED blinking application using two threads. Each thread will blink a couple of LEDs. First, a thread will be created to blink orange and red LEDs. When you press the user button on the board, the other thread is created, and the other two green and blue LEDs will blink too.

### Program Implementation

**Creating working area**

In order to create a static thread a working area must be declared using the macro WORKING\_AREA as shown:

static WORKING\_AREA(waThread1, 128);

This macro reserves 128 bytes of stack for the thread and space for all the required thread related structures. The total size and the alignment problems are handled inside the macro, you only need to specify the pure and simple desired stack size.

**Creating function:**

Each thread will blink a couple of LEDs as shown below:

while (true)

{

palSetPad(GPIOD, GPIOD\_LED3); /\* Orange. \*/

palSetPad(GPIOD, GPIOD\_LED4); /\* Yellow. \*/

chThdSleepMilliseconds(500);

palClearPad(GPIOD, GPIOD\_LED3); /\* Orange. \*/

palClearPad(GPIOD, GPIOD\_LED4); /\* Yellow. \*/

chThdSleepMilliseconds(500);

}

**Starting a thread:**

A static thread can be started by invoking chThdCreateStatic() as shown in this example:

chThdCreateStatic(waThread1, sizeof(waThread1), NORMALPRIO, Thread1, NULL);

**Calling halInit() function:**

This function invokes the low level initialization code then initializes all the drivers enabled in the HAL. Finally the board-specific initialization is performed by invoking boardInit().

With this function, we don’t need to initialize the GPIO by ourselves.

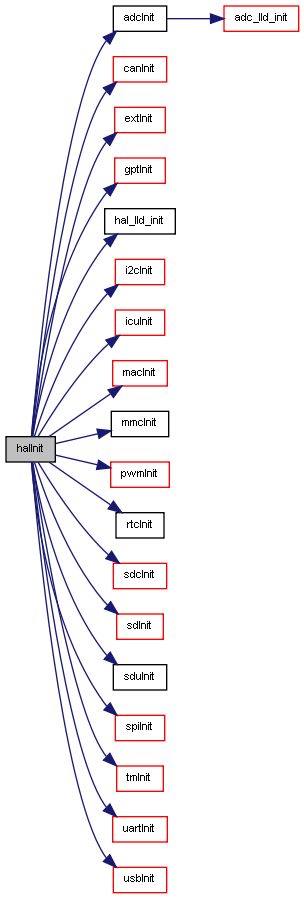


Figure : halInit() function

**Calling chSysInit() function:**

This function initializes the RTOS. After executing this function the current instructions stream becomes the main thread.

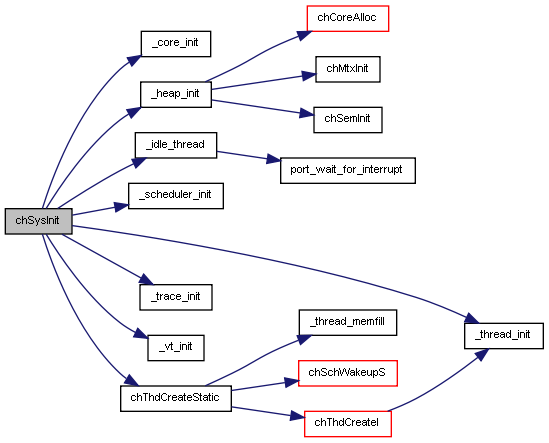


Figure : chSysInit() function

# CONCLUSION

STM32F4 Discovery is a very powerful kit yet so small and easy to use. It holds a great potential when combined with additional hardware and real time OS. With this, we can build a lot of multi-processing real time program for the purpose of researching and application alike.

This group has succeeded in researching about STM32F4 Discovery Kit, as well as ChibiOS. We also installed successfully ChibiOS on STM32F4 Discovery Kit and write a simple LED blinking demo program.

**REFERENCE**

* Dr Ngo Lam Trung’s Real time System Slides.
* STLink STM32F4Discovery board User manual:
* <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00039084.pdf>ChibiOS Documentation and Guides: <http://www.chibios.org/dokuwiki/doku.php?id=chibios:documents>
* GRAFIXMAFIA’s guide on using STM32F4 on Mac OS X: <http://grafixmafia.net/updated-using-the-stm32f4-discovery-board-with-mac-osx-10-9-mavericks/>