**Research Report**

Real Time System

**Installing and using ChibiOS on STM32F4 Discovery Kit**

Author: Ta Duy Hung – 20101682

Bui Tri Trung – 20102365

Lecturer: Dr. Ngo Lam Trung

TABLE OF CONTENTS

1 INTRODUCTION 3

1.1 PREFACE 3

1.2 TASK LIST 3

2 CONTENTS 3

2.1 About RTOS 3

2.1.1 RTOS Definition 3

2.1.2 Các thành phần trong RTOS: 5

2.2 STM32F4 Discovery 13

2.2.1 Overview of STM32F4 Discovery board 13

2.2.2 Quản lý tiến trình trong CooCox 15

2.2.3 Cơ chế lập lịch của CooCox 22

2.2.4 Quản lý thời gian 24

2.3 REQUIRED TOOLS 25

2.3.1 Tools 25

2.3.2 Setup 25

2.4 INSTALL CHIBIOS ON STM32F4 27

2.4.1 Source Code 27

2.4.2 Load firmware onto STM32F4 27

2.4.3 Debugging 28

2.5 BUILDING A DEMO PROGRAM RUNNING ON STM32F4 28

2.5.1 Mô tả chương trình: Phát nhạc trên STM32F4 28

2.5.2 Cấu hình hệ thống để chạy Demo 29

3 CONCLUSION 33

# 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:

* Introduction about STM32F4 and ChibiOS
* Scheduling in ChibiOS
* Install ChibiOS on STM32F4
* Building an LED Blinking demo application using ChibiOS running on STM32F4

# CONTENTS

## About RTOS

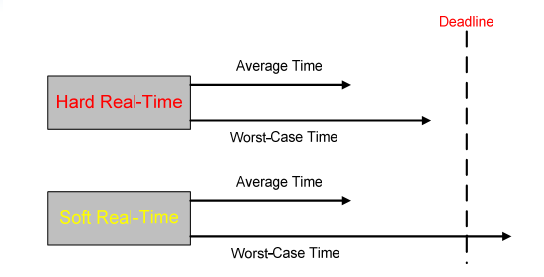
### RTOS Definition

A **real-time operating system** (**RTOS**) is an operating system (OS) intended to serve real-time application process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.

#### Real time System:

* Thời gian (Time): Sự chính xác của hệ thống không chỉ phụ thuộc vào kết quả tính toán logic mà còn phụ thuộc vào thời gian cho ra kết quả.
* Thực (Real): Đáp ứng của hệ thống với những sự kiện bên ngoài.
* Thời gian thực (Real-Time): Phải đảm bảo các yếu tố:
  + Đáp ứng nhanh.
  + Dự đoán được.
* Các tác vụ (Real-time Task) được xác định bằng deadline. Deadline là thời gian tối đa một tác vụ PHẢI hoàn thành việc thính toán.

#### Thời gian thực cứng (Hard Real-time) và thời gian thực mềm (Soft Real-Time)



Hình 1.1: Thời gian thực cứng và thời gian thực mềm.

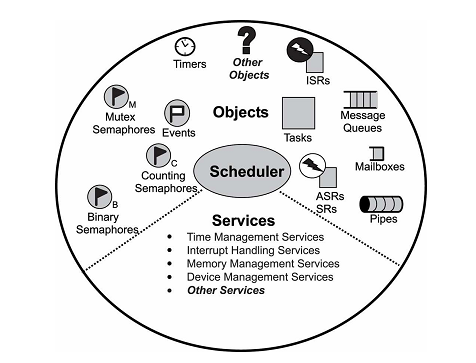
* Thời gian thực cứng: Một tác vụ là thời gian thực cứng nếu như thời gian tính toán vượt quá deadtime có thể gây ra sự phá vỡ môi trường điều khiển.
* Thời gian thực mềm: Một tác vụ là thời gian thực mềm nếu như đảm bảo thực thi trong deadtime cho phép và nếu như không đảm bảo thì sẽ không tạo ra những nguy hại nào đáng kể cho hệ thống và không làm ảnh hưởng đến sự ứng xử của hệ thống.

#### Định nghĩa hệ điều hành thời gian thực RTOS

Là hệ thống có:

* Lịch trình thực thi theo thời gian.
* Quản lý tài nguyên hệ thống.
* Cung cấp những nền tảng cơ bản để phát triển các ứng dụng trên nó.

### Các thành phần trong RTOS:



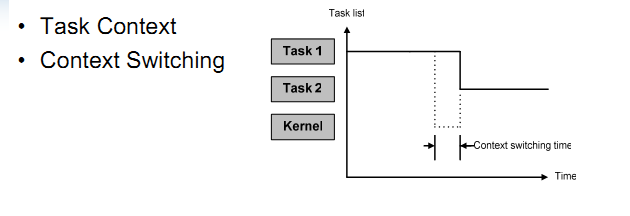
Hình 1.2 : Các đối tượng trong RTOS

* Bộ lịch trình (scheduler): Là một tập các thuật toán để xác định tác vụ (Task) nào được thực thi.
* Đối tượng (Object): Là những cấu trúc đặc biệt (Kernel) giúp người lập trình tạo ra các ứng dụng.
* Dịch vụ (Service): Là những điều khiển mà lõi (Kernel) thực thi trong đối tượng (object): Chia thời gian (Timing), Ngắt (Interrupt), Đáp ứng (Handling) và quản lý tài nguyên hệ thống (Resource management)

#### Bộ lịch trình (Scheduler):

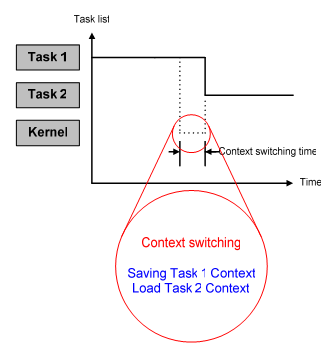
* Là một phần vô cùng quan trọng của lõi hệ thống điều khiển (Operating System Kernel).
* Nó giúp xác định tác vụ (Task) nào sẽ dành quyền CPU để thực thi tiếp theo. Sau đó, nó thực hiện việc chuyển trạng thái (Context Switching) được thực hiện bằng bộ điều phối (Dispatcher).

##### Chuyển đổi trạng thái (Context Switching)



Hình 1.3 : Chuyển đổi trạng thái (ngữ cảnh)

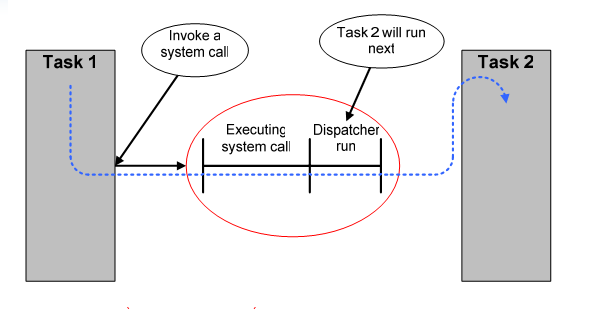
* Trạng thái (ngữ cảnh) của tác vụ (Task)
  + Mỗi task có một trạng thái riêng của nó, nó chính là trạng thái của những thanh ghi (registers).
  + Mỗi thời điểm 1 task mới được tạo ra, kernel sẽ tạo ra và lưu giữ một block điều khiển liên quan đến task đó (TCBs ). TCBs là những cấu trúc dữ liệu hệ thống mà kernel dùng để lưu trữ những thông tin đặc trung của task. TCBs chứa mọi thứ mà một kernel cần để biết về một task cụ thể nào đó. Khi task đnag chạy, trạng thái của nó là động. Trạng thái động này được lưu trữ trong TCB. Khi task không còn chạy nữa, trạng thái sẽ bị đóng bang (frozen) trong TCB, và được sử dụng cho lần thực thi tiếp theo của task.
* Công tắc chuyển đổi trạng thái (Context Switch):
  + Sẽ xãy ra khi bộ lịch trình (scheduler) chuyển từ một trạng thái này sang một trạng thái khác.
* Việc chuyển đổi trạng thái bao gồm:
  + Thời gian chuyển đổi: Là thời gian tiêu tốn để cho bộ lịch trình chuyển từ task này sang task khác. Nó không có liên quan nào đến các lệnh thực hiện trong task.
  + Nếu một ứng dụng được thiết kế mà xảy ra chuyển đổi trạng thái thường xuyên thì có thể dẫn đến những thực thi không cần thiết. Vì vậy, nên thiết kế ứng dụng theo cách mà tạo ra ít chuyển đổi trạng thái nhất.
  + Khi nào chuyển đổi xảy ra: Mỗi khi ứng dụng tạo một lời gọi hệ thống (System Call), bộ lịch trình sẽ xác định rằng có cần chuyển đổi trạng tháu hay không. Khi bộ lịch trình xác định việc chuyển đổi là cần thiết thì sẽ gọi bộ phân phối (dispatcher) để thực hiện việc chuyển đổi.
  + Ví dụ: Khi bộ thực thi của Kernel xác định cần dừng việc thực thi task 1 để chuyển qua trạng task 2 thì nó sẽ thực thi theo các bước sau: Kernel sẽ lưu thông tin trạng thái của Task 1 🡺 Load thông tin trạng thái của Task 2, task2 trở thành luổng (thread) thực thi hiện tại. Trạng thái chuyển đổi của Task 1 sẽ được đóng bang trong khi Task 2 đang được thực thi, nhưng nếu như bộ chuyển đổi xác định cần chạy lại task 1 thì task 1 sẽ quay lại vị trí ngay trước khi nó bị chuyển đổi.



Hình 1.4: Ví dụ về Contex Switch

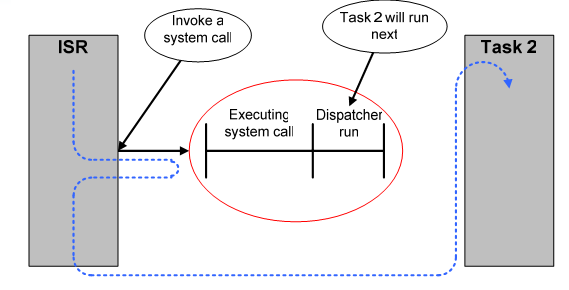
##### Bộ điều phối (Dispatcher):

* Dòng thưc thi (Flow of Execution): Tại bất kì thời điểm nào RTOS đang chạy, dòng thực thi sẽ chuyển đến 3 vùng: đến các Task ứng dụng (application Task), đến một chương trình phục vụ ngắt (ISR), hoặc đến Kernel.
* Khi nào bộ phân phối được thực hiên:
  + Khi một Task hoặc ISR tạo một lời gọi hệ thống, dòng điều khiển sẽ chuyển đến Kernel để thực thi một trong số những thủ tục được cung cấp bởi Kernel.
  + Khi rời khỏi kernel, Dispatcher sẽ có trách nhiệm là chuyển lệnh điều khiển đến một trong số những Task ứng dụng.
* Cách thực hiện như sau:
  + Không cần thiết để chuyển toàn bộ lệnh điều khiển đến cùng một task gọi System Call. Điều này sẽ được xác định bằng giải thuật phân chia thời gian biểu (scheduling algorithms ) của bộ lịch trình (scheduler).
  + Chuyển đổi thực thi từ một Task



Hình 1.5: Điều phối từ một Task

* Tùy thuộc vào cách truy cập vào Kernel như thế nào mà dispatching sẽ xãy ra khác nhau. Khi một Task thực hiện một lời gọi hệ thống, dispatcher sẽ được sử dụng để thoát khỏi Kernel mỗi khi Lời gọi hệ thống được hoàn thành. Trong trường hợp này, dispatcher được dùng như một lời gọi của lời gọi (call- by –call basic) để nó có thể hỗ trợ cho việc chuyển đổi trạng thái của Task. Bất kì Task nào cũng có thể gọi system call. Một hay nhiều Task có thể ở trạng thái sắn sàng cho thực thi.
* Chuyển đổi thực thi từ 1 chương trình phục vụ ngắt(ISR):

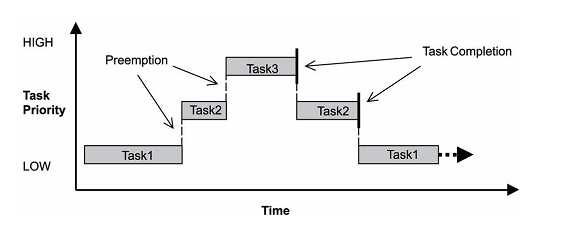


Hình 1.6: Điều phối từ một ISR

* Khi một ISR tạo một System Call, Dispatcher sẽ bị bỏ vô hiệu hóa cho đến khi ISR thưc thi xong. Quá trinh này sẽ đúng miễn là có đủ tài nguyên để chuyển đổi giữa các Task. Chuyển đổi trạng thái này sẽ không được diễn ra bởi vì ISR phải được thực thi xong mà không được ngắt bởi các Tasks. Sau khi ISR thực thi xong, Kernel sẽ thoát đến dispatcher để có thể điều phối đến đúng task thực thi tiếp theo.

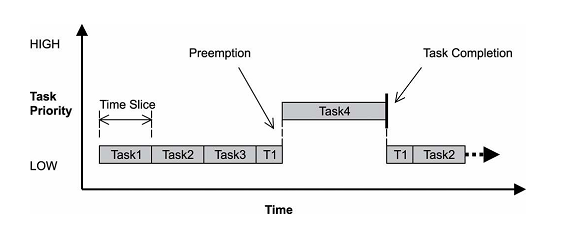
##### Giải thuật cho lịch trình:

* Lịch trình thay thế theo độ ưu tiên:



Hình 7: Giải thuật lịch trình theo độ ưu tiên.

* + Hầu hết các Real – time Kernel sử dụng giải thuật lịch trình thay thế theo độ ưu tiên (preemptive priority - based scheduling) làm mặc định.
  + Các task sẽ được thực thi tại bất kì một thời điểm là task có độ ưu tiên cao nhất so với các task khác đang ở trạng thái sẵn sàng.
  + Real Time Kernel hỗ trợ 256 cấp độ ưu tiên, với 0 là độ ưu tiên cao nhất và 255 là độ ưu tiên thấp nhất. Một số Kernel thì ngược lại với 255 là độ ưu tiên cao nhất và 0 là độ ưu tiên thâp nhất.
  + Với bộ chuyển đổi theo độ ưu tiên, mỗi task phải có một độ ưu tiên, và task có độ ưu tiên cao nhất chạy đầu tiên. Nếu một Task có độ ưu tiên cao hơn task đang chạy trở nên sẵn sàng để chạy thì kernel sẽ ngay lập tức lưu lại trạng thái Task hiện tại và chuyển sang Task có độ ưu tiên cao hơn.
  + Mặc dù việc phân chia độ ưu tiên của Task được thực hiện khi task đó được tạo nhưng độ ưu tiên của Task là có thể thay đổi một cách linh động sử dụng Lời gọi do kernel cung cấp (Kernel – provided calls). Khả năng này dùng để thay đổi một cách linh động cho phép một ứng dụng nhúng có độ linh hoạt để ứng xử với sự kiện bên ngoài khi chúng xuất hiện, tạo ra một hệ thống thời gian thực và đáp ứng tốt. Lưu ý là việc sử dụng không đúng khả năng thay đổi độ ưu tiên này có thể dẫn đến đảo độ ưu tiên (priority inversion), vùng chết (deadlock), và có thể dẫn đến treo hệ thống (system failure).
  + Ví dụ: (Hình 7) Task 1 được thay thế bởi task 2 có độ ưu tiên cao hơn, task 2 được thay thế bởi task 3 có độ ưu tiên cao hơn, khi Task 3 hoàn thành, task 2 sẽ tiếp tục thực thi trạng thái ngay trước khi bị dành quyền thực thi. Tương tự, khi task 2 hoàn thanh thì Task 1 sẽ được tiếp tục thực thi (resumes).
* Lịch trình theo Round Robin (Gọi vòng)



Hình 8 : Lịch trình theo Round Robin

* Mỗi Task sẽ cùng chia sẻ thời gian thực thi của CPU. Round Robin thuần túy không thỏa mãn yêu cầu của hệ thống thời gian thực bởi vì một hệ thống thời gian thực các task phải làm việc theo mức độ quan trong khác nhau.
* Ví dụ: (Hình 8)
  + Thay vì thay thế theo độ ưu tiên, các task round –robin được phân chia thời gian thực thi theo các khoảng thời gian (time slice).
  + Với time slicing, mỗi task sẽ được thực thi trong một khoảng thời gian nhất định, và theo vong trong. Một bộ đếm thời gian sẽ giám sát thời gian của mỗi Task, tăng lên theo mỗi xung clock. Khi thời gian thực thi một task đã hết, bộ đếm sẽ bị xóa, và task này sẽ được đặt ở cuổi cùng của chu kì (end of circle).
  + Nếu như một task round-robin bị thay thế bởi mọt task có độ ưu tiên cao hơn, thì bộ đếm thời gian sẽ lưu lại và phục hồi khi task bị thay thế dành quyền thực thi lại.

#### Các đối tượng (Objects) trong RTOS:

* *Tasks:* Là các luồng (thread) thực thi cùng tồn tại và độc lập nhau có thể “ cạnh tranh” nhau để dành quyền thực thi.
* *Semaphores:* Là đối tượng bắt sự kiện để đồng bộ giữa các tasks, có thể tăng hoặc giảm.
* *Message Queues:* Là một kiểu cấu trúc dữ liệu được dùng để đồng bộ hóa hoặc trao đổi thông tin giữa các Tasks.
* *Real-time embedded applications:* Là sự kết nối giữa các đối tượng của Kernel để giải quyết vấn đề thời gian thực như sự đồng thời, sự đồng bộ,và trao đổi dữ liệu

## STM32F4 Discovery

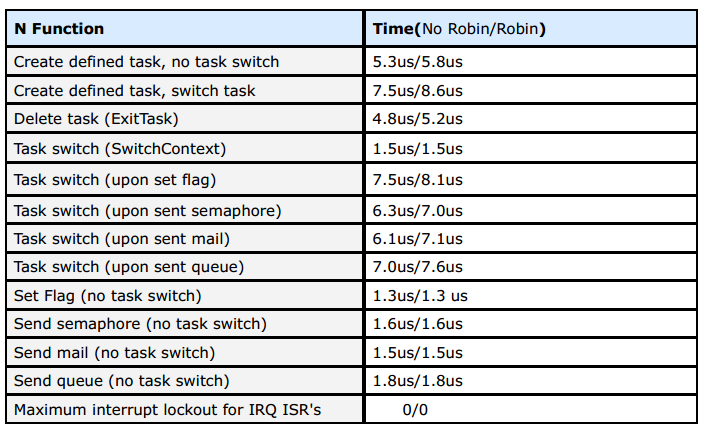
### Overview of STM32F4 Discovery board

CooCox CoOS là một hệ điều hành nhúng thời gian thực, đa tác vụ. CooCox CoOS thiết kế cho kiến trúc xử lý ARM Cortex M series.

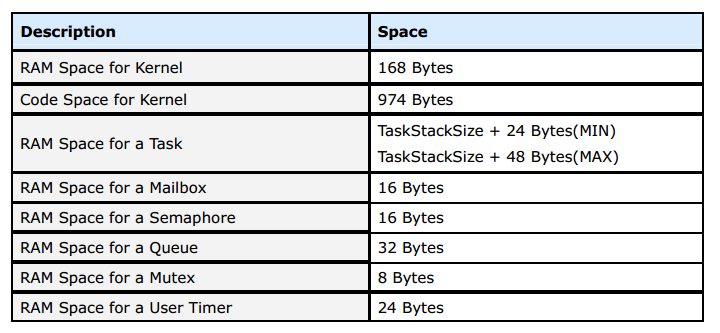
#### Tính năng của CooCox CoOS

* Thiết kế đặc biệt dành cho dòng vi xử lý Cortex-M.
* Là hệ điều hành thời gian thực mã nguồn mở miễn phí.
* Có tính mở rộng.
* Kernel của hệ thống nhỏ, kích thước chỉ 974Byte.
* Có khả năng thay đổi thuật toán lập lịch trong hệ thống.
* Hỗ trợ ngắt ưu tiên và round-robin.
* Độ trễ ngắt = 0.
* Hỗ trợ Semaphore, Mutex, Flag, Mailbox và Queue cho việc giao tiếp và đồng bộ.
* Có khả năng phát hiện tràn bộ nhớ trong stack.
* Hỗ trợ các nền tảng ICCARM, ARMCC, GCC.

#### Các thông số kỹ thuật

Bảng đặc tả thời gian

Bảng đặc tả không gian bộ nhớ



#### Các thiết bị được hỗ trợ

* ST STM32 Series
* Atmel ATSAM3U Series
* NXP LPC17xx LPC13xx LPC11xx Series
* Toshiba TMPM330 Series
* Luminary LM3S Series

### Quản lý tiến trình trong CooCox

#### Task

Một ứng dụng thường được chia nhỏ thành các task. Trong ngôn ngữ C task thường chứa các vong lặp.Task có thể có parameter và trả về các giá trị. Khi task không trả về giá trị thì phải đặt kiểu trả về là void.

Điểm khác biệt trong ngôn ngữ C là để thoát task thì phải gọi hàm API hệ thống. Nếu thoát task chỉ bằng cách kết thúc lệnh cuối cùng của task thì hệ thống sẽ bị breakdown (lỗi).

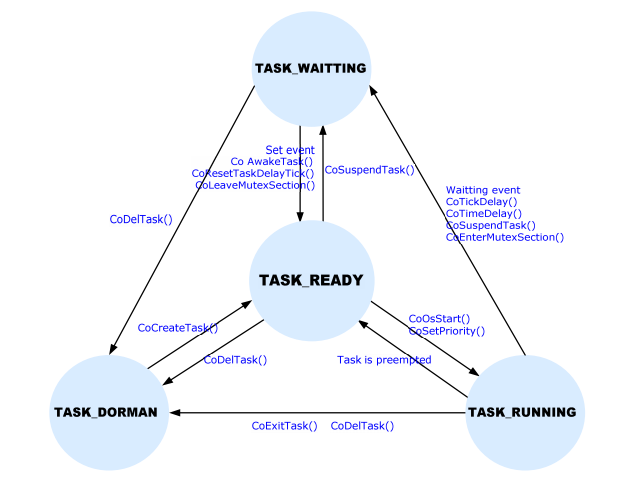
Có thể thoát task bằng cách gọi một trong 2 câu lênh sau: CoExitTask() hoặc CoDelTask(TaskID). CoExitTask() dùng để thoát task hiện tại đang chạy còn CoDelTask(taskID) dùng để thoát task khác có id là taskID. Nếu có taskID của task hiện tại thì cũng có thể sử dụng lênh CoDelTask(taskID) để thoát task đang chạy.

#### Trạng thái task

Có 4 trạng thái của task:

* ReadyState(TASK\_READY) ready tasks là những tiến trình sẵn sàng chạy nhưng không phải đang chạy bởi vì các task cùng độ ưu tiên hoặc có độ ưu tiên cao hơn đang chạy có trạng thái RUNNING. Các Task này sẽ có trạng thái tiếp theo là RUNNING khi khởi chạy.
* RunningState(TASK\_RUNNING) running task là tiến trình đang thực sự được chạy. Nó đang chiếm giữ tài nguyên xử lý.
* Waiting State(TASK\_WAITING) waiting task là những task đang chờ 1 sự kiện nào đó. Nó sẽ ở trạng thái chờ khi đang chờ 1 sự kiện nào đó xảy ra.
* Dormant State(TASK\_DORMANT) Dormant task là những task đã thoát và không thể lập lịch được. Nó không giống như trạng thái wating .Trạng thái wating có thể được reactivate và có thể được lập lịch khi 1 sự kiện nào đó xảy ra. Trạng thái Dormant không bao giờ reactivate.

Các trạng thái có thể được chuyển đổi cho nhau. Có thể ghi lệnh CoSuspendTask() để chuyển từ trạng thái running, ready sang waiting. Có thể gọi lệnh CoAwakeTask() để chuyển trạng thái từ waiting sang ready.



#### Task Control Block

Task control block là 1 cấu trúc dữ liệu để lưu trạng thái của 1 task trong CooCox CoOS. Mỗi khi 1 task được khởi tạo, CooCox CoOS sẽ tạo 1 task control block để mô tả trạng thái của task đó. Điều này sẽ khiến cho task có thể chạy chính xác khi nó lấy tài nguyên CPU.

Task control block luôn được kết hợp với task như 1 mô tả trạng thái của nó. Nó sẽ không bị thu hồi cho đến khi task bị thoát hoặc bị xóa khỏi hệ thống.

Cấu trúc TCB của task được mô tả như sau:

typedef struct TCB {

OS\_STK \*stkPtr; /\*!< The current point of task. \*/

U8 prio; /\*!< Task priority. \*/

U8 state; /\*!< TaSk status. \*/

OS\_TID taskID; /\*!< Task ID. \*/

#if CFG\_MUTEX\_EN > 0

OS\_MutexID mutexID; /\*!< Mutex ID. \*/

#endif

#if CFG\_EVENT\_EN > 0

OS\_EventID eventID; /\*!< Event ID. \*/

#endif

#if CFG\_ROBIN\_EN >0

U16 timeSlice; /\*!< Task time slice \*/

#endif

#if CFG\_STK\_CHECKOUT\_EN >0

OS\_STK \*stack; /\*!< The top point of task. \*/

#endif

#if CFG\_EVENT\_EN > 0

void\* pmail; /\*!< Mail to task. \*/

struct TCB \*waitNext; /\*!< Point to next TCB in the Event waitting list.\*/

struct TCB \*waitPrev; /\*!< Point to prev TCB in the Event waitting list.\*/

#endif

#if CFG\_TASK\_SCHEDULE\_EN == 0

FUNCPtr taskFuc;

OS\_STK \*taskStk;

#endif

#if CFG\_FLAG\_EN > 0

void\* pnode; /\*!< Pointer to node of event flag. \*/

#endif

#if CFG\_TASK\_WAITTING\_EN >0

U32 delayTick; /\*!< The number of ticks which delay. \*/

#endif

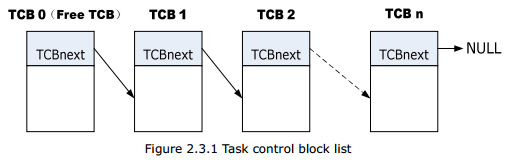
struct TCB \*TCBnext; /\*!< The pointer to next TCB. \*/

struct TCB \*TCBprev; /\*!< The pointer to prev TCB. \*/

}OSTCB,\*P\_OSTCB;

|  |  |
| --- | --- |
| stkPtr | Con trỏ trỏ đến địa chỉ đầu tiên của stack của task hiện tại. CooCox CoOS cho phép mỗi task có 1 stack riêng của mình với kích thước bất kỳ. Trong quá trình chuyển sang task khác, CoOS lưu trạng thái CPU đang chạy thông qua stack mà stkPtr trỏ tới vì vậy task có thể quay lại trạng thái CPU đã chạy trước đó khi nó được chạy lại. Từ khi Cortex-M3 có 16 thanh ghi thường 32-bit registers để mô tả trạng thái CPU, kích thước nhỏ nhất của stack dành cho mỗi task là 68 bytes(4 bytes khác dùng để kiểm tra tràn bộ nhớ stack ). |
| Prio | Độ ưu tiên của task. |
| State | Trạng thái của task |
| taskID | id của task. TaskID dùng để định danh task |
| mutexID | mutex ID mà task đang chờ |
| eventID | ID của sự kiện mà task đang chờ |
| timeSlice | time slice của task. |
| Stack | Con trỏ trỏ tới địa chỉ cuối cùng của stack. Có thể được dùng để kiểm tra tràn bộ nhớ stack. |
| Pmail | Con trỏ trỏ tới thông tin trả về cho task. |
| waitNext | TCB của task kế tiếp trong danh sách chờ sự kiện. |
| waitPrev | TCB của task trước đó trong danh sách chờ sự kiện. |
| taskFuc | Con trỏ trỏ tới hàm của task, dùng để khởi động task. |
| taskStk | Con trỏ trỏ đến địa chỉ đầu tiên của stack của task hiện tại, dùng để khởi động task. |
| Pnode | Con trỏ trỏ tới cờ báo sự kiện. |
| delayTick | Thời gian giữa previous delayed event (sự kiện tạm dừng trước) và task hiện tạikhi nó ở trạng thái delayed. |
| TCBnext | TCB kế tiếp khi task được đưa vào danh sách ready, waiting, tạm hoãn. Nếu task hiện tại đang ở trạng thái ready thì sẽ ở trong danh sách ready. Nếu task ở trạng thái waiting thì sẽ được xác định bởi mutexID và delayTick: nếu mutexID khác giá trị 0xFFFFFFFF thì task đang ở danh sách chờ tài nguyên, còn nếu delayTick khác giá trị 0xFFFFFFFF thì task đang ở danh sách tạm dừng. |
| TCBprev | TCB trước đó khi task được đưa vào danh sách ready, waiting, tạm hoãn. |

Hệ thống sẽ gán 1 TCB cho task hiện tại từ danh sách free TCB hiện tại khi tại task mới. Con trỏ FreeTCB sẽ trỏ tới danh sách free TCB trong CooCox CoOS. Nếu FreeTCB có giá trị NULL, sẽ không còn TCB để gán và việc khởi tạo task sẽ thất bại. Khi khởi tạo hệ thống, CoOS sẽ sắp xếp lại tài nguyên các TCB có thể gán được và sau đó phản ánh trạng thái hiện tại của TCB thông qua danh sách như sau:



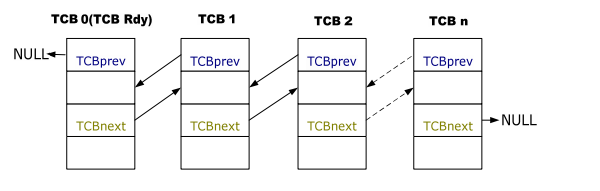
Mỗi khi tạo 1 task thành công, một free TCB sẽ được gán cho task đó và TCBnext của nó sẽ là free TCB mới cho đến khi nó có giá trị NULL. Khi 1 task được xóa hoặc thoát, Hệ thống sẽ lấy lại TCB đã được gán cho task đó khi khởi tạo task và đưa nó trở về free TCB để lần sau có thể sử dụng khi tạo task mới.

#### Danh sách các tiến trình ở trạng thái sẵn sàng

CooCox CoOS kết nối TCB của các task trong ready list dựa trên mức độ ưu tiên bằng danh sách liên kết hai chiều. Điều này đảm bảo rằng phần tử đầu tiên của danh sách luôn có độ ưu tiên cao nhất và cần được lập lịch trước tiên. CooCox CoOS cho phép nhiều task cùng chia sẻ một mức độ ưu tiên.

Vì vậy, chắc chắn sẽ có nhiều task có cùng độ ưu tiên tròn danh sách ready. CooCox CoOS sử dụng thuật toán "first-in-first out (FIFO)" đối với trường hợp này: đưa task vào muộn nhất vào sau task cuối có cùng độ ưu tiên vì thế tất cả chúng đều có thể sử dụng được tài nguyên CPU.

TCBRdy là bắt đầu của danh sách ready trong CooCox CoOS. Nói cách khác,TCBRdy là TCB của task có độ ưu tiên cao nhất trong danh sách ready. Vì vậy khi bắt đầu lập lịch task, chỉ cần kiểm tra xem liệu tiến trình mà TCBRdy trỏ đến có độ ưu tiên cao hơn tiến trình đang chạy hay không. Theo cách này hiệu năng của việc lập lịch sẽ được cải thiện tốt nhất.

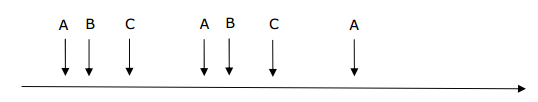


### Cơ chế lập lịch của CooCox

HĐH Coocox hỗ trợ 2 cơ chế lập lịch:

* Lập lịch dựa trên độ ưu tiên (pre-emptive priority): đây là cơ chế được sử dụng mặc định trong Coocox. Ở cơ chế này, mỗi tiến trình sẽ được đánh độ ưu tiên khi khởi tạo và tiến trình nào có độ ưu tiên cao hơn sẽ được chạy trước (độ ưu tiên nằm trong khoảng từ 0~64, giá trị nhỏ hơn sẽ chạy trước)

CoCreateTask(task, parameter, **priority**, stack,stack\_size);

* Lập lịch vòng tròn (round-robin): các tiến trình sẽ có độ ưu tiên như nhau và sẽ được chạy lần lượt 1 khoảng thời gian t được định trước sau đó quyền sử dụng CPU sẽ được chuyển sang tiến trình tiếp theo.  
    
  Để thiết lập chế độ lập lịch round-robin, ta sửa file OSConfig.h (CoOS/OSConfig.h)
  + Bật/tắt chế độ round-robin: 1-bật, 0-tắt  
    **#define** CFG\_ROBIN\_EN (1)
  + Thiết lập thời gian mỗi tiến trình chạy trong round-robin (mili-second)  
    **#define** CFG\_TIME\_SLICE (10)

HĐH CooCox sẽ chạy tiến trình lập lịch (hàm schedule()) khi gặp 1 trong 3 tình huống sau:

1. Một tiến trình có độ ưu tiên cao hơn tiến trình đang chạy chuyển sang trạng thái sẵn sàng

**Task with a higher priority is ready**

/\* Is higher PRI task coming in? \*/

if(RdyPrio < RunPrio )

{

TCBNext = pRdyTcb; /\* Yes, set TCBNext and reorder ready list\*/

pCurTcb->state = TASK\_READY;

pRdyTcb->state = TASK\_RUNNING;

InsertToTCBRdyList(pCurTcb);

RemoveFromTCBRdyList(pRdyTcb);

}

1. Tiến trình đang chạy chuyển sang trạng thái chờ hoặc trạng thái kết thúc.

**The state of the current task changes**

/\* Does Running task status change \*/

else if(pCurTcb->state != TASK\_RUNNING)

{

TCBNext = pRdyTcb; /\* Yes, set TCBNext and reorder ready list\*/

pRdyTcb->state = TASK\_RUNNING;

RemoveFromTCBRdyList(pRdyTcb);

}

1. Tiến trình đang chạy dùng hết khoảng thời gian được quy định (time slice) và trong các tiến trình ở trạng thái sẵn sàng có tiến trình có độ ưu tiên ngang bằng với tiến trình đang chạy.

**The task scheduling among the same priority tasks**

/\* Is it the time for robinning \*/

else if((RunPrio == RdyPrio) && (OSCheckTime == OSTickCnt))

{

TCBNext = pRdyTcb; /\* Yes, set TCBNext and reorder ready list\*/

pCurTcb->state = TASK\_READY;

pRdyTcb->state = TASK\_RUNNING;

InsertToTCBRdyList(pCurTcb);

RemoveFromTCBRdyList(pRdyTcb);

}

### Quản lý thời gian

Hệ điều hiều CooCox sử dụng ngắt thời gian của hệ thống (ngắt timer) để cài đặt “system tick”

Coocox sẽ thiết lập các giá trị mặc định như sau:  
/\*!< System frequency (Hz) \*/

**#define** CFG\_CPU\_FREQ (72000000)

/\*!< Systick frequency (Hz) \*/

**#define** CFG\_SYSTICK\_FREQ (100)

* CFG\_CPU\_FREQ: CPU chạy với tần số 72MHz
* CFG\_SYSTEM\_FREQ: sau 100Hz (tương đương 10 ms) thì “system tick” xảy ra 1 lần

Mỗi lần ngắt diễn ra (system tick ISR), hệ thống sẽ tăng thời gian lên 1.Ta có thể lấy thời gian hệ thống hiện tại bằng cách gọi hàm CoGetOSTime ().

Khi ngắt thời gian xảy ra , HĐH CooCox cũng sẽ kiểm tra xem danh sách trì hoãn(delay list) và danh sách đếm thời gian (timer list) – các task ở trạng thái waiting. Nếu danh sách không rỗng, giảm thời gian trễ của tiến trình đầu tiên trong danh sách sau đó đánh giá xem thời gian chờ của tiến trình đã đủ hay chưa, nếu đã đủ thì gọi chức năng tương ứng, nếu không thì chuyển sang phần xử lý tiếp theo.

## 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 (gcc-arm-none-eabi)**: 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 2: 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

### Source Code

### 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 3: Using GDB to debug program

## BUILDING A DEMO PROGRAM RUNNING ON STM32F4

### Mô tả chương trình: Phát nhạc trên STM32F4

Board STM32F4 được tích hợp sẵn chip xử lý âm thanh CS43L22. Đây là chip DAC tích hợp ampli class D, công suất ~1W với tải 8Ω, nguồn nuôi 5v ở chế độ stereo.

Chip CS43L22, để hiểu đơn giản, thì nó có 2 đường: 1 đường dữ liệu âm thanh digital , 1 đường điều khiển. Hai đường này được điều khiển qua 2 giao thức, đường âm thanh digital qua I2S, đường điều khiển qua I2C. Ứng với board STM32F4 thì đường I2S kết nối với chân SPI3, còn đường I2C kết nối với chân I2C1. Ngoài 2 chân này, ta cần phải cấu hình một vài chân GPIO khác (clock, reset, enable…).

### Cấu hình hệ thống để chạy Demo

\* Cấu hình Clock:

Trước hết phải bật clock lên. Để làm việc này thì cần bật clock cho các chân GPIO sau: GPIOA (I2S\_WS signal), GPIOB (I2C\_SDA & I2S\_SCL), GPIOC (I2S\_MCK, I2S\_SCK, I2S\_SD) và GPIOD (Reset CS43L22). Việc cấu hình và bật clock thông qua thanh ghi Reset and Clock Control” (RCC) (nằm trong thư viện **stm32f4xx\_rcc.h**). Sử dụng lệnh sau để làm các việc trên:

***RCC\_AHB1PeriphClockCmd****(RCC\_AHB1Periph\_GPIOA | RCC\_AHB1Periph\_GPIOB | RCC\_AHB1Periph\_GPIOC | RCC\_AHB1Periph\_GPIOD, ENABLE);*

Tiếp theo, ta cần bật clock cho chân SPI3 và I2C1 (chân data và chân điều khiển chip CS43L22).

***RCC\_AHB1PeriphClockCmd****(RCC\_AHB1Periph\_GPIOA | RCC\_AHB1Periph\_GPIOB | RCC\_AHB1Periph\_GPIOC | RCC\_AHB1Periph\_GPIOD, ENABLE);*

I2S có module PLL dùng để tạo tần số lấy mẫu chuẩn. Để bật, dùng lệnh sau:

***RCC\_PLLI2SCmd****(ENABLE);*

Cấu hình GPIO:

Cần set chế độ làm việc của chân (in, out), cấu hình đầu ra (push-pull hay là open-drain, tốc độ…). Thư viện “stm32f4xx\_gpio.h” cung cấp 1 struct cho phép làm những việc này **GPIO\_InitTypeDef**. Ví dụ với tín hiệu reset thì struct sẽ như sau:

*GPIO\_InitTypeDef PinInitStruct;*

*PinInitStruct.GPIO\_Pin = GPIO\_Pin\_4;*

*PinInitStruct.GPIO\_Mode = GPIO\_Mode\_OUT;*

*PinInitStruct.GPIO\_OType = GPIO\_OType\_PP;*

*PinInitStruct.GPIO\_PuPd = GPIO\_PuPd\_DOWN;*

*PinInitStruct.GPIO\_Speed = GPIO\_Speed\_50MHz;*

***GPIO\_Init****(GPIOD, &PinInitStruct);*

Chế độ làm việc của I2S và I2C sẽ là “Alternate function” (GPIO\_Mode\_AF). Đầu ra của chân I2C sẽ là “open drain” (GPIO\_OType\_OD) và “no pull” (GPIO\_PuPd\_NOPULL). Đầu ra của chân I2S sẽ là “push-pull” (GPIO\_OType\_PP). Do hầu hết cá chân đều được ghép giữa nhiều “alternate functions”, nên ta cần phải gán nó cho 1 cái thích hợp. Việc này sẽ được thực hiện nhờ câu lệnh sau:

***GPIO\_PinAFConfig****(GPIOA, GPIO\_PinSource4, GPIO\_AF\_SPI3);*

Cấu hình I2S:

Bây giờ thì các chân GPIO đã được cấu hình, và ta có thể sử dụng được. Do I2S là 1 phần của SPI nên các hàm cần thiết sẽ nằm trong “stm32f4xx\_spi.h”

*I2S\_InitTypeDef I2S\_InitType;*

*I2S\_InitType.I2S\_AudioFreq = I2S\_AudioFreq\_48k;*

*I2S\_InitType.I2S\_MCLKOutput = I2S\_MCLKOutput\_Enable;*

*I2S\_InitType.I2S\_Mode = I2S\_Mode\_MasterTx;*

*I2S\_InitType.I2S\_DataFormat = I2S\_DataFormat\_16b;*

*I2S\_InitType.I2S\_Standard = I2S\_Standard\_Phillips;*

*I2S\_InitType.I2S\_CPOL = I2S\_CPOL\_Low;*

***I2S\_Init****(SPI3, &I2S\_InitType);*

Sau khi khởi tạo xong, ta có thể bật ngoại vi này lên:

***I2S\_Cmd****(SPI3, ENABLE);*

Cấu hình I2C và kết nối đến CS43L22:

Giống GPIO và I2S, chân I2C cũng cần phải được cấu hình để giao tiếp được với DAC. Các hàm cần thiết nằm trong thư viện “stm32f4xx\_i2c.h”

*I2C\_InitType.I2C\_ClockSpeed = 100000;*

*I2C\_InitType.I2C\_Mode = I2C\_Mode\_I2C;*

*I2C\_InitType.I2C\_OwnAddress1 = 99;*

*I2C\_InitType.I2C\_Ack = I2C\_Ack\_Enable;*

*I2C\_InitType.I2C\_AcknowledgedAddress = I2C\_AcknowledgedAddress\_7bit;*

*I2C\_InitType.I2C\_DutyCycle = I2C\_DutyCycle\_2;*

***I2C\_Init****(I2C1, &I2C\_InitType);*

***I2C\_Cmd****(I2C1, ENABLE);*

Đến đây là hoàn thành các bước cấu hình ở phía STM32F4.

Cấu hình DAC. Đầu tiên, ta cần bật nó lên bằng các đưa mức tín hiệu cao vào chân reset:

***GPIO\_SetBits****(GPIOD, GPIO\_Pin\_4);*

CS43L22 giống như 1 con chip vi điều khiển, nó có các thanh ghi , và ta sử dụng giao thức I2C để đọc, ghi vào các thanh ghi đó. Các hàm để đọc ghi như sau:

* *I2C\_GenerateSTART*
* *I2C\_Send7bitAddress*
* *I2C\_SendData*
* *I2C\_ReceiveData*
* *I2C\_GenerateStop*

Trước khi truyền tải, ta cần kiểm tra xem CS43L22 có sẵn sàng cho việc truyền tải không bằng 2 lệnh:

* I2C\_CheckEvent  sử dụng để kiểm tra các event nếu có
* I2C\_GetFlagStatus để kiểm tra trạng thái các cờ (ví dụ kiểm tra xem I2C bus có bận không trước khi truyền tải)

Bây giờ thì có thể bắt đầu công việc truyền tải. Sử dụng lệnh sau:

*SPI\_I2S\_SendData(SPI3, theData);*

theData là 1 giá trị 16 bit không dấu (unsigned integer). Khi gọi liên tục hàm này thì sẽ tự đổi từ kênh trái kênh phải. Trước khi truyền thì nên kiểm tra xem buffer truyền có trống không bằng lệnh sau (hoặc dùng ngắt):

*SPI\_I2S\_GetFlagStatus(SPI3, SPI\_I2S\_FLAG\_TXE);*

# 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.

Group 00 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.