TẬP ĐOÀN CÔNG NGHIỆP – VIỄN THÔNG QUÂN ĐỘI

TỔNG CÔNG TY NGHIỆP CÔNG NGHỆ CAO VIETTEL

BÁO CÁO THU HOẠCH THỬ VIỆC

NGƯỜI VIẾT NGƯỜI HƯỚNG DẪN

LÊ VIẾT NHẬT QUANG TĂNG THIÊN VŨ

NGƯỜI QUẢN LÝ TRỰC TIẾP PHÒNG TỔ CHỨC LAO ĐỘNG

PHẠM VĂN HÀ LÊ THU HẰNG

Hồ Chí Minh, ngày tháng năm

MỤC LỤC

[MỤC LỤC 2](#_Toc12396015)

[DANH MỤC HÌNH VẼ 5](#_Toc12396016)

[DANH MỤC BẢNG BIỂU 7](#_Toc12396017)

[DANH MỤC TỪ VIẾT TẮT 9](#_Toc12396018)

[LỜI NÓI ĐẦU 10](#_Toc12396019)

[A. NHẬN THỨC VỀ TẬP ĐOÀN – VĂN HÓA VIETTEL 11](#_Toc12396020)

[1 Tập đoàn Công nghiệp Viễn thông Quân đội 11](#_Toc12396021)

[1.1 Những mốc son trong lịch sử phát triển của Tập đoàn 11](#_Toc12396022)

[1.2 Các dấu mốc phát triển của các dịch vụ bưu chính viễn thông 12](#_Toc12396023)

[1.3 Mô hình tổ chức của Tập đoàn tại thời điểm viết bài thu hoạch 13](#_Toc12396024)

[2 Tổng công ty công nghiệp công nghệ cao (VHT) 16](#_Toc12396025)

[2.1 Giới thiệu chung 16](#_Toc12396026)

[2.2 Nhiệm vụ 16](#_Toc12396027)

[2.3 Bộ máy tổ chức 16](#_Toc12396028)

[2.4 Trung tâm Kỹ thuật Công nghệ 17](#_Toc12396029)

[3 Triết lý thương hiệu, triết lý kinh doanh 18](#_Toc12396030)

[3.1 Tầm nhìn thương hiệu 18](#_Toc12396031)

[3.2 Ý nghĩa slogan: “Hãy Nói Theo Cách Của Bạn” 20](#_Toc12396032)

[3.3 Ý nghĩa logo 20](#_Toc12396033)

[3.4 Triết lý kinh doanh 21](#_Toc12396034)

[4 Giá trị cốt lõi của Văn hóa Viettel 21](#_Toc12396035)

[4.1 Thực tiễn là tiêu chuẩn để kiểm nghiệm chân lý 22](#_Toc12396036)

[4.2 Trưởng thành qua những thách thức và thất bại 23](#_Toc12396037)

[4.3 Thích ứng nhanh là sức mạnh cạnh tranh 23](#_Toc12396038)

[4.4 Sáng tạo là sức sống 24](#_Toc12396039)

[4.5 Tư duy hệ thống 24](#_Toc12396040)

[4.6 Kết hợp Đông – Tây 25](#_Toc12396041)

[4.7 Truyền thống và cách làm người lính 25](#_Toc12396042)

[4.8 Viettel là ngôi nhà chung 26](#_Toc12396043)

[B. CHUYÊN MÔN 27](#_Toc12396044)

[1 Kiến thức lập trình 27](#_Toc12396045)

[1.1 Linux Kernel coding style 27](#_Toc12396046)

[1.2 Tìm hiểu môi trường lập trình C trên Linux 32](#_Toc12396047)

[1.3 Sử dụng hệ điều hành Linux cho lập trình C 32](#_Toc12396048)

[1.3.1 GCC Compiler 32](#_Toc12396049)

[1.3.2 GNU Debugger 33](#_Toc12396050)

[1.3.3 Makefile và “make” system 38](#_Toc12396051)

[1.4 Ngôn ngữ lập trình C 41](#_Toc12396052)

[1.4.1 Data types, Operations, Expression 41](#_Toc12396053)

[1.4.2 Control Flow, Control Statement 44](#_Toc12396054)

[1.4.3 Function and Program Structure 48](#_Toc12396055)

[1.4.4 Con trỏ cho các 50](#_Toc12396056)

[1.4.5 Advanced Data type 51](#_Toc12396057)

[1.5 Data structure 53](#_Toc12396058)

[1.5.1 Elementary Data Structure 53](#_Toc12396059)

[1.5.2 Abstract Data Types 54](#_Toc12396060)

[1.6 Sorting algorithm 55](#_Toc12396061)

[1.7 C programming in Linux 56](#_Toc12396062)

[1.7.1 Process và Thread 56](#_Toc12396063)

[1.7.1.1 Process 56](#_Toc12396064)

[1.7.1.2 Signal 59](#_Toc12396065)

[1.7.1.3 Thread 64](#_Toc12396066)

[1.7.2 Inter-process communication 70](#_Toc12396067)

[1.7.3 Socket 75](#_Toc12396068)

[1.7.3.1 Khái niệm 75](#_Toc12396069)

[1.7.3.2 Kết nối 75](#_Toc12396070)

[1.7.3.3 Network information 78](#_Toc12396071)

[1.7.3.4 Datagram 79](#_Toc12396072)

[1.8 FreeRTOS 81](#_Toc12396073)

[1.8.1 Giới thiệu hệ điều hành FreeRTOS 81](#_Toc12396074)

[1.8.2 Quản lý Heap 85](#_Toc12396075)

[1.8.3 Quản lý Task (Section quan trọng nhất của FreeRTOS) 88](#_Toc12396076)

[1.8.4 Quản lý Queue 96](#_Toc12396077)

[1.8.5 Quản lý Software Timer 97](#_Toc12396078)

[1.8.6 Quản lý Ngắt 100](#_Toc12396079)

[1.8.7 Event group 104](#_Toc12396080)

[1.8.8 Truyền thông Task 104](#_Toc12396081)

[1.8.9 Tính năng mở rộng 105](#_Toc12396082)

[2 Tìm hiểu dự án S-Tracking 109](#_Toc12396083)

[2.1 Tổng quan dự án 109](#_Toc12396084)

[2.2 Thiết kế thiết bị S-Tracking (outdoor) 110](#_Toc12396085)

[2.2.1 Tổng quan thiết kế 110](#_Toc12396086)

[2.2.2 Chi tiết API 111](#_Toc12396087)

[2.2.3 Luồng xử lý 126](#_Toc12396088)

[2.2.4 Mã hóa gói tin 130](#_Toc12396089)

[2.3 Các chế độ debug 131](#_Toc12396090)

[2.3.1 UART debug 131](#_Toc12396091)

[2.3.2 BLE debug 135](#_Toc12396092)

[C. Kết quả đạt được trong thời gian Thử việc/Tập nghề 136](#_Toc12396093)

[D. Các Tài liệu tham khảo đã đọc, học trong thời gian Thử việc/Tập nghề 138](#_Toc12396094)

[E. Nhận xét của các cấp 139](#_Toc12396095)

[1 Nhận xét, đánh giá của người quản lý trực tiếp: 139](#_Toc12396096)

[2 Nhận xét của người quản lý chức năng (nếu có) 140](#_Toc12396097)

[3 Xét duyệt của Ban giám đốc 141](#_Toc12396098)

DANH MỤC HÌNH VẼ

[Hình 1: Sơ đồ phòng ban 15](#_Toc12398069)

[Hình 2: Mô hình tổ chức Công ty Công nghiệp Công nghệ cao Viettel 17](#_Toc12398070)

[Hình 3: Logo Viettel 20](#_Toc12398071)

[Hình 4: Một cách để giảm indentation level 27](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398072)

[Hình 5: Vị trí đặt dấu ngoặc nhọn 27](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398073)

[Hình 6: Cách đặt dấu ngoặc nhọn nếu số lượng lệnh khác nhau 28](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398074)

[Hình 7: Vị trí hàm EXPORT 29](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398075)

[Hình 8: Do-while trong việc định nghĩa MACRO 29](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398076)

[Hình 9: Truyền size của struct 30](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398077)

[Hình 10: Cấp phát bộ nhớ trong kernel space 30](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398078)

[Hình 11: Điều kiện if 44](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398079)

[Hình 12: Sơ đồ khối điều kiện if else 45](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398080)

[Hình 13: Sơ đồ khối điều kiện if else liên tiếp 45](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398081)

[Hình 14: Sơ đồ khối điều kiện if else lồng vào nhau 46](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398082)

[Hình 15: Sơ đồ khối câu lệnh switch 46](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398083)

[Hình 16: Sơ đồ khối vòng lặp for 47](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398084)

[Hình 17: Sơ đồ khối vòng lặp while 47](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398085)

[Hình 18: Sơ đồ khối vòng lặp do while 48](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398086)

[Hình 19: Phân vùng nhớ của chương trình đơn luồng (trái) và đa luồng (phải) 65](file:////Users/quanglvn/Desktop/quanglvn-bao-cao-thu-viec-1%202.docx#_Toc12398087)

[Hình 20: Vị trí của socket trong mô hình OSI 75](#_Toc12398088)

[Hình 21: Cấp trúc mảng heap\_1 được cấp phát mỗi khi tạo một task 86](#_Toc12398089)

[Hình 22: Heap\_2 xóa và tạo task 87](#_Toc12398090)

[Hình 23: Quá trình cấp phát và thu hồi vùng nhớ của Heap\_4 88](#_Toc12398091)

[Hình 24: cấu trúc hàm task 89](#_Toc12398092)

[Hình 25: Tick interrupt 91](#_Toc12398093)

[Hình 26: Task state 92](#_Toc12398094)

[Hình 27: execution sequence khi sử dụng vTaskDelay() thay cho null loop 92](#_Toc12398095)

[Hình 28: Mode 1 94](#_Toc12398096)

[Hình 29: Mode 1 với 2 task có cùng priority 94](#_Toc12398097)

[Hình 30: Mode 1 với 2 task có cùng priority nhưng cờ configIDLE\_SHOUD\_YIELD được set thành 1 95](#_Toc12398098)

[Hình 31: Mode 2 95](#_Toc12398099)

[Hình 32: Mode 3 95](#_Toc12398100)

[Hình 33: đọc ghi data tới queue 97](#_Toc12398101)

[Hình 34: Sự khác nhau giữa one-shot và periodic software timer. 98](#_Toc12398102)

[Hình 35: Các hàm chuyển đổi trạng thái cho periodic (trái) và one-shot (phải) 98](#_Toc12398103)

[Hình 36: Software timer gửi command tới daemon task 99](#_Toc12398104)

[Hình 37: Task 2 có độ ưu tiên cao nhất sẽ hoàn thành công việc của ISR 101](#_Toc12398105)

[Hình 38: Sử dụng binary semaphore để implement deferred interrupt processing 102](#_Toc12398106)

[Hình 39: Sử dụng binary semaphore để đồng bộ task với ngắt 103](#_Toc12398107)

[Hình 40: Các chức năng phân tích của FreeRTOS+Trace được đồ thị hóa 106](#_Toc12398108)

[Hình 41: Kết quả của hàm vTaskGetRunTimeStats() 108](#_Toc12398109)

[Hình 42: Sơ đồ tổng quát thiết bị outdoor 110](#_Toc12398110)

[Hình 43: Kiến trúc phần mềm 111](#_Toc12398111)

[Hình 44: Luồng khởi động 126](#_Toc12398112)

[Hình 45: Luồng đăng nhập 127](#_Toc12398113)

[Hình 46: Luồng kiểm tra phần cứng 128](#_Toc12398114)

[Hình 47: Luồng chạy chính 129](#_Toc12398115)

[Hình 48: Luồng gửi thiết bị 130](#_Toc12398116)

[Hình 49: Mã hóa gói tin 130](#_Toc12398117)

[Hình 50: Lưu đồ giải thuật để vào chế độ UART debug 132](#_Toc12398118)

[Hình 51: Lưu đồ giải thuật để vào chế độ BLE debug 135](#_Toc12398119)

DANH MỤC BẢNG BIỂU

[Bảng 1: Các mốc son lích sử của Tập Đoàn 11](#_Toc12399047)

[Bảng 2: Mốc phá triển trong dịch vụ bưu chính viễn thông 12](#_Toc12399048)

[Bảng 3: Đáp ứng của Viettel đối với khách hàng 19](#_Toc12399049)

[Bảng 4: Cờ của gcc 32](#_Toc12399050)

[Bảng 5: GDB command 33](#_Toc12399051)

[Bảng 6: Điều kiện trong make 39](#_Toc12399052)

[Bảng 7: Phân loại kiểu dữ liệu trong C 41](#_Toc12399053)

[Bảng 8: Kiểu nguyên 41](#_Toc12399054)

[Bảng 9: Kiểu chấm động 41](#_Toc12399055)

[Bảng 10: Các kiểu trả về của hàm 41](#_Toc12399056)

[Bảng 11: Chuỗi escape của C 42](#_Toc12399057)

[Bảng 12: Toán tử số học 42](#_Toc12399058)

[Bảng 13: Toán tử gán 42](#_Toc12399059)

[Bảng 14: Toán tử so sánh 43](#_Toc12399060)

[Bảng 15: Toán tử logic 43](#_Toc12399061)

[Bảng 16: Toán tử bitwise 43](#_Toc12399062)

[Bảng 17: Các toán tử khác 43](#_Toc12399063)

[Bảng 18: Các cách truyền tham số 49](#_Toc12399064)

[Bảng 19: Giá trị khởi tạo mặc định trong C 49](#_Toc12399065)

[Bảng 20: Tiền xử lý trong C 49](#_Toc12399066)

[Bảng 21: Các khái niệm liên quan tới mảng 50](#_Toc12399067)

[Bảng 22: Thao tác với con trỏ 51](#_Toc12399068)

[Bảng 23: Linked list vs Array pros and cons 53](#_Toc12399069)

[Bảng 24: So sánh các giải thuật sắp xếp 55](#_Toc12399070)

[Bảng 25: Các trạng thái của process 56](#_Toc12399071)

[Bảng 26: Kiểm tra quá trình kết thúc của process con thông qua biến stat\_loc 58](#_Toc12399072)

[Bảng 27: Các hàm an toàn để sử dụng trong signal handler 61](#_Toc12399073)

[Bảng 28: Các signal thể hiện sự kết thúc bất thường của một process 62](#_Toc12399074)

[Bảng 29: Signal gây ra sự kết thúc bất thường của process 63](#_Toc12399075)

[Bảng 30: Các signal gây tạm dùng process 63](#_Toc12399076)

[Bảng 31: Signal dùng để điều khiển việc dừng và chạy tiếp của process con 63](#_Toc12399077)

[Bảng 32: Thay đổi sau khi đóng fd và gọi hàm dup 73](#_Toc12399078)

[Bảng 33: Các chọn lựa hợp lệ 74](#_Toc12399079)

[Bảng 34: Một số khái niệm được dùng khi giới thiệu về hệ điều hành FreeRTOS 82](#_Toc12399080)

[Bảng 35: FreeRTOS convention 83](#_Toc12399081)

[Bảng 36: Các mode hoạt động của FreeRTOS scheduler 93](#_Toc12399082)

[Bảng 37: UART driver 111](#_Toc12399083)

[Bảng 38: SPI driver 112](#_Toc12399084)

[Bảng 39: I2C driver 113](#_Toc12399085)

[Bảng 40: GPS API 114](#_Toc12399086)

[Bảng 41: Iridium API 115](#_Toc12399087)

[Bảng 42: M95 API 118](#_Toc12399088)

[Bảng 43: GPRS API 120](#_Toc12399089)

[Bảng 44: Tracking API 121](#_Toc12399090)

[Bảng 45: User API 123](#_Toc12399091)

[Bảng 46: Các gói tin 130](#_Toc12399092)

[Bảng 47: Kết nối chân giữa board UART-To-USB với S-ALARM 131](#_Toc12399093)

[Bảng 48: Tập lệnh hỗ trợ kiểm tra module và cấu hình thiết bị 132](#_Toc12399094)

[Bảng 49: Tập lệnh hỗ trợ kiểm tra module và cấu hình thiết bị (tiếp theo) 133](#_Toc12399095)

DANH MỤC TỪ VIẾT TẮT

|  |  |
| --- | --- |
| BLE | Bluetooth Low Energy |
| FIFO | First In First Out |
| GCC | GNU Compiler Collection |
| GDB | GNU Debugger |
| ISR | Interrupt Service Routine |
| PID | Process Identifier |
| RTOS | Real Time Operating System |
| VHT | Viettel High Technology |

LỜI NÓI ĐẦU

Qua gần 30 năm sáng tạo và phát triển, Tập đoàn Viễn thông Quân đội Viettel đã vươn mình trở thành tập đoàn viễn thông hàng đầu Việt Nam và là thương hiệu uy tín ở Việt Nam cũng như trên Thế giới. Với các lĩnh vực ngành nghề hoạt động rộng khắp và phương châm phát triển nhanh, bền vững, Viettel đã và đang có mặt ở mọi miền đất nước, không chỉ mang đến những dịch vụ đột phá làm thay đổi bộ mặt nền viễn thông nước nhà mà còn đóng góp vào nền kinh tế quốc gia những giá trị to lớn. Bên cạnh đó, Viettel còn đạt được những thành công to lớn khi vươn ra nước ngoài, đạt được những mục tiêu thứ hạng cao trên bản đồ viễn thông Thế giới.

Sau thời gian gần hai tháng tham gia thử việc tại Trung tâm Nghiên cứu Công nghệ mạng Viettel, tôi nhận thức được sức mạnh của Viettel đến từ chính sự kỷ luật và sáng tạo của mỗi nhân viên, đóng góp vào thành công chung của tập thể. Báo cáo thử việc này được thực hiện dưới sự phân công và hướng dẫn của Trung tâm Kỹ thuật Công nghệ. Nội dung báo cáo bao gồm:

Phần A: Nhận thức về Tập đoàn – Văn hóa Viettel

Phần B: Chuyên môn

Phần C: Các kết quả đạt được trong thời gian thử việc

Phần D: Các tài liệu đã đọc, học trong thời gian thử việc

Phần E: Nhận xét của các cấp

Trong quá trình thử việc tôi đã nhận được sự hướng dẫn tận tình của các anh/chị trong nhóm firmware của dự án EnodeB và dự án S-Tracking, cùng toàn thể anh chị em trong Trung tâm Kỹ Thuật Công nghệ. Tôi xin chân thành cảm ơn!

Hồ Chí Minh, ngày 28 tháng 06 năm 2019

Người viết báo cáo

**Lê Viết Nhật Quang**

# NHẬN THỨC VỀ TẬP ĐOÀN – VĂN HÓA VIETTEL

# Tập đoàn Công nghiệp Viễn thông Quân đội

## Những mốc son trong lịch sử phát triển của Tập đoàn

Bảng 1: Các mốc son lích sử của Tập Đoàn

|  |  |
| --- | --- |
| Thời gian | Sự kiện |
| 01/06/1989 | Thành lập Tổng Công ty Điện tử thiết bị thông tin trực thuộc Bộ Tư lệnh Thông tin liên lạc, Bộ Quốc phòng. |
| 07/1993 | Tổng công ty Điện tử thiết bị thông tin được tổ chức lại thành Công ty Điện tử thiết bị thông tin. |
| 27/07/1993 | Bộ Quốc phòng ra quyết định số 336/QĐ-QP thành lập lại doanh nghiệp Nhà nước: Công ty Điện tử thiết bị thông tin, tên giao dịch là SIGENCO, trụ sở chính tại 16 Cát Linh, Hà Nội. |
| 14/07/1995 | Bộ Quốc phòng ra Quyết định số 615/QĐ- QP đổi tên Công ty Điện tử thiết bị thông tin thành Công ty Điện tử - viễn thông Quân đội, tên giao dịch quốc tế là VIETEL. |
| 29/04/2003 | Thủ tướng Chính phủ ra Quyết định số 80/2003/QĐ-TTg phê duyệt phương án tổng thể sắp xếp, đổi mới doanh nghiệp Nhà nước trực thuộc Bộ Quốc phòng giai đoạn 2003-2005. |
| 27/04/2004 | Bộ trưởng Bộ Quốc phòng ra quyết định số 51/QĐ-QP quyết định từ 01 tháng 7 năm 2004 điều chuyển Công ty viễn thông Quân đội từ Bộ Tư lệnh Thông tin về trực thuộc Bộ Quốc Phòng với tên gọi Công ty Viễn thông Quân đội tên giao dịch là VIETTEL. |
| 02/03/2005 | Thủ tướng Phan Văn Khải đã ký quyết định thành lập Tổng công ty Viễn thông Quân đội (từ Công ty phát triển thành Tổng Công ty). |
| 06/04/2005 | Bộ Quốc phòng có quyết định số 45/2005/BQP về việc thành lập Tổng công ty Viễn thông Quân đội, tên giao dịch quốc tế tiếng Anh là VIETTEL CORPORATION, viết tắt là VIETTEL. |
| 14/12/2009 | Thủ tướng Chính phủ ký quyết định số 2078/QĐ-TTg về việc phê duyệt  Đề án thí điểm thành lập Tập đoàn Viễn thông Quân đội và Quyết định số 2079/QĐ-TTg thành lập Tập đoàn Viễn thông Quân đội. |
| 12/01/2010 | Tại trụ sở số 1 Giang Văn Minh, Ba Đình, Hà Nội, Viettel đã long trọng tổ chức Lễ ra mắt Tập đoàn và đón nhận Huân chương Độc lập Hạng Ba. |
| 05/01/2018 | Chính phủ ban hành Nghị định số 05/2018/NĐ-CP về Điều lệ tổ chức và hoạt động của Công ty mẹ - Tập đoàn Viettel. Theo đó, Tập đoàn Công nghiệp – Viễn thông Quân đội được đổi tên từ Tập đoàn Viễn thông Quân đội là doanh nghiệp quốc phòng, an ninh, do Nhà nước nắm giữ 100% vốn điều lệ. |

## Các dấu mốc phát triển của các dịch vụ bưu chính viễn thông

Bảng 2: Mốc phá triển trong dịch vụ bưu chính viễn thông

|  |  |
| --- | --- |
| Thời gian | Sự kiện |
| 01/07/1997 | Triển khai dịch vụ Bưu chính. |
| 05/10/2000 | Thử nghiệm dịch vụ điện thoại đường dài 178, công nghệ VoIP. |
| 09/10/2002 | Khai trương dịch vụ Internet. |
| 09/2003 | Triển khai dịch vụ điện thoại cố định. |
| 15/10/2004 | Khai trương dịch vụ Điện thoại Di động. |
| 2006 | Đầu tư sang Campuchia. |
| 2007 | Đầu tư sang Lào. |
| 03/2007 | Triển khai dịch vụ Điện thoại cố định không dây. |
| 19/02/2009 | Khai trương dịch vụ Metfone tại Campuchia. |
| 16/10/2009 | Khai trương dịch vụ Unitel tại Lào. |
| 25/03/2010 | Khai trương dịch vụ 3G tại Việt Nam. |
| 07/09/2011 | Khai trương mạng viễn thông Natcom tại Haiti. |
| 15/05/2012 | Khai trương mạng viễn thông Movitel tại Mozambique. |
| 10/7/2013 | Khai trương mạng Telemor tại Đông Timor. |
| 12/9/2014 | Khai trương mạng Nexttel tại Camaroon. |
| 15/10/2014 | Khai trương mạng Bitel tại Peru. |
| 01/2015 | Thủ tướng phê duyệt dự án A1. |
| 03/2015 | Khai trương Lumitel tại Burundi. |
| 10/2015 | Khai trương Halotel tại Tanzania. |
| 12/2015 | Thử nghiệm 4G tại Vũng Tàu. |

## Mô hình tổ chức của Tập đoàn tại thời điểm viết bài thu hoạch

Trụ sở giao dịch: Số 1 – Trần Hữu Dực – Mỹ Đình – Từ Liêm – Hà Nội.

Website: [www.viettel.com.vn](http://www.viettel.com.vn/)

Tên cơ quan sáng lập: Bộ Quốc phòng

Tập đoàn Viễn thông Quân đội được thành lập theo quyết định 2097/2009/QĐ-TTg của Thủ tướng Chính phủ ký vào ngày 14/12/2009, là doanh nghiệp kinh tế quốc phòng

100% vốn nhà nước với số vốn điều lệ 50.000 tỷ đồng, có tư cách pháp nhân, có con dấu,

biểu tượng và điều lệ tổ chức riêng.

**Hoạt động kinh doanh:**

* Cung cấp dịch vụ Viễn thông
* Truyền dẫn
* Bưu chính
* Phân phối thiết bị đầu cuối
* Đầu tư tài chính
* Truyền thông
* Đầu tư bất động sản
* Xuất nhập khẩu
* Đầu tư nước ngoài

**Ban Tổng giám đốc tập đoàn:**

Chủ tịch kiêm Tổng Giám đốc: Ủy viên Trung ương Đảng, Thiếu tướng Nguyễn Mạnh Hùng

Quyền chủ tịch: Thiếu tướng Lê Đăng Dũng

Phó Tổng Giám đốc: Đại tá Hoàng Sơn

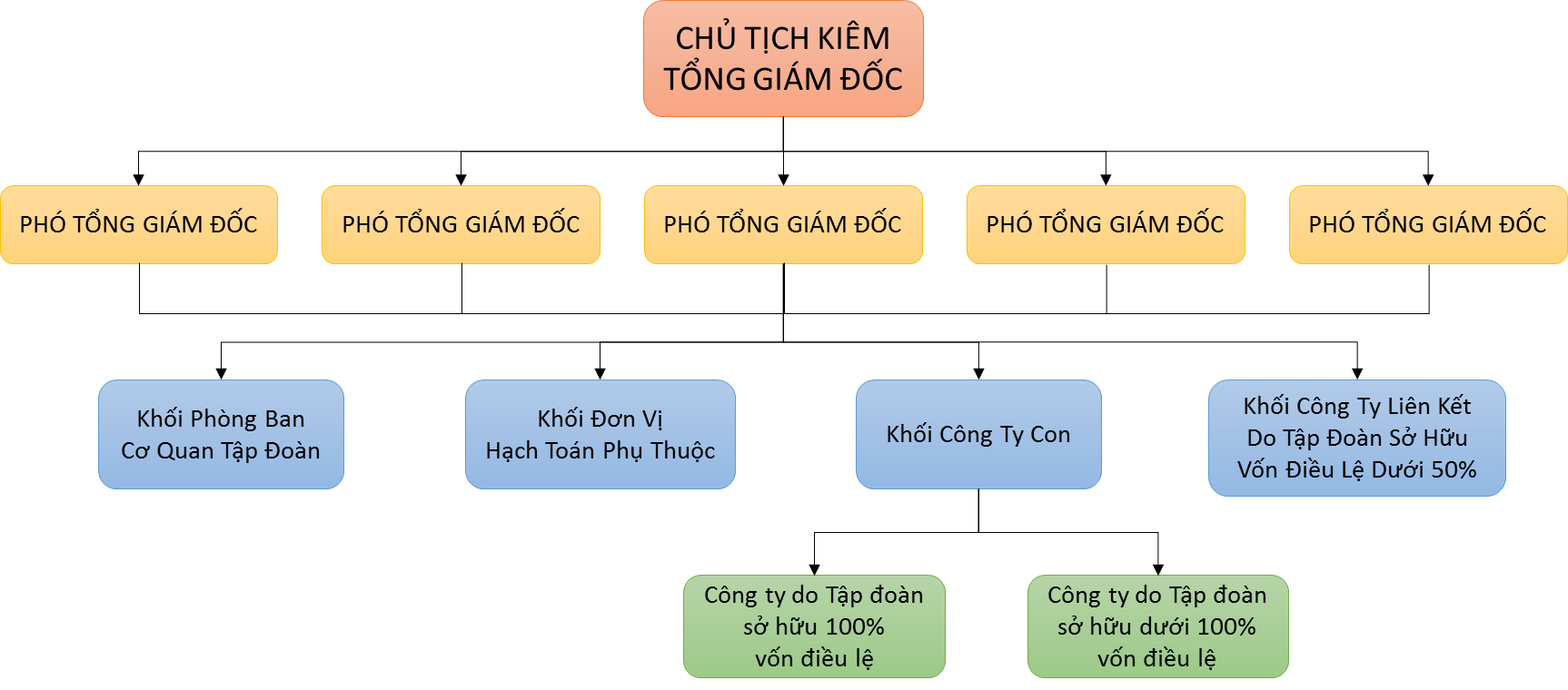
Phó Tổng Giám đốc: Đại tá Nguyễn Đình Chiến

Phó Tổng Giám đốc: Thượng tá Đỗ Minh Phương

Phó Tổng Giám đốc: Thượng tá Tào Tức Thắng Phó Tổng Giám đốc: Thượng tá Nguyễn Thanh Nam

Sơ đồ tổ chức tập đoàn Viettel:

Tập đoàn Viễn thông Quân đội Viettel gồm các khối phòng ban và cơ quan chức năng như sau:



Hình 1: Sơ đồ phòng ban

# Tổng công ty công nghiệp công nghệ cao (VHT)

## Giới thiệu chung

Công ty công nghiệp công nghệ cao Viettel được thành lập với tiền thân là Trung tâm nghiên cứu và phát triển công nghệ mạng (VTTEK), được xây dựng dựa trên cơ sở sáp nhập nguyên trạng Trung tâm Nghiên cứu thiết bị mạng viễn thông Viettel (VTCore) và Trung tâm Nghiên cứu thiết bị truy cập vô tuyến băng rộng Viettel. Công ty công nghiệp công nghệ cao Viettel là đơn vị hạch toán phụ thuộc Tập đoàn, có đăng ký hoạt động, có con dấu và mã số thuế riêng, hoạt động trong phạm vi ủy quyền bằng văn bản của Tổng giám đốc Tập đoàn cho Công ty công nghiệp công nghệ cao Viettel. Tên tiếng Anh: Viettel High Technologies (viết tắt: VHT).

## Nhiệm vụ

Nhiệm vụ chính của VHT tập trung vào việc nghiên cứu công nghệ, sản xuất thiết bị mạng viễn thông gồm: hệ thống truy cập di động, cố định; thiết bị truyền dẫn; hệ thống mạng lõi; nền tảng VAS, cụ thể gồm:

Nghiên cứu thiết bị mạng lõi 3G, 4G EPC.

Nghiên cứu sản xuất thiết bị truy cập vô tuyến 4G, 5G.

Nghiên cứu sản xuất các thiết bị truyền dẫn, cố định băng rộng: Site Router, GPON.

Nghiên cứu tối ưu làm chủ nền tảng sản phẩm tính cước thời gian thực (vOCS).

Nghiên cứu phát triển các công nghệ lõi, công nghệ nền tảng. Làm chủ từ công nghệ, thiết kế, chế tạo; hệ thống cung ứng, qui trình sản xuất, đánh giá chất lượng sản phẩm.

Các nhiệm vụ khác do Ban giám đốc Tập đoàn giao.

## Bộ máy tổ chức

Tổ chức bộ máy của VPT gồm Ban giám đốc, khối phòng ban quản lý, hỗ trợ và khối trực tiếp sản xuất.

Ban giám đốc tổng công ty gồm:

* Tổng Giám đốc: Trung tá Nguyễn Vũ Hà
* Phó Tổng Giám đốc: Thiếu tá Nguyễn Minh Quang
* Phó Tổng Giám đốc: Trung tá Đỗ Thanh Hải
* Phó Tổng Giám đốc: Đại úy Đào Vũ Kiên
* Phó Tổng Giám đốc: Thiếu tá Nguyễn Cương Hoàng



Hình 2: Mô hình tổ chức Công ty Công nghiệp Công nghệ cao Viettel

## Trung tâm Kỹ thuật Công nghệ

Trung tâm Kỹ thuật Công nghệ thuộc khối trực tiếp nghiên cứu, sản xuất kinh doanh của VHT, có chức năng, nhiệm vụ như sau:

*Chức năng*

* Nghiên cứu, làm chủ các công nghệ nền tảng dùng chung, ứng dụng và triển khai vào các dự án sản xuất trong Trung tâm VPT.
* Nghiên cứu các xu hướng công nghệ mới và đánh giá khả năng ứng dụng vào các dự án sản xuất cũng như đánh giá khả năng xây dựng các sản phẩm, dịch vụ mới.
* Hợp tác với các đơn vị nghiên cứu, sản xuất trong và ngoài nước trong các lĩnh vực liên quan.
* Tham mưu, đề xuất với Ban Giám đốc Trung tâm các giải pháp, xu hướng về công nghệ ứng dụng trong các sản phẩm của Trung tâm.

*Nhiệm vụ*

* Nghiên cứu chuyên sâu, làm chủ các công nghệ sử dụng trong các hệ thống hiệu năng cao, xây dựng nền tảng dùng chung cho các sản phẩm, thiết bị mạng viễn thông.
* Nghiên cứu công nghệ 5G, tham gia các tổ chức xây dựng chuẩn 5G nhằm định hướng cho việc phát triển các sản phẩm 5G.
* Nghiên cứu, xây dựng nền tảng tính toán trên bộ nhớ để thay thế các hệ thống inmemory đang sử dụng tại các sản phẩm của trung tâm.
* Nghiên cứu nền tảng ảo hóa, các công nghệ NFV, SDN làm nền tảng để ảo hóa các hệ thống mạng lõi, xây dựng mạng lõi cho mạng 5G.
* Nghiên cứu, xây dựng nền tảng báo hiệu trong mạng viễn thông để sử dụng trong các hệ thống mạng lõi viễn thông: OCS, SMSC, LBS, MSC,...
* Nghiên cứu các công nghệ, nền tảng sử dụng trong việc phân tích, đánh giá dữ liệu lớn, học máy và khoa học dữ liệu làm căn cứ để triển khai thông minh hóa các hệ thống viễn thông.
* Nghiên cứu, tìm hiểu các xu hướng công nghệ mới.
* Độc lập giám sát, phân tích hiệu năng của các công nghệ ứng dụng trong các dự án của Trung tâm, đảm bảo hiệu năng, độ tin cậy của các giải pháp được triển khai thực tế.
* Chịu trách nhiệm và báo cáo Ban Giám đốc Trung tâm về các công việc theo chức năng nhiệm vụ. Thực hiện các nhiệm vụ khác khi được Ban Giám đốc Trung tâm giao.

# Triết lý thương hiệu, triết lý kinh doanh

## Tầm nhìn thương hiệu

Tầm nhìn thương hiệu được cô đọng từ việc thấu hiểu những mong muốn của khách hàng và những nỗ lực đáp ứng của VIETTEL.

VIETTEL hiểu rằng, khách hàng luôn muốn được lắng nghe, quan tâm chăm sóc như những cá thể riêng biệt.

Còn VIETTEL sẽ nỗ lực để sáng tạo phục vụ những nhu cầu riêng biệt ấy với một sự chia sẻ, thấu hiểu nhất.

Bảng 3: Đáp ứng của Viettel đối với khách hàng

|  |  |  |
| --- | --- | --- |
| Mong muốn của Khách hàng | | Sự đáp ứng của Viettel |
| Được đối xử | Mong muốn được lắng nghe.  Muốn được người khác hiểu nhu cầu và ước muốn.  Muốn được nhìn nhận như một cá thể riêng biệt. | Viettel là người tiên phong. |
| Tiêu chuẩn dịch vụ | Muốn được đáp ứng.  Muốn phục vụ bởi những nhà cung cấp tin cậy về chất lượng.  Muốn được quan tâm, chăm sóc. | Viettel luôn luôn đổi mới.  Viettel luôn sáng tạo. |
| Quyền của Khách hàng | Muốn được đối xử công bằng, thẳng thắn.  Muốn có được nhiều sự lựa chọn.  Khách hàng muốn được lắng nghe, quan tâm chăm sóc như những cá thể riêng biệt. | Viettel lắng nghe.  Viettel thẳng thắn.  Viettel nhân từ.  Viettel thành thật.  Viettel có tinh thần giúp đỡ, hỗ trợ.  Viettel thông cảm với mọi người.  Viettel là nhà sáng tạo với một trái tim nhân từ. |

Ý nghĩa tầm nhìn thương hiệu Viettel

Tầm nhìn thương hiệu định ra một hướng đi chung cho các hoạt động của Viettel, được cô đọng từ việc tổng hợp cơ sở mong muốn của khách hàng và sự đáp ứng của Viettel, kết hợp giữa văn hoá phương Đông và phương Tây.

*INNOVATOR (phương Tây):*

* Tiên phong, sáng tạo.
* Liên tục đổi mới, cải cách.
* Làm việc và tư duy logic có hệ thống.
* Cá thể hoá.

*CARING (phương Đông):*

Luôn lắng nghe, quan tâm, chăm sóc.

Tư duy trực quan sinh động.

Cơ chế cân bằng, ổn định.

Tình cảm, có trách nhiệm xã hội, tham gia các hoạt động nhân đạo.

## Ý nghĩa slogan: “Hãy Nói Theo Cách Của Bạn”

VIETTEL luôn mong muốn phục vụ khách hàng như những cá thể riêng biệt. VIETTEL hiểu rằng, muốn làm được điều đó phải thấu hiểu khách hàng, phải lắng nghe khách hàng. Và vì vậy, khách hàng được khuyến khích nói theo cách mà họ mong muốn và bằng tiếng nói của chính mình – “Hãy nói theo cách của bạn”.

## Ý nghĩa logo



Hình 3: Logo Viettel

Logo VIETTEL được thiết kế dựa trên ý tưởng lấy từ hình tượng dấu ngoặc kép. Khi bạn trân trọng câu nói của ai đó, bạn sẽ trích dẫn trong dấu ngoặc kép. VIETTEL quan tâm và trân trọng từng nhu cầu cá nhân của mỗi khách hàng. VIETTEL luôn luôn biết lắng nghe và cảm nhận, trân trọng những ý kiến của mọi người như những cá thể riêng biệt – các thành viên của Tập đoàn, khách hàng và đối tác. Đây cũng chính là nội dung của câu khẩu hiệu của VIETTEL: Hãy nói theo cách của bạn (Say it your way).

Logo VIETTEL mang hình elip được thiết kế đi từ nét nhỏ đến nét lớn, nét lớn lại đến nét nhỏ, biểu tượng cho sự chuyển động liên tục, sáng tạo không ngừng (văn hóa phương

Tây) đồng thời cũng biểu tượng cho âm dương hòa quyện vào nhau (văn hóa phương Đông). Khối chữ VIETTEL được thiết kế có sự liên kết với nhau, thể hiện sự gắn kết, đồng lòng, kề vai sát cánh của các thành viên trong Tập đoàn.

Ba màu trên logo biểu tượng cho những ý nghĩa đặc biệt: màu xanh (thiên), màu vàng (địa), màu trắng (nhân). Màu xanh thiên thanh biểu hiện cho màu của trời, màu của khát vọng vươn lên, màu của không gian sáng tạo. Màu vàng đất biểu thị cho đất, màu của sự đầm ấm, gần gũi, đôn hậu, đón nhận. Màu trắng là nền của chữ Viettel, thể hiện sự chân thành, thẳng thắn, nhân từ. Sự kết hợp giao hòa giữa trời, đất và con người “Thiên thời – Địa lợi – Nhân hoà”theo những quan điểm của triết học và cũng gắn liền với lịch sử, định hướng của Tập đoàn thể hiện cho sự phát triển vững bền của thương hiệu VIETTEL.

## Triết lý kinh doanh

Mỗi khách hàng là một con người – một cá thể riêng biệt, cần được tôn trọng, quan tâm và lắng nghe, thấu hiểu và phục vụ một cách riêng biệt. Liên tục đổi mới, cùng với khách hàng sáng tạo ra các sản phẩm, dịch vụ ngày càng hoàn hảo.

Nền tảng cho một doanh nghiệp phát triển là xã hội. VIETTEL cam kết tái đầu tư lại cho xã hội thông qua việc gắn kết các hoạt động sản xuất kinh doanh với các hoạt động xã hội, hoạt động nhân đạo.

# Giá trị cốt lõi của Văn hóa Viettel

Là một Tập đoàn thuộc Quân Đội nên Viettel đã lấy bản chất Bộ đội Cụ Hồ để xây dựng nền văn hoá của riêng mình. Nó không cứng nhắc, không mang tính cục bộ, mà luôn mềm dẻo, hướng ngoại, học hỏi. Qua 29 năm phát triển, nét văn hoá riêng của Viettel đã mang đậm tính chất doanh nhân, được thể hiện một cách linh hoạt qua cách ứng xử với thị trường, sẵn sàng giúp đỡ, thấu hiểu và chịu trách nhiệm cao thông qua việc đưa ra các giải pháp, các sản phẩm, dịch vụ hoàn hảo nhằm đáp ứng tốt nhất nhu cầu của khách hàng, đem lại sự lựa chọn tối ưu nhất của mọi khách hàng.

Nét văn hoá của Viettel từng bước được xây dựng mang đậm tính chuyên nghiệp, thể hiện của một Tập đoàn có tầm cỡ quốc tế. Viettel luôn coi con người là chủ thể để phát triển, do vậy mà nét văn hoá tổ chức được xây dựng có tính hướng ngoại với quan điểm cá thể con người với con người (Viettel với khách hàng) nên khách hàng là đối tượng được trân trọng, sẵn sàng được phục vụ, đáp ứng các dịch vụ tốt nhất. Mặt khác, luôn coi trọng nhân viên Tập đoàn với tình cảm chân thành, tạo điều kiện, khuyến khích, hỗ trợ phát huy khả năng, năng lực, quan tâm đến chính sách đào tạo và trọng dụng nhân tài, đề cao vai trò của từng cá nhân, con người trong sự phát triển của Tập đoàn và chăm lo đến đời sống tinh thần, vật chất cho cán bộ công nhân viên của mình, điều này xác định tính hướng nội của nét văn hoá và cũng là chính sách nhân sự của Viettel.

Nét văn hoá của Viettel là sự kết hợp hài hoà hai phong cách *hướng ngoại* và *hướng nội*, với đặc trưng là sự phối hợp của hai tính chất kiểu văn hoá tổ chức *doanh nhân* và *chuyên nghiệp*. Đây là niềm tự hào mà mỗi nhân viên Viettel đều cảm nhận và tự giác tuân thủ thực hiện để xây dựng và phát triển truyền thống tốt đẹp, khẳng định một phong cách riêng, một văn hoá tổ chức của chính mình.

Nét văn hoá của Viettel đã được khẳng định và được đúc kết thành 8 giá trị cốt lõi của Viettel:

1. Thực tiễn là tiêu chuẩn để kiểm nghiệm chân lý.
2. Trưởng thành qua những thách thức và thất bại.
3. Thích ứng nhanh là sức mạnh cạnh tranh.
4. Sáng tạo là sức sống.
5. Tư duy hệ thống.
6. Kết hợp Đông – Tây.
7. Truyền thống và cách làm người lính.
8. Viettel là ngôi nhà chung.

## Thực tiễn là tiêu chuẩn để kiểm nghiệm chân lý

*Chúng ta nhận thức:*

* Lý thuyết màu xám, chỉ có cây đời là mãi xanh tươi. Lý luận để tổng kết thực tiễn rút ra kinh nghiệm, tiệm cận chân lý và dự đoán tương lai. Chúng ta cần có lý luận và dự đoán để dẫn dắt. Nhưng chỉ có thực tiễn mới khẳng định được những lý luận và dự đoán đó đúng hay sai.
* Chúng ta nhận thức và tiếp cận chân lý thông qua thực tiễn hoạt động.

*Chúng ta hành động:*

* Phương châm hành động của chúng ta “Dò đá qua sông”và liên tục điều chỉnh cho phù hợp với thực tiễn.
* Chúng ta đánh giá con người thông qua quá trình thực tiễn.

## Trưởng thành qua những thách thức và thất bại

*Chúng ta nhận thức:*

* Thách thức là chất kích thích. Khó khăn là lò luyện. “Vứt nó vào chỗ chết thì nó sẽ sống”.
* Chúng ta không sợ mắc sai lầm. Chúng ta chỉ sợ không dám nhìn thẳng vào sai lầm để tìm cách sửa. Sai lầm là không thể tránh khỏi trong quá trình tiến tới mỗi thành công. Sai lầm tạo ra cơ hội cho sự phát triển tiếp theo.

*Chúng ta hành động:*

* Chúng ta là những người dám thất bại. Chúng ta động viên những ai thất bại. Chúng ta tìm trong thất bại những lỗi sai của hệ thống để điều chỉnh. Chúng ta không cho phép tận dụng sai lầm của người khác để đánh đổ người đó. Chúng ta sẽ không lặp lại những lỗi lầm cũ.
* Chúng ta phê bình thẳng thắn và xây dựng ngay từ khi sự việc còn nhỏ. Chúng ta thực sự cầu thị, cầu sự tiến bộ.

## Thích ứng nhanh là sức mạnh cạnh tranh

*Chúng ta nhận thức:*

* Cái duy nhất không thay đổi đó là sự thay đổi. Trong môi trường cạnh tranh sự thay đổi diễn ra từng ngày, từng giờ. Nếu nhận thức được sự tất yếu của thay đổi thì chúng ta sẽ chấp nhận thay đổi một cách dễ dàng hơn.
* Mỗi giai đoạn, mỗi qui mô cần một chiến lược, một cơ cấu mới phù hợp. Sức mạnh ngày hôm nay không phải là tiền, là qui mô mà là khả năng thay đổi nhanh, thích ứng nhanh.
* Cải cách là động lực cho sự phát triển.

*Chúng ta hành động:*

* Tự nhận thức để thay đổi. Thường xuyên thay đổi để thích ứng với môi trường thay đổi. Chúng ta sẽ biến thay đổi trở thành bình thường như không khí thở vậy.
* Liên tục tư duy để điều chỉnh chiến lược và cơ cấu lại tổ chức cho phù hợp.

## Sáng tạo là sức sống

*Chúng ta nhận thức:*

* Sáng tạo tạo ra sự khác biệt. Không có sự khác biệt tức là chết. Chúng ta hiện thực hoá những ý tưởng sáng tạo không chỉ của riêng chúng ta mà của cả khách hàng.

*Chúng ta hành động:*

* Suy nghĩ không cũ về những gì không mới. Chúng ta trân trọng và tôn vinh từ những ý tưởng nhỏ nhất.
* Chúng ta xây dựng một môi trường khuyến khích sáng tạo để mỗi người Viettel hàng ngày có thể sáng tạo.
* Chúng ta duy trì Ngày hội ý tưởng Viettel.

## Tư duy hệ thống

*Chúng ta nhận thức:*

Môi trường kinh doanh ngày càng phức tạp. Tư duy hệ thống là nghệ thuật để đơn giản hoá cái phức tạp.

* Một tổ chức phải có tư tưởng, tầm nhìn chiến lược, lý luận dẫn dắt và hệ thống làm nền tảng. Một hệ thống muốn phát triển nhanh về qui mô thì phải chuyên nghiệp hoá.
* Một hệ thống tốt thì con người bình thường có thể tốt lên. Hệ thống tự nó vận hành phải giải quyết được trên 70% công việc. Nhưng chúng ta cũng không để tính hệ thống làm triệt tiêu vai trò các cá nhân.

*Chúng ta hành động:*

* Chúng ta xây dựng hệ thống lý luận cho các chiến lược, giải pháp, bước đi và phương châm hành động của mình.
* Chúng ta vận dụng qui trình 5 bước để giải quyết vấn đề: Chỉ ra vấn đề  Tìm nguyên nhân  Tìm giải pháp  Tổ chức thực hiện  Kiểm tra và đánh giá thực hiện.
* Người Viettel phải hiểu vấn đề đến gốc: Làm được là 40%  Nói được cho người khác hiểu là 30%  Viết thành tài liệu cho người đến sau sử dụng là 30% còn lại.
* Chúng ta sáng tạo theo quy trình: Ăn  Tiêu hoá  Sáng tạo.

## Kết hợp Đông – Tây

*Chúng ta nhận thức:*

* Có hai nền văn hoá, hai cách tư duy, hai cách hành động lớn nhất của văn minh nhân loại. Mỗi cái có cái hay riêng có thể phát huy hiệu quả cao trong từng tình huống cụ thể. Vậy tại sao chúng ta không vận dụng cả hai cách đó?
* Kết hợp Đông – Tây cũng có nghĩa là luôn nhìn thấy hai mặt của một vấn đề. Kết hợp không có nghĩa là pha trộn.

*Chúng ta hành động:*

* Chúng ta kết hợp tư duy trực quan với tư duy phân tích và hệ thống.

Chúng ta kết hợp sự ổn định và cải cách.

* Chúng ta kết hợp cân bằng và động lực cá nhân.

## Truyền thống và cách làm người lính

*Chúng ta nhận thức:*

* Viettel có cội nguồn từ Quân đội. Chúng ta tự hào với cội nguồn đó.
* Một trong những sự khác biệt tạo nên sức mạnh Viettel là truyền thống và cách làm quân đội.

*Chúng ta hành động:*

* Truyền thống: Kỷ luật, Đoàn kết, Chấp nhận gian khổ, Quyết tâm vượt khó khăn, Gắn bó máu thịt.
* Cách làm: Quyết đoán, Nhanh, Triệt để.

## Viettel là ngôi nhà chung

*Chúng ta nhận thức:*

* Viettel là ngôi nhà thứ hai mà mỗi chúng ta sống và làm việc ở đó. Mỗi người Viettel phải trung thành với sự nghiệp của Công ty. Chúng ta phải hạnh phúc trong ngôi nhà này thì chúng ta mới làm cho khách hàng của mình hạnh phúc được.
* Mỗi chúng ta là một cá thể riêng biệt, nhưng chúng ta cùng chung sống trong một ngôi nhà chung Viettel - ngôi nhà mà chúng ta cùng chung tay xây dựng. Đoàn kết và nhân hoà trong ngôi nhà ấy là tiền đề cho sự phát triển.

*Hành động:*

* Chúng ta tôn trọng nhau như những cá thể riêng biệt, nhạy cảm với các nhu cầu của nhân viên. Chúng ta lấy làm việc nhóm để phát triển các cá nhân. Các cá nhân, các đơn vị phối hợp với nhau như các bộ phận trong một cơ thể.
* Mỗi người chúng ta qua các thế hệ sẽ góp những viên gạch để xây lên ngôi nhà ấy.

Chúng ta lao động để xây dựng đất nước, Viettel phát triển, nhưng chúng ta phải được hưởng xứng đáng từ những thành quả lao động đó. Nhưng chúng ta luôn đặt lợi ích của đất nước của doanh nghiệp lên trên lợi ích cá nhân.

# CHUYÊN MÔN

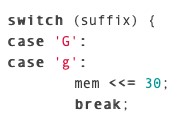
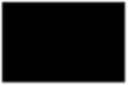
# Kiến thức lập trình

## Linux Kernel coding style

***Thụt đầu dòng***

Mỗi thụt đầu dòng có độ dài tương đương 8 ký tự. Một số lập trình viên thay đổi mặc định thành 4 ký tự cho mỗi thụt đầu dòng để các hàng code có thể lọt vào trong một trang mà không cần phải cuộn ngang để xem code. Tuy nhiên, sử dụng thụt đầu dòng với khoảng cách là 8 ký tự sẽ giúp chúng ta cố gắng giữ mức thụt đầu dòng tối đa là 3. Với thói quen này, các hàm sẽ tránh được hiện tượng deeply nested. Một thủ thuật để giảm thụt đầu dòng không cần thiết là đặt switch và case cùng 1 mức thụt đầu dòng (hình 4)

Hình : Một cách để giảm indentation level



Phong cách viết code kernel là đơn giản, tường minh, dễ hiểu, tránh đặt nhiều dòng lệnh trên cùng một hàng và thể hiện các mô tả theo hướng mẹo vặt.

***Cắt các dòng code quá dài***

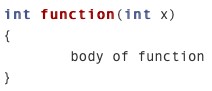
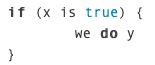
Các dòng code chỉ nên có độ dài từ 80 ký tự trở xuống, ngoại trừ dòng lệnh prink(). Chúng ta nên xuống dòng 1 cách khoa học nhằm mục đích làm cho code dễ nhìn hơn. Lưu ý nên mang dấu của biến theo cùng biến:

net\_profit = gross\_profit - overhead- cost\_of\_goods - payroll;

***Vị trí đặt dấu ngoặc nhọn và dấu cách***

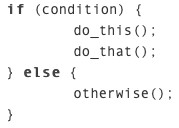
Dấu mở ngoặc nhọn nên đặt cùng hàng trừ các định nghĩa hàm (Hình 5) và block code chỉ có 1 lệnh duy nhất. Hình 5a thể hiện sai. Hình 5b thể hiện đúng.

Hình : Vị trí đặt dấu ngoặc nhọn



Nếu lệnh if else có đoạn 1 lệnh, đoạn nhiều lệnh thì tất cả phải dùng dấu ngoặc kếp (Hình 6).

Hình : Cách đặt dấu ngoặc nhọn nếu số lượng lệnh khác nhau



Dấu cách được sử dụng sau: if, switch, case, for, do, while

Dấu cách được sử dụng trước và sau: = + - < > \* / % | & ^ <= >= == != ? : Dấu ‘\*’ cho con trỏ được đặt sát tên biến: char \*ptr, char \*\*retptr. Dấu cách không được dùng phía sau: & \* + - ~ ! sizeof typeof alignof \_\_attribute\_\_ defined ++ —-. Không đặt dấu cách cuối mỗi dòng lệnh

Đặt tên

Tên hàm nên đặt dài và có khả năng thể hiện tốt ý nghĩa. Ví dụ:

count\_active\_users() thay vì: cntusr()

Tên biến cục bộ nên đặt ngắn gọn. Ví dụ: int cnt; or int c; thay vì: int counter; int idx; thay vì: int index;

Typedefs

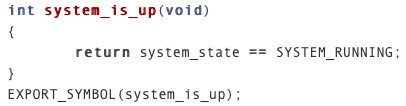
Chúng ta nên hạn chế sử dụng typedef trừ những trường hợp sau:

* Giấu object
* u8 u16 u32 u64
* Để test loại biến toàn hệ thống
* Để khai báo những biết thuộc về chuẩn C99 (Ví dụ: uint32\_t)
* Khai báo các biến được chia sẻ chung giữa những người dùng

Hàm

Độ dài của hàm nên vừa vặn trong 1 màn hình. Nếu dài hơn, chúng ta nên chia nhỏ thành từng module hoặc thay đổi cách code. Số lượng biến cục bộ trong 1 hàm nên ít hơn 10. Giữa các định nghĩa hàm nên để 1 dòng trống. EXPORT nên để phía dưới dấu đóng ngoặc nhọn (Hình 7)

Hình : Vị trí hàm EXPORT



Commenting

Chúng ta nên code sao cho nhìn vào code là có thể hiểu được code. Hạn chế phải comment vào trong code để giải thích cách code hoạt động như thế nào. Chúng ta chỉ nên giải thích ngay trước các hàm như cách giải thích của 1 API. Nếu chúng ta đang viết 1 kernel API, có một loạt các yêu cầu về cách viết document theo link sau: https://www.kernel.org/doc/html/ v4.10/doc-guide/sphinx.html

Làm gì nếu code đã không theo kernel style ngay từ đầu

GNU emacs là một công cụ giúp chúng ta tự động điều chỉnh định dạng code theo kernel style mong muốn. Chúng ta cod thể cấu hình file .emacs để điều chỉnh một số format mặc định không mong muốn.

Kconfig configuration files

Đối với file Kconfig, phía dưới config các thụt đầu dòng là 1 tab. Phía dưới help, thụt đầu dòng là 1 tab và 2 dấu cách:

config AUDIT

bool "Auditing support"

depends on NET

help

Enable auditing infrastructure that can be used with another

kernel subsystem, such as SELinux (which requires this for

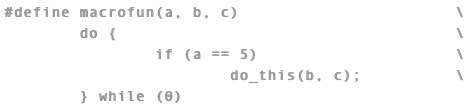
logging of avc messages output). Does not do system-call

auditing without CONFIG\_AUDITSYSCALL.

Macros, Enums and RTL

Tên đặt cho define và enum nên được viết hoa toàn bộ ngoại trừ #define func(). Nếu cần định nghĩa nhiều hằng số liên quan, chúng ta nên sử dụng enum. Nếu MACRO chứa nhiều dòng lệnh, chúng ta nên đặt nó vào trong một do-while block như Hình 8.

Hình : Do-while trong việc định nghĩa MACRO



In kernel messages

Các tin nhắn thông báo từ kernel nên ngắn gọn, tường mình, và dễ hiểu. Chúng ta nên sử dụng chẩn đoán model và driver MACRO từ linux/device.h để chọn được driver và device tương thích với nhau. Các tin nhắn cần được đặt đúng theo loại của nó: dev\_err(), dev\_warn(), dev\_info(), … Nếu tin nhắn không đi kèm cùng 1 thiết bị nào, sử dụng thư viện linux/printk.h với các định nghĩa pr\_notice(), pr\_info(), pr\_warn, pr\_err(). Các hàm pr\_XXX() in vô điều kiện thì hàm pr\_debug() lại không; Nó chỉ được compile khi DEBUG hoặc CONFIG\_DYNAMIC\_DEBUG được định nghĩa. Điều đó cũng đúng với dev\_dbg(), sử dụng VERBOSE\_DEBUG để thêm tin nhắn dev\_vdbg() đã được kích hoặc bởi DEBUG.

Một số subsystem có lựa chọn Kconfig debug để bật -DDEBUG trong Makefile tương ứng, trong một số trường hợp lại được định nghĩa: #define DEBUG. Trong một số trường hợp, khi tin nhắn debug đã được nằm sẵn trong #ifdef, hàm printk(KERN\_DEBUG ...) có thể được sử dụng.

Cấp phát bộ nhớ

Để cấp phát bộ nhớ trong kernel space, chúng ta sử dụng các hàm kmalloc(), kzalloc(), kmalloc\_array(), kcalloc(), vmalloc(), và vzalloc(). Hình 9 thể hiện cách truyển vào để cấp phát theo kích thước của 1 struct.

Hình : Truyền size của struct



Hình 10 thể hiện cách cấp phát một mảng trong kernel space (Nếu cần khởi tạo mảng với giá trị 0, chúng ta sử dụng hàm kcalloc()). Nếu cấp phát thất bại, hàm kmalloc()và Kcalloc() sẽ trả về NULL.

Hình : Cấp phát bộ nhớ trong kernel space



Mối tương quan giữa tên hàm và giá trị trả về

Các hàm có tên đặt theo hướng là câu hỏi hay kiểm tra thì nên trả về 0 nếu thất bại hoặc trả về một giá trị khác 0 nếu thành công. Các hàm có tên đặt theo hướng là một mệnh lệnh hoặc hành động làm gì đó thì nên trả về các giá trị tương đương với loại lỗi xảy ra bên trong hàm đó.

Đặc biệt tất cả các hàm được EXPORT cần tuân theo quy tắc này. Các hàm private (static) thì không cần. Trong trường hợp nếu hàm trả về một pointer, Khi lỗi xảy ra chúng ta có thể trả về NULL hoặc sử dụng cơ chế ERR\_PTR để thông báo lỗi.

Không nên định nghĩa lại MACRO

Các MACRO được định nghĩa trong include/linux/kernel.h không được thay đổi.

Editor modelines

Không nên đính kèm những marker configuration đặc biệt bên trong mã nguồn vì nó có thể không tương thích giữa các editor.

Conditional compilation

Chúng ta nên hạn chế sử dụng #if và #ifdef bởi vì các tiền xử lý này làm cho code trở nên rất khó đọc. Giải pháp là chúng ta nên sử dụng đính kèm có điều kiện trong các header files có chứa định nghĩa các hàm. Bằng cách này, trình biên dịch có thể tránh tạo ra code cho các stub calls.

## Tìm hiểu môi trường lập trình C trên Linux

## Sử dụng hệ điều hành Linux cho lập trình C

### GCC Compiler

GCC là từ viết tắt của GNU Compiler Collections. GCC là trình biên dịch phổ biến cho ngôn ngữ C/C++/Object C/Oject C++. GCC tương thích với nhiều loại CPU (ví dụ như ARM, Atmel, x86, MIPS, PowerPC, Intel, …) và hệ điều hành (Ví dụ như Linux, UNIX, MacOS, Symbian, Freescale Power Architecture, …). Quá trình biên dịch của gcc trải qua 4 giai đoạn:

Preprocessing: gcc –E \*.c –o \*.i

Compiling to assembly code: gcc –S \*.i –o \*.s

Compiling to machine code: as \*.s –o \*.o

Linking: ld \*.o -o \*

Bảng 4 cung cấp cho chúng ta một số cờ phổ biến khi sử dụng chương trình gcc.

Bảng 4: Cờ của gcc

|  |  |
| --- | --- |
| Flags | Functionality |
| -o opt | Cờ này sẽ biên dịch mã nguồn source.c như thường nhưng file thực thi sẽ có tên opt |
| -Werror | Cờ này sẽ yêu cầu trình biên dịch gcc in ra những cảnh báo trong quá trình biên dịch nếu có xuất hiện lỗi. |
| -Wall | Cờ này sẽ yêu cầu in cả lỗi lẫn cảnh báo như “unused varible”. Sử dụng cờ này thường xuyên là một thói quen tốt. |
| -ggdb3 | Biên dịch chương trình ở chế độ debug |
| -lm | Liên kết thư viện math.h vào file thực thi cuối cùng. Cờ -l dùng để link một thư viện nào đó. Để link thư viện math.h chúng ta sử dụng cờ -lm. |
| -std=c11 | Sử dụng các tiêu chuẩn của phiên bản c11 để biên dịch. |
| -c | Cờ này để biên dịch thành object files. Thường được sử dụng để viết thư viện. |
| -v | Cờ này yêu cầu chương trình được gắn in nhiều thông tin chi tiết hơn để sử dụng cho các quá trình debug. |

### GNU Debugger

GNU Project Debugger là một phần mềm hữu ích để debug ngôn ngữ C/C. Nó giúp chúng ta thực thi từng dọc lệnh một, đặt các break point, xem giá trị biến trên toàn bộ chương trình để giúp chúng ta hiểu được nguyên nhân thực sự khiến chương trình của chúng ta đang hoặc động sai. Bảng 5 là một số lệnh sử dụng trong chương trình gdb.

Bảng 5: GDB command

|  |  |
| --- | --- |
| Chạy chương trình | |
| gbd <program> [core dump] | Khởi chạy gdb (với option core dump) |
| gbd –args <program> <args…> | Khời chạy gdb và truyền tham số |
| gbd –pid <pid> | Khời chạy gdb với 1 process |
| Set args <args…> | Truyền tham số cho chương trình được debug |
| run | Chạy chương trình cần debug |
| kill | Kết thúc chương trình đang chạy |
| Tạo breakpoints | |
| break <where> | Chọn breakpoint |
| delete <breakpoint#> | Xóa breakpoint |
| clear | Xóa tất cả breakpoint |
| enable <breakpoint#> | Kích hoạt một breakpoint |
| disable <breakpoint#> | Tắt một breakpoint |
| Watchpoints | |
| watch <where> | Tạo watchpoint |
| delete/enable/disable <watchpoint#> | tương tự breakpoint |
| <where> | |
| function\_name | Tên chương trình cần watch/break |
| line\_number | Dòng code trong chương trình hiện tại cần watch/break |
| file:line\_number | Dòng code trong một mã nguồn cụ thể cần watch/break |
| Điều kiện | |
| break/watch <where> if <condition> | Break/watch nếu thỏa mãn điều kiện |
| condition <breakpoint#> <condition> | Cài đặt hoặc thay đổi điều kiện của 1 breakpoint |
| Khảo sát stack | |
| backtrace  where | Hiện call stack |
| backtrace full  where full | Hiện call stack và tất cả biến cục bộ trong cùng frame |
| frame <frame#> | Chọn stack frame |
| Stepping | |
| step | Thực thi dòng lệnh tiếp thep và nhảy vào bên trong hàm |
| next | Thực thi dòng lệnh tiếp theo nhưng không nhảy vào bên trong hàm |
| finish | Thực thi đến return tiếp theo |
| continue | Tiếp tục thực thi |
| Biến và vùng nhớ | |
| print/format <what> | In giá trị của biến/vùng nhớ/thanh ghi |
| display/format <what> | In sau khi thực hiện dòng lệnh |
| undisplay <display#> | Loại bỏ tag display |
| enable display <display#>  disable display <display#> | Tắt hoặc bật một tag display nào đó |
| x/nfu <address> | In một vùng nhớ  n: Số lượng đơn vị cần in (Mặc định 1).  f: Định dạng cần in  u: Đơn vị  Đơn vị:  b: Byte,  h: Half-word (2 bytes)  w: Word (4 bytes)  g: Giant word (8 bytes)  Định dạng:  a Pointer.  c đọc kiểu integer, in kiểu character.  d Integer, signed decimal.  f Floating point number.  o Integer, print as octal.  s Đọc kiểu C string.  t Đọc kiểu integer, in kiểu binary  u Integer, unsigned decimal.  x Integer, in kiểu hexadecimal. |
| <What> | |
| expression | C expression hoặc function call (cần type casting để gdb biết kiểu trả về của hàm) |
| file\_name::variable\_name | In biến static |
| function::variable\_name | In biến của hàm |
| {type}address | In giá trị tại một địa chỉ nhớ |
| $register | In giá trị của thanh ghi. Một số thanh ghi có ích: $esp (stack pointer), $ebp (frame pointer) và $eip (instruction pointer). |
| Thread | |
| thread <thread#> | Chọn thread để chạy |
| set var <variable\_name>=<value> | Thay đổi giá trị của một biến bất kỳ |
| return <expression> | By pass function, ép buộc trả về một giá trị |
| directory <directory> | Thêm search path cho việc tìm kiếm mã nguồn |
| list  list <filename>:<funttion>  list <filename>:<line\_number>  list <first>,<last> | In context của mã nguồn |
| set listsize <count> | Giới hạn dòng in của list |
| Signals | |
| Handle <signal> <options> | Thiết lập các tùy chọn để khảo sát signals. Các tùy chọn bao gồm:  (no)print: (Donʻt) In thông báo khi có signal.  (no)stop: (Donʻt) Dừng chương trình khi có signal.  (no)pass: (Donʻt) Truyền signal cho chương trình. |
| Thông tin chi tiết | |
| disassemble  disassemble <where> | Remove một hàm hoặc vị trí nào đó |
| info args | In tham số được truyền vào hàm trong stack frame hiện tại |
| info breakpoints | In thông tin về break- và watchpoints. |
| info display | In thông tin về display |
| info locals | In các biến cục bộ của stack frame hiện tại |
| info sharedlibrary | In các shared library đã được load vào |
| info signals | In tất cả signal và cách chúng được handle |
| info threads | In tất cả thread |
| show directories | In tất cả search path |
| show listsize | Số lượng được hiển thị trong lệnh list |
| whatis variable\_name | In kiểu dữ liệu của biến |

### Makefile và “make” system

Makefile là một công cụ giúp đơn giản hóa quá trình biên dịch mã nguồn. Đặc biệt là khi chúng ta có quá nhiều tập tin mã nguồn dẫn đền việc mỗi lần có sự thay đổi cần biên dịch lại, chúng ta cần phải nhập lại lệnh biên dịch rất dài.

Rules:

target [target...] : [dependent ....]

[ command ...]

* make all – Biên dịch lại toàn bộ mã nguồn
* make install – Sao chép file thực thi vào filesystem
* make clean – Xóa tất cả các file object, file tạm và file thực thi.

Implicit rules:

.c:

$(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@

Quy tắc này quy định cách biên dịch file thực thi x từ x.c. Quy tắc này là ngầm bởi vì chúng ta chưa đề cập đến bất kì target nào. Quy tắc tắc được sử dụng trong mọi trường hợp. Ví du:

.c.o:

$(CC) $(CFLAGS) -c $<

Hoặc

.c.o:

$(CC) $(CFLAGS) -c $\*.c

Macros:

Macros trong makefile tương tự cơ chế substitute trong C. Ví dụ:

MACROS = -me

PSROFF = groff -Tps

Các macros đặc biệt:

* $@ Tên của file được tạo ra.
* $? Tên của các phụ thuộc được thay đổi.
* $< Tên của các file liên quan đang thực hiện lệnh.
* $\* Tiền tố chung giữa target và dependent files.

Ví dụ:

hello: main.cpp hello.cpp factorial.cpp

$(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@

Trong đó: @ = hello

Dependencies:

hello: main.o factorial.o hello.o

$(CC) main.o factorial.o hello.o -o hello

=>

main.o: main.cpp functions.h

$(CC) -c main.cpp

factorial.o: factorial.cpp functions.h

$(CC) -c factorial.cpp

hello.o: hello.cpp functions.h

$(CC) -c hello.cpp

Directive:

* Conditional
* Include
* Override

Conditional directive:

conditional-directive

text-if-true

else

text-if-false

endif

libs\_for\_gcc = -lgnu

normal\_libs =

foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs\_for\_gcc)

else

$(CC) -o foo $(objects) $(normal\_libs)

endif

Bảng 6: Điều kiện trong make

|  |  |
| --- | --- |
| ifeq | Macro này hoạt động tương tự strcmp() trong C |
| ifneq | Macro này hoặc động ngược lại với ifeq |
| ifdef | Kiểm tra nếu một macro nào đó đã được khai báo hay chưa. Nếu rồi thì phần lệnh tiếp theo sẽ được thực thi |
| ifndef | Kiểm tra xem một macro nào đó đã được khai báo hay chưa. Nếu chưa thì phần lệnh tiếp theo sẽ được thực thi |
| else | Nếu điều kiện if phía trước không thỏa thì phần lệnh sau else sẽ được thực thi |
| endif | Đánh dấu kết thúc của phần lệnh của ifeq ifneq ifdef ifndef else |

Include directive:

include filenames...

include foo \*.mk $(bar)

Tương đương với:

include foo a.mk b.mk c.mk bish bash

override directive

override variable = value

Hoặc:

override variable := value

Recompilation:

Công cụ make có khả năng xác định được những thay đổi trong mã nguồn dẫn tới việc mỗi lần cần biên dịch lại, nó chỉ cần biên dịch những file cần thiết và liên quan tới quá trình thay đổi, giúp tiết kiệm thời gian cho lập trình viên.

## Ngôn ngữ lập trình C

### Data types, Operations, Expression

Bảng 7: Phân loại kiểu dữ liệu trong C

|  |  |
| --- | --- |
| STT | Kiểu dữ liệu và mô tả |
| 1 | Kiểu cơ bản  Kiểu dữ liệu trong C được chi ra 2 loại cơ bản đó là số nguyên và dấu chấm động. |
| 2 | Kiểu enum  Kiểu dữ liệu enum chỉ cho phép gán một vài số nguyên đã được xác định trước. |
| 3 | Kiểu void  Kiểu dữ liệu void là kiểu dữ liệu không có giá trị. |
| 4 | Kiểu derive  Kiểu dữ liệu này bao gồm: Con trỏ, mảng, struct, union, function. |

Bảng 8: Kiểu nguyên

|  |  |  |
| --- | --- | --- |
| Kiểu | Kích thước | Khoảng giá trị |
| char | 1 byte | -128 tới 127 or 0 tới 255 |
| unsigned char | 1 byte | 0 tới 255 |
| signed char | 1 byte | -128 tới 127 |
| int | 2 hoặc 4 bytes | -32,768 tới 32,767 or -2,147,483,648 tới 2,147,483,647 |
| unsigned int | 2 hoặc 4 bytes | 0 tới 65,535 or 0 tới 4,294,967,295 |
| short | 2 bytes | -32,768 tới 32,767 |
| unsigned short | 2 bytes | 0 tới 65,535 |
| long | 8 bytes | -9223372036854775808 tới 9223372036854775807 |
| unsigned long | 8 bytes | 0 tới 18446744073709551615 |

Bảng 9: Kiểu chấm động

|  |  |  |  |
| --- | --- | --- | --- |
| Kiểu | Kích thước | Khoảng giá trị | Độ chính xác |
| float | 4 byte | 1.2E-38 tới 3.4E+38 | 6 số nguyên sau dấu phẩy |
| double | 8 byte | 2.3E-308 tới 1.7E+308 | 15 số nguyên sau dấu phẩy |
| long double | 10 byte | 3.4E-4932 tới 1.1E+4932 | 19 số nguyên sau dấu phẩy |

Bảng 10: Các kiểu trả về của hàm

|  |  |
| --- | --- |
| STT | Mô tả |
| 1 | Trả về kiểu void  Hàm trả về kiểu void là hàm không trả về bất kì giá trị nào. |
| 2 | Tham số kiểu void  Hàm nhận tham số kiểu voi là hàm không nhận bất kì tham số nào. |
| 3 | Con trỏ tới void  Kiểu con trỏ tới void thể hiện địa chỉ của một object chứ không phải là một kiểu dữ liệu. Ví dụ, khi chúng ta khai báo một hàm void \*malloc( size\_t size );, hàm này trả về con trỏ tới void. Nhờ vậy chúng ta có thể cast qua bất kì kiểu dữ liệu nào. |

Bảng 11: Chuỗi escape của C

|  |  |
| --- | --- |
| Chuỗi escape | Ý nghĩa |
| \a | Còi |
| \b | Dấu cách |
| \f | Form Feed |
| \n | Xuống dòng |
| \r | Carriage Return |
| \t | Tab |
| \v | Vertical Tab |
| \\ | Dấu gạch chéo ngược |
| \' | Dấu kép đơn |
| \" | Dấu kép |
| \? | Dấu hỏi |
| \nnn | Số octa |
| \xhh | Số hexa |
| \0 | Null |

Bảng 12: Toán tử số học

|  |  |
| --- | --- |
| Operator | Meaning of Operator |
| + | Cộng |
| - | Trừ |
| \* | Nhân |
| / | Chia |
| % | Chia lấy dư |

Toán tử tăng giảm: ++: Toán tử tăng 1 đơn vị; --: Toán tử giảm 1 đơn vị

Các kiểu toán tử tăng giảm:

* Tăng trước khi thực hiện lệnh: ++var;
* Giảm trước khi thực hiện lệnh: var++;

Bảng 13: Toán tử gán

|  |  |  |
| --- | --- | --- |
| Toán tử | Ví dụ | Cách tương đương |
| = | a = b | a = b |
| += | a += b | a = a+b |
| -= | a -= b | a = a-b |
| \*= | a \*= b | a = a\*b |
| /= | a /= b | a = a/b |
| %= | a %= b | a = a%b |

Bảng 14: Toán tử so sánh

|  |  |  |
| --- | --- | --- |
| Toán tử | Ý nghĩa | Ví dụ |
| .== | Bằng | 5 == 3 trả về 0 |
| > | Lớn hơn | 5 > 3 trả về 1 |
| < | bé hơn | 5 < 3 trả về 0 |
| != | Khác | 5 != 3 trả về 1 |
| >= | Lớn hơn hoặc bằng | 5 >= 3 trả về 1 |
| <= | Bé hơn hoặc bằng | 5 <= 3 trả về 0 |

Bảng 15: Toán tử logic

|  |  |  |
| --- | --- | --- |
| Toán tử | Ý nghĩa | Ví dụ |
| && | AND | c = 5, d = 2 thì ((c == 5) && (d > 5)) bằng 0. |
| || | OR | c = 5, d = 2 thì ((c == 5) || (d > 5)) bằng 1. |
| ! | NOT | c = 5 thì (c == 5) bằng 0. |

Bảng 16: Toán tử bitwise

|  |  |
| --- | --- |
| Toán tử | Ý nghĩa |
| & | AND |
| | | OR |
| ^ | XOR |
| ~ | NOT |
| << | SHIFT LEFT |
| >> | SHIFT RIGHT |

Bảng 17: Các toán tử khác

|  |  |
| --- | --- |
| Toán tử | Ý nghĩa |
| , | Ngắn cách các expression liên quan. |
| <conditional expression> ? <expression1> : <expressiont2> | Toán tử điều kiện nếu thỏa thì gán với expression 1, nếu không thì expression 2 |
| (<type>) <expression> | Chuyển đổi kiểu dữ liệu. |

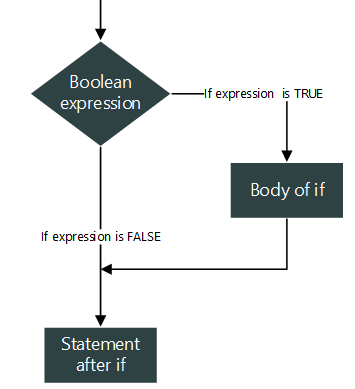
### Control Flow, Control Statement

Có 3 loại lệnh điều khiển:

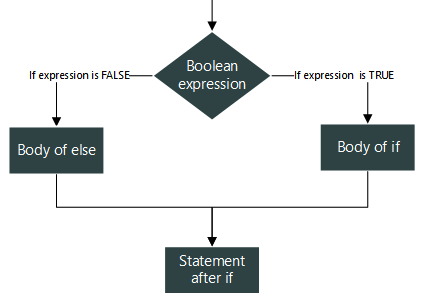
* Decision making statements
* Looping statements
* Jump statements

Decision making statements:

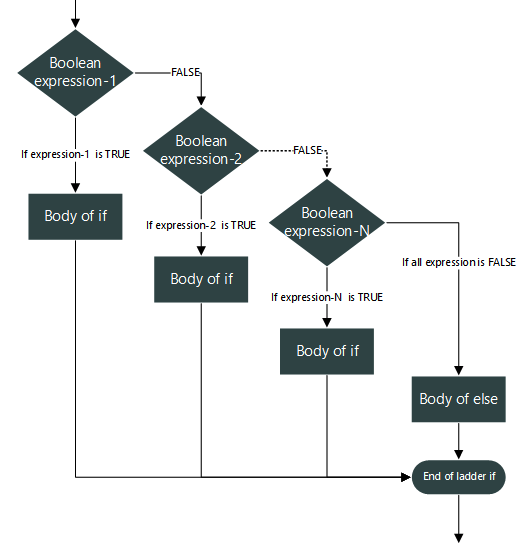
Loại này sẽ ra quyết định dựa trên việc xem xét một điều kiện là đúng hay sai (Hình 11)

**if**

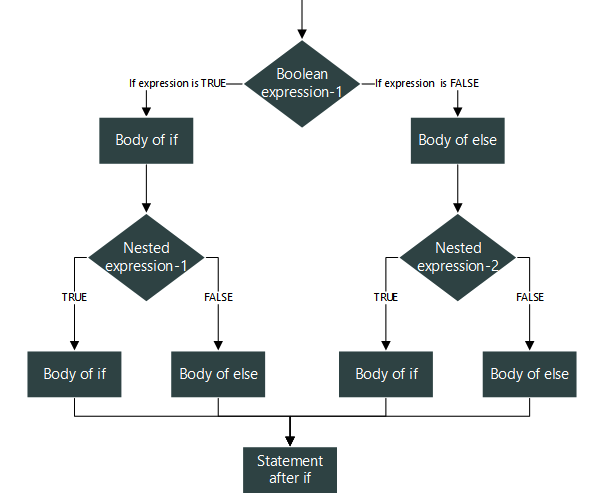
Hình : Điều kiện if

If else (Hình 12)

Hình : Sơ đồ khối điều kiện if else

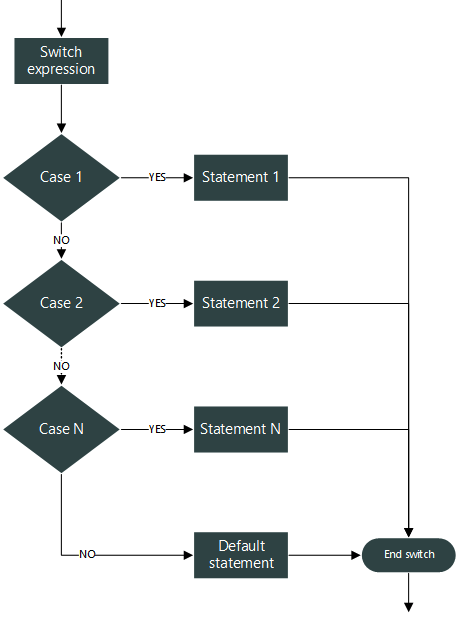
Nhiều vòng if else lồng vào nhau (Hình 13)

Hình : Sơ đồ khối điều kiện if else liên tiếp



Hình : Sơ đồ khối điều kiện if else lồng vào nhau

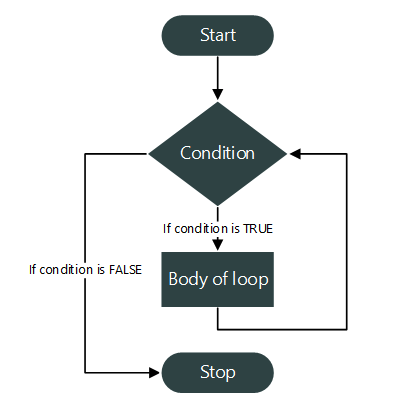
switch case (Hình 15)



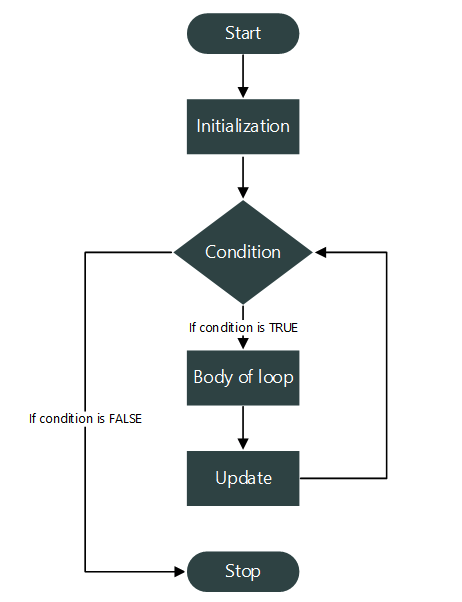
Hình : Sơ đồ khối câu lệnh switch

Looping statement:

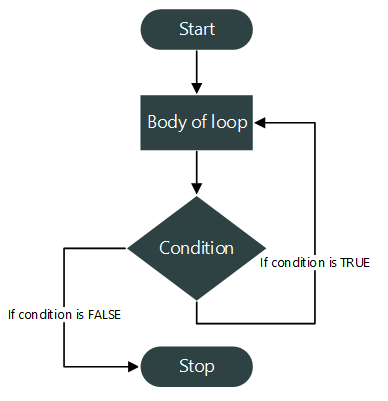
Loại này thường được gọi là vòng lặp. Trong ngôn ngữ C có 3 loại vòng lặp chính:

Vòng lặp for (Hình 16)

Hình : Sơ đồ khối vòng lặp for

Vòng lặp while (Hình 17)

Hình : Sơ đồ khối vòng lặp while

Vòng lặp do while (Hình 18)

Hình : Sơ đồ khối vòng lặp do while

Jump statements:

Đây là lệnh điểu khiển vô điều kiện. Lệnh này có 3 loại:

* break
* continue
* goto

### Function and Program Structure

<function declaration>

<main>

<function definition>

return\_type function\_name( parameter list );

return\_type function\_name( parameter list ) {

body of the function

}

Một hàm cơ bản phải có ít nhất hai phần. Đó là thân hàm và tên hàm. Dưới đây là tất cả các thành tố của một hàm:

* Kiểu trả về: Một hàm có thể trả về một giá trị, một con trỏ hoặc không trả về gì cả.
* Tên hàm: Đây là tên sẽ được gọi khi chúng ta cần thực hiện hàm. Tên của hàm không được trùng với các hàm khác.
* Tham số: Một hàm có thể có tham số hoặc không, các tham số này rất hữu ích trong việc đưa dữ liệu và thông số cho hàm xử lý.
* Thân hàm: Thân hàm là tập hợp các dòng lệnh mà hàm sẽ thực hiện.

Bảng 18: Các cách truyền tham số

|  |  |
| --- | --- |
| STT | Khái niệm và ý nghĩa |
| 1 | Truyền bằng giá trị  Phương pháp này tạo ra một bản copy giá trị truyền vào bên trong hàm được gọi, khi hàm kết thúc thì giá trị này cũng được xóa đi. |
| 2 | Truyền bằng địa chỉ  Phương pháp này tạo một bản copy địa chỉ của giá trị cần truyền. Do đó khi hàm thực thi, giá trị có thể sẽ được thay đổi trên chính vùng nhớ gốc và được giữ nguyên sau khi hàm kết thúc. Chỉ có địa chỉ copy vào là được giải phóng. |

Phạm vi trong lập trình là một phân đoạn code nơi mà các biến đã khai báo được hiện hữu. Ra khỏi vùng hiện hữu này thì các biến này không thể được nhìn thấy hay truy xuất. Dựa vào phạm vi này, biến trong ngôn ngữ lập trình C được chia làm 3 loại: Biến cục bộ, biến toàn cục và Tham số hàm.

Bảng 19: Giá trị khởi tạo mặc định trong C

|  |  |
| --- | --- |
| Kiểu dữ liệu | Giá trị mặc định |
| int | 0 |
| char | \0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

Tiền xử lý không phải là một phần của trình biên dịch nhưng nó là một công đoạn cần thiết trong quá trình biên dịch. Tiền xử lý về cơ bản sẽ tạo ra một bản mã nguồn khác chỉ chứa các mã nguồn cần thiết để biên dịch.

Bảng 20: Tiền xử lý trong C

|  |  |
| --- | --- |
| STT | Mô tả |
| 1 | #define Thay thế các macros được khai báo bởi giá trị của nó. |
| 2 | #include sao chép các mã nguồn trong các tập tin được include. |
| 3 | #undef Xóa một macro đã được định nghĩa. |
| 4 | #ifdef Kiểm tra xem một macro nào đó đã được định nghĩa hay chưa. |
| 5 | #ifndef Kiểm tra xem nếu một macro nào đó chưa được định nghĩa. |
| 6 | #if Kiểm tra một điều kiện để thực hiện compile phần mã nguồn sau đó. |
| 7 | #else rẽ nhánh cho điều kiện #if. |
| 8 | #elif thay thế cho #else và #if trên cùng 1 dòng. |
| 9 | #endif đánh dấu kết thúc phần mã nguồn cần compile cho các điều kiện #ifdef #ifndef #if #else #endif. |
| 10 | #error In dữ liệu và file descripter 2. |
| 11 | #pragma Dự trữ cho các compiler để tự định nghĩa tiền xử lý riêng. |

### Con trỏ cho các

Con trỏ là một kiểu dữ liệu cho phép chúng ta lưu địa chỉ của một dữ liệu nào đó.

Mảng là một kiểu dữ liệu cho phép chúng ta lưu trữ trong một chuỗi các ô nhớ có độ dài xác định có cùng kiểu dữ liệu. Các thao tác liên quan tới mảng: Khai báo mảng, Khởi tạo mảng và Truy xuất mảng.

Bảng 21: Các khái niệm liên quan tới mảng

|  |  |
| --- | --- |
| STT | Khái niệm và mô tả |
| 1 | Mảng đa chiều  Ngôn ngữ C cung cấp kiểu dữ liệu mảng đa chiều xây dựng dựa trên mảng một chiều. Hình thái đơn giản nhất của mảng đa chiều là một ma trận 2 chiều. |
| 2 | Truyền mảng cho hàm  Chúng ta có thể truyền mảng vào hàm bằng cách truyền con trỏ trỏ tới phần tử đầu tiên của mảng đó. |
| 3 | Hàm trả về mảng  Trong ngôn ngữ C, chúng ta có thể trả về mảng bằng cách trả về con trỏ trỏ tới phần tử đầu tiên của mảng đó. |
| 4 | Con trỏ tới mảng  Con trỏ tới mảng trong C chính là con trỏ trỏ tới phần tử đầu tiên của mảng hay nói cách khác, nó chỉnh là tên của mảng. |

Bảng 22: Thao tác với con trỏ

|  |  |
| --- | --- |
| STT | Khái niệm và mô tả |
| 1 | Toán tử mảng: ++, --, +, - |
| 2 | Mảng của con trỏ là mảng một chiều mà trong đó, mỗi phần tử chính là một con trỏ. Đây chính là tiền để để tạo ra mảng đa chiều trong ngôn ngữ C. |
| 3 | C cho phép chúng ta tạo con trỏ trỏ tới một con trỏ khác. Tên của một mảng đa chiều cũng thuộc trường hợp này. |
| 4 | Truyền con trỏ vào hàm chính là truyền địa chỉ của biến vào hàm. Khi đó hàm sẽ thao tác trực tiếp trên vùng nhớ của biến đó thay vì sao chép biến thành một biến tạm thời khác. |
| 5 | Hàm có thể thực hiện trả về con trỏ tương tự như việc trả về mảng. |

### Advanced Data type

Structure:

Struct là một kiểu dữ liệu cho người dùng tự định nghĩa, kết hợp sử dụng các kiểu dữ liệu có sẵn trong C.

Cú pháp:

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

Truy cập các phần tử của struct, chúng ta sử dụng toán tử (.). Để truyền struct như một tham số vào hàm, chúng ta sử dụng struct như một biến con trỏ. Con trỏ đến struct cũng giống như con trỏ của con trỏ.

Bit Fields:

Bit Fields cho phép đóng gói dữ liệu trong một struct. Chức năng này đặc biệt hữu dụng khi chương trình của chúng ta phải chạy trong một môi trường có bộ nhớ giới hạn. Ví dụ: Chúng ta ghép 1-bit flag và 7-bit integer thành một số 8-bit integer. Ngôn ngữ C cung cấp cho chúng ta tính năng này đối với struct bằng cách thêm :<bit#> đằng sau tên mỗi biến khi khai báo struct. Trong đó <bit#> là lượng bit cần dùng để lưu trữ biến trước nó. Ví dụ:

struct packed\_struct {

unsigned int f1:1;

unsigned int f2:1;

unsigned int f3:1;

unsigned int f4:1;

unsigned int type:4;

unsigned int my\_int:9;

} pack;

C – Unions:

Union là một kiểu dữ liệu đặc biệt trong ngôn ngữ C cho phép lưu nhiều kiểu dữ liệu khác nhau trên cùng một vùng nhớ.

Cú pháp:

union [union tag] {

member definition;

member definition;

...

member definition;

} [one or more union variables];

Để truy cập một phần tử trong union, chúng ta cần sử dụng toán tử (.).

Enum:

Enum là một tập hợp các hằng số có mối liên hệ với nhau. Cú pháp:

enum flag { const1, const2, ..., constN };

## Data structure

### Elementary Data Structure

Linked list là một kiểu dữ liệu tuyến tính. Các phần tử không được lưu thành một dải địa chỉ ảo liên tục. Mỗi phần tử sẽ dành ra bộ nhớ để lưu địa chỉ của phần tử tiếp theo (Hình).

Các thành tố cơ bản:

* Link – Chứa địa chỉ của một phần tử trong list.
* Next – trỏ tới link của phần tử tiếp theo.
* Data – Dữ liệu của một phần tử

Các operation cơ bản:

* Insertion – Thêm một phẩn tử ở vị trí bất kì trong list
* Deletion − Xóa một phẩn tử ở vị trí bất kì trong list
* Display – in toàn bộ list
* Search – Tìm kiếm một phần tử trong list sử dụng một giá trị key
* Delete – Xóa một phần tử trong list sử dụng một giá trị key

Bảng 23: Linked list vs Array pros and cons

|  |  |  |
| --- | --- | --- |
|  | Ưu điểm | Nhược điểm |
| Mảng | * Dễ sử dụng * Có thể truy cập ngẫu nhiên | * Không thể thay đổi kích thước * Cấp phát theo block * Rất khó để chèn và xóa |
| Linked list | * Dễ dàng chèn và xóa * Bộ nhớ có thể phân mảnh * Cấp phát động | * Phải truy cập tuần tự * Thời gian truy cập phụ thuộc kích thước list và vị trí node * Tốn bộ nhớ để lưu địa chỉ |

Linked list có thể phân loại thành 3 nhóm: Singly, Doubly và Circular.

Singly:

Doubly:

Circular:

### Abstract Data Types

## Sorting algorithm

Phân loại giải thuật:

* In-place sorting và not-in-place sorting
* Stable sorting và unstable sorting
* Adaptive và non-adaptive sorting

Trong đó, in-place sorting không yêu cầu một vùng nhớ khác mà thực hiện sắp xếp trên chính vùng nhớ hiện tại của dữ liệu. Stable sorting vẫn giữ thứ tự của những elements có cùng comparing key. Adaptive sorting là giải thuật sắp xếp có độ phức tạp không thay đổi trong mọi trường hợp.

Bảng 24: So sánh các giải thuật sắp xếp

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Giải thuật | In-place | Tính ổn định | Adap-  tibility | TH Tốt | TH thông thường | TH xấu | Lưu ý |
| selection sort | ✔ |  | ✔ | ½ *n* 2 | ½ *n* 2 | ½ *n* 2 | Khi n thay đổi, trường hợp tốt nhất vẫn đạt độ phức tạp bậc 2 |
| insertion sort | ✔ | ✔ |  | *n* | ¼ *n* 2 | ½ *n* 2 | Chỉ nên sử dụng cho mảng có kích thước nhỏ hoặc mảng đã được sắp xếp một phần |
| bubble sort | ✔ | ✔ |  | *n* | ½ *n* 2 | ½ *n* 2 | Ít khi được sử dụng, nên dùng selection sort hơn bubble sort |
| mergesort |  | ✔ | ✔ | ½ *n* lg *n* | *n* lg *n* | *n* lg *n* | Giải thuật này đảm bảo luôn dưới *n* log *n* |
| quicksort | ✔ |  |  | *n* lg *n* | 2 *n* ln *n* | ½ *n* 2 | Tỉ lệ cao *n* log *n,* Trong thực tế là nhanh nhất |
| heapsort | ✔ |  |  | *n* | 2 *n* lg *n* | 2 *n* lg *n* | *n* log *n* và in-place |

## C programming in Linux

### Process và Thread

#### Process

Process là một không gian địa chỉ nơi mà một thread hoặc hơn được thực thi, kèm theo đó là những tài nguyên cần thiết cho việc thực thi.

Linux có một kiểu dữ liệu gọi là bảng process. Bảng này mô tả tất và các process đang được load vào RAM cùng với một số thông tin ví dụ như PID, owner, độ ưu tiên, biến môi trường, trạng thái, ID của process cha, con trỏ tới vị trí chưa phần binary thực thi của process. Ví dụ:

$ ps -ef

UID PID PPID C STIME TTY TIME CMD

root 433 425 0 18:12 tty1 00:00:00 [bash]

root 445 426 0 18:12 tty2 00:00:00 -bash

root 456 427 0 18:12 tty3 00:00:00 [bash]

root 467 433 0 18:12 tty1 00:00:00 sh /usr/X11R6/bin/startx

root 474 467 0 18:12 tty1 00:00:00 xinit /etc/X11/xinit/xinitrc –

$ ps ax

PID TTY STAT TIME COMMAND

1 ? Ss 0:03 init [5]

2 ? S 0:00 [migration/0]

3 ? SN 0:00 [ksoftirqd/0]

4 ? S< 0:05 [events/0]

Trong đó, trạng thái của process là thông tin rất quan trọng được biểu diễn bằng code như trong Bảng 25.

Bảng 25: Các trạng thái của process

|  |  |
| --- | --- |
| STATE code | Mô tả |
| S | Sleeping. Process thông thường sẽ đợi chờ một signal hay một input nào đó. |
| R | Running. Đang chạy hoặc chờ trong hàng đợi để chạy. |
| D | Uninterruptible Sleep (Chờ). Thông thường là chờ input/output để kết thúc. |
| T | Stopped. Được dừng bởi shell hoặc debugger. |
| Z | Zombie process. |
| N | “Nice”. Độ ưu tiên thấp |
| S | Process là session leader. |
| + | Process thuộc nhóm foreground (được chạy và chờ hoàn thành). |
| L | Process chạy ở chế độ đa luồng. |
| < | Độ ưu tiên cao. |

Linux kernel có thể chạy trên các vi xử lý đơn nhân và đa nhân. Do đó để chạy nhiều process cùng một lúc, Linux Kernel cần quá trình lập lịch để quyết định xem process nào sẽ được nhận khung thời gian tiếp theo. Để đưa ra quyết định này, Linux Kernel dựa trên thông số độ ưu tiên của process. Process có độ ưu tiên cao hơn thì sẽ được nhận khung thời gian chạy thường xuyên hơn và ngược lại. Độ ưu tiên của process được tính dựa vào giá trị “nice” (mặc định là 0) và hành vi của nó.

Chúng ta có thể tạo một process con mới sử dụng hàm system, exec và fork.

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

Khi một process con được tạo ra, nó sẽ được chạy độc lập với process cha. Để kiểm tra xem process con đã xong chưa thì chúng ta sử dụng hàm wait.

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*stat\_loc);

Hàm wait buộc process cha phải tạm dừng cho tới khi process con kết thúc. Hàm này trả về PID của process con. Để hiểu thêm thông tin về quá trình kết thúc của process con, chúng ta sử dụng các macro trong Bảng 26 để kiểm tra biến stat\_loc.

Bảng 26: Kiểm tra quá trình kết thúc của process con thông qua biến stat\_loc

|  |  |
| --- | --- |
| Macro | Giá trị và ý nghĩa |
| WIFEXITED(stat\_val) | Khác 0 nếu process con kết thúc bình thường. |
| WEXITSTATUS(stat\_val) | Nếu WIFEXITED khác 0, macro này sẽ là exit code của process con. |
| WIFSIGNALED(stat\_val) | Khác 0 nếu process con kết thúc với signal chưa được handle. |
| WTERMSIG(stat\_val) | Nếu WIFSIGNALED khác 0, macro này bằng với giá trị signal chưa được handle. |
| WIFSTOPPED(stat\_val) | Bằng 0 nếu process con đã kết thúc. |
| WSTOPSIG(stat\_val) | Nếu WIFSTOPPED khác không, macro này sẽ bằng với giá trị của signal. |

Một điều quan trọng khi sử dụng hàm fork là chúng ta phải kiểm soát tất cả các process con. Nếu một process con đã kết thúc nhưng vẫn chưa được gọi hàm wait thì nó vẫn tổn tại trong bảng process và trở thành zoombie process.

Để thay thế cho hàm wait, chúng ta có thể sử dụng hàm waitpid để chờ một process con cụ thể nào đó.

pid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);

Với pid là là PID của process con chúng ta cần chờ. Hàm này trả về error-code integer. Tham số option giúp chúng ta điều chỉnh hành vi của hàm waitpid. Ví dụ, WNOHANG sẽ ngăn việc gọi hàm waitpid làm gián đoạn quá trình thực thi của process. Chúng ta có thể dùng nó để kiểm tra xem một process con đã kết thúc hay chưa mà không cần phải dừng lại.

#### Signal

Hiện tại phương pháp tạo và nhận signals sử dụng syscall signal và friends đã lỗi thời. Các phiên bản linux mới hơn sử dụng hàm sigaction:

int sigaction(int sig, const struct sigaction \*act, struct sigaction \*oact);

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

Struct sigaction dùng để định nghĩa một số thao tác trong việc nhận signal sig. Trong đó:

void (\*) (int) sa\_handler /\* Handler tương ứng với signal sig

sigset\_t sa\_mask /\* Mô tả một nhóm các signals sẽ được thêm vào process signal mask trước khi signal handler trong sa\_handler được gọi. Nhóm này sẽ bị block và không thể gửi được đến process. Điều này giúp chúng ta tránh được tình huống signal được nhận trước khi handler của nó được thực thi xong. Nói cách khác, sa\_mask giúp tránh hiện tượng race condition đã được để cập ở trên.

int sa\_flags /\* Tuy nhiên, các signals handler được set bởi sigaction không phải là handler gốc của nó. Để sử dụng lại handler gốc của một signal nào đó bởi hàm sigaction thì chúng ta cần set sa\_flags qua SA\_RESETHAND. Khi đó hàm sigaction sẽ bypass sa\_hander và sử dụng handler gốc của sig được truyền vào.

Hàm sigaction thực hiện các lệnh liên quan tới signal sig. Nếu struct oact khác NULL, hàm này sẽ ghi signal action trước đó vào vị trí phù hợp. Nếu act khác NULL, sa\_handler sẽ được set.

Kiểu dữ liệu sigset\_t được định nghĩa trong thư viện signal.h. Kiểu dữ liệu này được sử dụng trong hàm sigaction và một số hàm khác để điều chỉnh cách process nhận các signals.

Các hàm để thao tác với biến sa\_mask có kiểu dữ liệu sigset\_t:

int sigaddset(sigset\_t \*set, int signo); // Thêm signal vào set

int sigemptyset(sigset\_t \*set); // Khởi tạo signal set trống

int sigfillset(sigset\_t \*set); // Khởi tạo signal set chứa tất cả các signal đã được khai báo

int sigdelset(sigset\_t \*set, int signo); // Xóa signal khỏi set

Các hàm này trả về error-code integer

int sigismember(sigset\_t \*set, int signo); // Kiểm tra xem một signal có trong set hay không.

Hàm này trả về succeeded boolean.

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset); // Hàm này thay đổi process signal mask theo nhiều cách dựa vào tham số how:

* SIG\_BLOCK: Các signals trong set sẽ được thêm vào signal mask
* SIG\_SETMASK: Signal mask sẽ được lấy từ set
* SIG\_UNBLOCK: Các signal trong set sẽ được loại bỏ khỏi signal mask

Trong trường hợp set được truyền vào là NULL thì how sẽ không được sử dụng. Lúc này, hàm sigprocmask chỉ thực hiện copy tất cả giá trị trong signal mask của process hiện tại vào oset.

Hàm sigprocmask trả về error-code integer. Nếu một signal bị chặn, nó sẽ không đến được process và ở trạng thái pending. Để kiểm tra xem những signal nào trong signal mask đã được gửi tới process (và đang ở trong trạng thái pending), chúng ta sử dụng hàm sigpending để ghi các signal này vào set.

int sigpending(sigset\_t \*set);

Chúng ta có thể tạm dừng việc thực thi process cho đến khi nhận được một signal nào đó bằng cách dùng hàm sigsuspend:

int sigsuspend(const sigset\_t \*sigmask);

Hàm sigsuspend tạm thời thay thế signal mask của process bằng set sigmask và tạm dừng process. Process sẽ được tiếp tục khi signal handler được chạy hoặc khi nhận được signal yêu cầu kết thúc process. Sau khi signal handler chạy, signal mask gốc sẽ được khôi phục về trạng thái trước khi gọi hàm sigsuspend. Nếu chúng ta thêm SIGKILL hoặc SIGSTOP vào sigmask thì cả 2 vẫn không thể bị block được. Hàm này luôn trả về -1.

Thuộc tính sa\_flags của struct sigaction sử dụng trong hàm sigaction được gán một trong các giá trị sau:

SA\_RESETHAND: Khôi phục handler gốc cho signal. Nhiều system call sử dụng trong một chương trình đều có thể bị ngắt. Khi đó, các system call này sẽ trả về lỗi và lưu giá trị errno thành EINTR để thể hiện rằng hàm trả về do một signal gây ra. Hành vi này cần phải được code rất kỹ lưỡng. Nếu chúng ta lưu cờ sa\_flags thành SA\_RESTART thì khi hàm sigaction được gọi, hiện tượng ngắt xảy ra.

Thông thường, khi handler của một signal nào đó được chạy, signal này sẽ được thêm vào signal mask của process trong suốt quá trình handler chạy. Điều này ngăn việc nhiều signal cùng một loại tới cùng lúc dẫn tới việc handler của nó phải chạy lại liên tục và gây ra lỗi. Tuy nhiên, nếu cờ sa\_flags được set thành SA\_NODEFER, tính năng này sẽ được bỏ qua và signal mask sẽ không có sự thêm bớt nào.

Một số hàm có thể được gọi một cách an toàn bên trong signal handler. Điều này được đảm bảo bởi X/Open trong cả trường hợp đăng ký lại lẫn trường hợp hàm không tự tạo signal. Tất cả các hàm còn lại được xem là không an toàn.

Bảng 27: Các hàm an toàn để sử dụng trong signal handler

|  |  |  |  |
| --- | --- | --- | --- |
| access | alarm | cfgetispeed | cfgetospeed |
| cfsetispeed | cfsetospeed | chdir | chmod |
| chown | close | creat | dup2 |
| dup | execle | execve | \_exit |
| fcntl | fork | fstat | getegid |
| geteuid | getgid | getgroups | getpgrp |
| getpid | getppid | getuid | kill |
| link | lseek | mkdir | mkfifo |
| open | pathconf | pause | pipe |
| read | rename | rmdir | setgid |
| setpgid | setsid | setuid | sigaction |
| sigaddset | sigdelset | sigemptyset | sigfillset |
| sigismember | signal | sigpending | sigprocmask |
| sigsuspend | sleep | stat | sysconf |
| tcdrain | tcflow | tcflush | tcgetattr |
| tcgetpgrp | tcsendbreak | tcsetattr | tcsetpgrp |
| time | times | umask | uname |
| unlink | utime | wait | waitpid |

Các handler mặc định của các signal trong Bảng 28 được gọi là kết thúc bất thường của một process với tất cả các hệ quả của \_exit. Điều này tương tự với việc exit mà không thực hiện bất cứ hành động cleanup nào trước khi trả về cho kernel. Tuy nhiên, chúng ta vẫn có thể sử dụng hàm wait và waitpid để chỉ ra việc kết thúc bất thường do signal nào gây ra.

Bảng 28: Các signal thể hiện sự kết thúc bất thường của một process

|  |  |
| --- | --- |
| Tên signal | Mô tả |
| SIGALRM | Tạo bởi timer từ hàm alarm |
| SIGHUP | Gửi tới process đang được điều khiển bởi một terminal bị mất kết nối hoặc từ một process đang trong quá trình terminate các foreground process. |
| SIGINT | Ctrl + C |
| SIGKILL | Signal này không thể bị handle hoặc ignore. Buộc process kết thúc. |
| SIGPIPE | Tạo ra bởi một quá trình cố gắng viết vào PIPE. |
| SIGTERM | Gửi tiến hiệu yêu cầu kết thúc tới process. Được sử dụng khi shutdown. Signal này được mặc định gửi khi sử dụng lệnh kill. |
| SIGUSR1, SIGUSR2 | Dùng để liên lạc giữa các process. |

Các signal trong Bảng 29 mặc định cũng được xem là gây ra việc kết thúc bất thường. Thêm vào đó, trong một số tình huống như việc tạo file core, các signal này được xem là bình thường.

Bảng 29: Signal gây ra sự kết thúc bất thường của process

|  |  |
| --- | --- |
| Tên signal | Mô tả |
| SIGFPE | Tạo ra bởi floating-point arithmetic exception. |
| SIGILL | Lệnh không hợp lệ. Thương được gây ra bởi một chương trình lỗi hoặc shared memory module không hợp lệ. |
| SIGQUIT | Ctrl + \ |
| SIGSEGV | Vi phạm vùng nhớ. Bao gồm truy cập ngoài vùng của một mảng. Tràn phân vùng stack. Truy cập sử dung pointer không hợp lệ. |

Mặc định, các process sẽ tạm dừng khi nhận một trong các signals trong Bảng 30.

Bảng 30: Các signal gây tạm dùng process

|  |  |
| --- | --- |
| Tên signal | Mô tả |
| SIGSTOP | Dừng việc thực thi. Signal này không thể bị handle hoặc ignore. |
| SIGTSTP | Ctrl + Z. |
| SIGTTIN, SIGTTOU | Background process dừng. |

Signal SIGCONT được dùng để tái khởi động một process bị dừng và được bỏ qua nếu process đang không dừng.

Bảng 31: Signal dùng để điều khiển việc dừng và chạy tiếp của process con

|  |  |
| --- | --- |
| Tên signal | Mô tả |
| SIGCONT | Continue executing, if stopped. |
| SIGCHLD | Raised when a child process stops or exits. |

#### Thread

Các điểm quan trọng cần lưu ý về thread:

* Tạo một thread mới từ một process
* Đồng bộ dữ liệu giữa các thread trong cùng một process
* Điều chỉnh các thuộc tính của một thread
* Điều khiểm thread này từ thread khác trong cùng một process.

Một chuỗi các điều khiển trong một process gọi là thread. Khi một process chạy lệnh fork, một bản copy của chính process này sẽ được tạo ra với PID và biến mới. Process mới này sẽ được lập lịch và hoạt động gần như độc lập với process đã tạo ra nó. Trái lại, khi chúng ta tạo một thread mới từ một process, thread này sẽ sở hữu stack riêng của nó (chứa các biến local) nhưng lại chia sẻ chung các biến global, fd, signal handler, và trạng thái đường dẫn hiện tại với process đã tạo ra nó.

Ưu điểm và nhược điểm khi sử dụng thread so với việc tạo ra một process mới:

***Ưu điểm:***

* Thuận lợi cho việc xử lý nhiều công việc trong cùng một chương trình khi mà yêu cầu khoá tài nguyên và giữ cho dữ liệu không bị xung đột trở nên quan trọng. Việc này thực hiện trên đa luồn dễ hơn đa tiến trình.
* Hiệu xuất của ứng dụng sẽ được cải thiện trong trường hợp ứng dụng cần kết hợp các ngõ vào, các kết quả tính toán trung gian, và ngõ ra. Cụ thể hơn, nếu chung ta sử dụng 3 threads để quản lí ba hoạt động này, trong khi đang chờ kết nối của ngõ vào và ngõ ra, chúng ta có thể thực hiện việc tính toán.
* Tận dụng tối ưu được tài nguyên sẵn có từ các vi xử lý đa nhân nếu đặc tính của chương trình phù hợp.
* Việc đổi qua đổi lại giữa những thread sẽ ít tốn “công” (tài nguyên, thời gian xử lý, …) hơn so với giữa những process.

***Nhược điểm:***

* Chính khả năng chia sẽ biến giữa những thread làm cho việc lập trình một chương trình đa luồng sẽ khó khăn hơn trong việc canh thời gian và tránh việc chia sẻ ngoài ý muốn.
* Debug một chương trình đa luồng sẽ khó hơn chương trình đơn luồng rất nhiều vì sự tương tác giữa các thread rất khó để kiểm soát.
* Một chương trình chia việc tính toán thành nhiều thread chỉ nhanh hơn khi chạy trên vi xử lý đa lõi và các phép tính có thể thực hiện đồng thời.

Hình : Phân vùng nhớ của chương trình đơn luồng (trái) và đa luồng (phải)

Các thư viện liên quan đến thread đa phần bắt đầu bằng pthread. Để sử dụng những thư viện này, chúng ta cần định nghĩa \_REENTRANT, thêm thư viện pthread.h, và link với thư viện sử dụng cờ -lpthread.

Các thư viện của UNIX và POSIX ban đầu được thiết kế với giả sử chương trình sử dụng nó chỉ đơn luồng. Một ví dụ rõ ràng nhất là errno - một biến sử dụng để thể hiện thông về lỗi gặp phải khi một hàm nào đó không thành công. Nếu chúng ta lập trình một chương trình đa luồng, biến errno sẽ được chia sẻ giữa các thread. Điều này sẽ dẫn tới việc nhiều lỗi cùng ghi đè lên biến này và chúng ta chỉ lưu được lỗi cuối cùng gần nhất.

Các phần mã nguồn được tái đăng ký sẽ có bản sao chép dữ liệu riêng cho mỗi lần gọi tới phần code này. Trong chương trình đa luồng, chúng ta sử dụng tính năng này bằng cách định nghĩa cờ \_REENTRANT trước khi include các mã nguồn vào chương trình của chúng ta. Việc này có 3 tác dụng:

* Một số hàm trong thư viện có hỗ trợ sẵn việc đăng ký lại. Các hàm này thường được thêm hậu tố \_r. Các hàm này sẽ được biên dịch ở chế độ tái đăng ký nếu \_REENTRANT được khái báo.
* Thư viện stdio.h thông thường đều có tính năng này.
* Biến errno từ thư viện errno.h sẽ được hỗ trợ đa luồng.

Dưới đây là một số hàm căn bản trong thư viện pthread.h. Hàm pthread\_create tương đường với hàm fork trong process.

#include <pthread.h>   
int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg);   
Trong đó:

pthread\_t là một định danh được ghi tới vùng nhớ mà thread trỏ tới.

pthread\_attr\_t là phần thuộc tính của thread được tạo. Thông thường, chúng ta không sử dụng đến các thuộc tính đặc biệt thì chỉ cần để NULL.

\*(\*start\_routine)(void \*)sẽ là hàm cần được thực thi dưới dạng một thread. Chúng ta cần truyền vào địa chỉ của một hàm có tham số là một con trỏ kiểu void và trả về một con trỏ kiểu void. Nhờ vậy, chúng ta có thể truyền vào bất kì hàm nào có một tham số và trả về con trỏ. Trong khi sử dụng hàm fork sẽ dẫn tới việc tiếp tục thực thi tại cùng một ví trí với sự trả về khác nhau, sử dụng một thread mới sẽ cấp con trỏ hàm tới thread mới cần được thực thi.

\*arg là tham số được truyền vào hàm cần thực thi.

Hàm này trả về error-code integer.

Để kết thúc một thread, chúng ta sử dụng hàm pthread\_exit. Hàm này trả về con trỏ và xoá hết tất cả các biến cục bộ nên chúng ta không được trả về con trỏ tới các biến cục bộ.

void pthread\_exit(void \*retval);

Hàm pthread\_join có chức năng tương tự như hàm wait của process trong việc thu thập lại các process con.

int pthread\_join(pthread\_t th, void \*\*thread\_return);

Trong đó:

th là thread cần đợi (được tạo bởi hàm pthread\_create). thread\_return lấy giá trị trả về từ thread. Hàm này trả về error-code integer.

Tóm lại, các thread trong cùng 1 process chia sẻ những thông tin sau:

* Execution context (bao gồm bảng fd, bảng signal handler, user group id, đường dẫn làm việc hiện tại)
* Phân vùng heap, code, data

Các thông tin riêng của từng thread:

* Thread ID, PC
* Phân vùng stack và thanh ghi (nhưng phân vùng stack này vẫn có thể tiếp cận bởi những thread khác)
* errno, signal mask, độ ưu tiên

Khi các threads xử lý liên quan mật thiết với nhau vì chúng sử dụng chung rất nhiều tài nguyên, quá trình đồng bộ trở nên quan trọng hơn bao giờ hết. Hai phương pháp đồng bộ được giới thiệu đầu tiên là *semaphore* và *mutex*. Trong khi semaphore có hành vi như một người canh gác quanh một phần code nào đó thì mutex hoạt động dựa trên nguyên lý loại trừ lẫn nhau quan một phần code nào đó. Hai phương pháp này tương tự nhau và có thể implement lẫn nhau. Tuy nhiên, trong một số trường hợp cái này sẽ ưu việt hơn cái kia. Ví dụ, để kiểm soát việc truy cấp tới một vùng nhớ chia sẻ (vùng này chỉ có duy nhất một thread được truy cập trong một thời điểm) sẽ thích hợp với mutex hơn. Ngược lại, nếu chúng ta cần cấp quyền truy cập cho một đối tượng cụ thể nào đó thì semaphore sẽ phù hợp hơn.

Đối với semaphore, có 2 tập hàm đến từ POSIX Realtime Extension (dùng cho thread) và từ System V (dùng để đồng bộ process). Tập hàm thứ 2 sẽ được đề cập ở phần kế tiếp. Thông thường, semaphore chỉ cho phép một thread truy cập một phần code tại bất kể thời điểm nào. Tuy nhiên, chúng ta có thể tăng số lượng thread lên nhiều hơn một bằng cách sử dụng *counting semaphore*.

Hàm semaphore không sử dụng tiền tố pthread\_ như các hàm thread hay sử dụng mà lại dùng sem\_. Có 4 hàm semaphore cơ bản được sử dụng trong thread:

#include <semaphore.h>  
int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

int sem\_wait(sem\_t \* sem);

int sem\_post(sem\_t \* sem);

int sem\_destroy(sem\_t \* sem);

Hàm sem\_init dùng để tạo semaphore trỏ đến sem, cài đặt chế độ chia sẻ thông qua pshared và khởi tạo giá trị ban đầu bằng biến value. Nếu pshared = 0, semaphore sẽ được giới hạn cục bộ cho chỉ process hiện tại và ngược lại. Hiện tại chúng ta chỉ xét pshared = 0.

Hàm sem\_post dùng để tăng giá trị semaphore thêm 1. Nếu hai thread cùng tăng 1 semaphore thì nó sẽ thành tăng 2. Ngược lại, hàm sem\_wait lại giảm giá trị semaphore đi một (Nếu giá trị hiện tại là 0 thì nó sẽ đợi tới lúc ai đó tăng lên 1 rồi mới tiến hành giảm đi). Nếu có hai thread cùng gọi hàm sem\_wait với một semaphore bằng 0 thì chỉ một thread thắng và thread còn lại phải đợi. Điều này tạo nên đặc tính giá trị của semaphore.

Hàm sem\_destroy dùng để giải phóng vùng nhớ mà con trỏ sem đang trỏ tới. Hàm này sẽ báo lỗi khi chúng ta clean một semaphore mà có một thread nào đó đang chờ nó.

Tất cả 4 hàm này đều trả về error-code integer.

Đối với mutex, phương pháp đồng bộ này cho phép người tập trình có thể khoá một đối tượng nào đó với điều kiện chỉ cho phép một thread truy cập tại mỗi thời điểm. Để làm được điều này, chúng ta cần khoá phần code cần sử dụng trước khi vào sử dụng và mở khoá khi sử dụng xong. Có bốn hàm cơ bản trong mutex tương tự với semaphore:

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr);

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex));

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

Bốn hàm này trả về error-code integer (lưu ý errno không được set). Trong đó mutexatrr thường được mặc định là “fast”. Điều này dẫn tới sự bất lợi là nếu chúng ta gọi hàm pthread\_mutex\_lock đã được khoá, chương trình sẽ bị khoá luôn. Lý do là thread đang giữ chìa khoá lại bị khoá. Hiện tượng này được gọi là *deadlocked*.

Quay trở lại tham số pthread\_attr\_t \*attr trong hàm pthread\_create, chúng ta có thể tác động lên thread bằng cách điều chỉnh thuộc tính của nó. Như đã đề cập, để tái đồng bộ các thread trước khi cho phép chương trình kết thúc, chúng ta sử dụng hàm pthread\_join. Tuy nhiên, chúng ta chỉ cần làm điều náy nếu chúng ta muốn cho phép một thread nào đó trả kết quả về cho thread đã sinh ra nó hoặc chờ đợi thread nào đó kết thúc trước khi tiếp tục làm gì đó. Ví dụ chúng ta muốn tạo thêm một thread để đảm nhận nhiệm vụ autosave một file Word. Nếu người dùng cài đặt cứ mỗi phút autosave một lần thì tương ứng mỗi lần sẽ có một thread mới được sinh ra, thực hiện vụ lưu rồi kết thúc chứ không cần phải trả về hay main thread phải chờ nó. Các thread loại này được gọi là *detached thread*. Để tạo loại thread này, chúng điều chỉnh attr bằng cách sử dụng hàm:

int pthread\_attr\_init(pthread\_attr\_t \*attr);

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);

int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);

int pthread\_attr\_getschedpolicy(const pthread\_attr\_t \*attr, int \*policy);

int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param);

int pthread\_attr\_getschedparam(const pthread\_attr\_t \*attr, struct sched\_param \*param);

int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr, int inherit);

int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr, int \*inherit);

int pthread\_attr\_setscope(pthread\_attr\_t \*attr, int scope);

int pthread\_attr\_getscope(const pthread\_attr\_t \*attr, int \*scope);

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, int scope);

int pthread\_attr\_getstacksize(const pthread\_attr\_t \*attr, int \*scope);

Trong đó:

* + \_setdetachstate: Thuộc tính này giúp thread không cần rejoin. Với detachstate: PTHREAD\_CREATE\_JOINABLE (mặc định) hoặc PTHREAD\_CREATE\_DETACHED
  + \_setschedpolicy: Hàm này điều khiểu cách thread được lập lịch. Với policy: SCHED\_OTHER (mặc định), SCHED\_RP (real-time, cần chạy dưới quyền root) , SCHED\_FIFO.
  + \_setschedparam: Hàm này sử dụng cùng với hàm \_setdschedpolicy khi sử dụng tham số SCHED\_OTHER để điều khiển việc lập lịch cho thread.
  + \_setinheritsched: Hàm này nhận giá trị PTHREAD\_EXPLICIT\_SCHED (mặc định, việc lập lẽ được quyết định bởi các tùy chọn về thuộc tính) hoặc PTHREAD\_INHERIT\_SCHED (thừa hưởng các thuộc tính lập tính từ thread sinh ra nó) cho inherit.
  + \_setscope: Hàm này điều khiển cách mà việc lập lịch được tính toán.
  + \_setstacksize: Hàm này điều khiển kích thước stack của thread (đơn vị: byte)

Có nhiều cách để kết thúc một thread: gửi signal hoặc yêu cầu cancel. Ở phía main thread, chúng ta gọi hàm pthread\_cancel. Ở phía thread cần được kết thúc, chúng ta dùng hàm pthread\_setcancelstate.

int pthread\_cancel(pthread\_t thread);

int pthread\_setcancelstate(int state, int \*oldstate);

Trong đó, thread là thread ID cần được kết thúc. state nhận giá trị PTHREAD\_CANCEL\_ENABLE để cho phép thread được nhận yêu cầu cancel hoặc PTHREAD\_CANCEL\_DISABLE cho điều ngược lại. Tham số oldstate cho chúng ta lấy lại trạng thái cũ, nếu không dùng thì truyền vào NULL. Trong trường hợp chúng ta cho phép thread được cancel, chúng ta có thêm một mức điều khiển nữa – đó là kiểu cancel, được chọn bởi hàm pthread\_setcanceltype.

int pthread\_setcanceltype(int type, int \*oldtype);

Tham số type nhận giá trị PTHREAD\_CANCEL\_ASYNCHRONOUS để buộc yêu cầu cancel được thực hiện ngay hoặc PTHREAD\_CANCEL\_DEFFERED để chờ cho thread thực hiện mooijt trong các hàm: pthread\_join, pthread\_cond\_wait, pthread\_cond\_timewait, pthread\_testcancel, sem\_wait, sigwait rồi mới cancel. Tham số oldtype cho phép sử dụng lại state trước đó hoặc dùng NULL nếu không sử dụng tính năng này.

Chú ý lại rằng, khi một thread bắt đầu, trạng thái cancel mặc định của thread là PTHREAD\_CANCEL\_ENABLE và kiểu cancel là PTHREAD\_CANCEL\_DEFERRED.

### Inter-process communication

Như đã trình bày ở trên trong khi chương trình đa luồng có thể sử dụng chung tất cả ngoại trừ biến cục bộ của từng thread và cơ chế semaphore/mutex để tránh xung đột khi sử dụng chung tài nguyên giúp cho các thread có khả năng trao đổi thông tin và tương tác nhau cực kì tốt. Trong khi đó, chúng ta thấy các process gần như độc lập và cách tương tác giữa chúng là sử dụng signal. Cơ chế này có vẻ hạn chế lượng thông tin có thể trao đổi giữa các process. Trong phần này, chúng ta sẽ tìm hiểu các khái niệm mới liên quan tới inter-process communication:

* + Pipe
  + Process pipe
  + Pipe calls
  + Process cha và process con.
  + FIFO pipe

Chúng ta sử dụng thuật ngữ *pipe* để diễn tả việc kết nối luồng dữ liệu từ một process đến các process khác. Thông thường, chúng ta có thể pipe output của một process đến input của một process khác.

Cách lập trình đơn giản nhất mà các lập trình viên thường dùng để truyền data giữa hai chương trình là sử dụng hàm popen và pclose.

FILE \*popen(const char \*command, const char \*open\_mode);

int pclose(FILE \*stream\_to\_close);

Hàm popen cho phép một chương trình gọi một chương trình khác như một process mới của nó và truyền data tới process đó hoặc nhận data từ process đó. command là tên của chương trình cần chạy với tham số open\_mode là r hoặc w. Với tùy chọn r, output của chương trình trình mới sẽ được đọc bằng cách dùng con trỏ trả về kiểu FILE. Ngược lại, w cho phép gửi data tới process mới thông qua standard input. Process mới sẽ hoàn toàn không quan tâm là input của nó đến từ một process mới mà chỉ đơn giản là đọc từ standard input stream. Lưu ý, chúng ta không có tùy chọn rw. Do đó cách để truyền thông tin hai chiều đó là sử dụng hai pipe.

Khi một process được bắt đầu sử dụng hàm popen đã kết thúc, chúng ta có thể đóng file stream bằng hàm pclose. Hàm này sẽ trả về khi process được mở bằng popen đã kết thúc hoặc chờ cho đến khi nó kết thúc. Nếu process đó đã được wait trước khi pclose thì exit status sẽ bị mất vì process đó đã kết thúc và pclose trả về -1 (errno được set qua ECHILD).

Hàm popen chạy một chương trình được yêu cầu bằng cách gọi một shell mới, sh, rồi truyền command như một tham số. Các triển khai này vừa tốt vừa xấu. Trong Linux, việc mở rông tham số được thực hiện bởi shell ví dụ như xác định các file \*.c liên quan trước khi chương trình bắt đầu. Điều này cho phép popen thực hiện những shell command phức tạp. Trong khi đó, các hàm như execl sẽ trở nên phức tạp hơn vì việc gọi process của nó phải tự thực hiện mở rộng tham số. Trái lại mỗi lần gọi popen đều sử dụng shell dẫn tới phải sử dụng hai process làm tốn tài nguyên và việc chạy command sẽ chậm hơn.

Cách gọi pipe ở mức thấp hơn là dùng hàm pipe. Hàm nãy thực hiện truyền data giữa hai chương trình mà cần dùng tới trong khi cho phép chúng ta can thiệp nhiều hơn vào cách data được viết và đọc.

int pipe(int file\_descriptor[2]);

file\_descriptor chứa hai fd được kết nối một cách đặc biệt. Data viết vào file\_descriptor[1] sẽ được đọc bằng file\_descriptor[0] theo phương pháp FIFO. Có một điều quan trọng cần lưu ý là chúng ta đang sử dụng fd (kiểu integer sử dụng ở kernel space). Do đó chúng ta cần sử dụng syscall read() và write() để truy cập data thay vì dùng fread() và fwrite() (chỉ sử dụng cho kiểu dữ liệu FILE). Hàm này trả về error-code integer và set errno thành EMFILE (quá nhiều fd đang được process dùng), ENFILE (system file table bị đầy) và EFAULT (fd không hợp lệ).

Nếu hai process sử dụng pipe call là cha và con thì chúng ta dùng fork() hoặc exec(). Trong trường hợp sau, chúng ta chỉ cần truyền giá trị của file\_descriptor[0] cho process con được sinh ra thay thế cho process cha.

Lưu ý rằng, syscall read() sẽ làm process tạm dừng tới khi có data. Nếu pipe được đóng bởi một số nguyên nhân như process kết thúc, pipe sẽ trả về 0 và hàm read() sẽ không bị block nữa. Cơ chế này giúp process đang thực hiện quá trình đọc phát hiện được đầu bên kia đã bị đóng. Trong trường hợp chúng ta dùng fork(), có hai fd của pipe trong cả hai process. Do đó, chúng ta cần đóng write tới fd trên cả 2 process để pipe trở thành đóng đầu ghi thì read() mới phát hiện được.

Có một cách khác để kết nối hai process sử dụng pipe. Chúng ta có thể dùng một fd của pipe để lưu một giá trị biết trước. Giá trị này thường là 0 (standard input) hoặc 1 (standard output). Cách làm này sẽ làm cho quá trình khởi tạo trong process cha phức tạp hơn một chút nhưng quá trình con sẽ đơn giản hơn rất nhiều. Ưu điểm lớn nhất của phương pháp sử dụng pipe như standard IO này là chúng ta có thể gọi những chương trình không cần có tham số là fd. Để làm được điều này, chúng ta dùng hàm dup:

int dup(int file\_descriptor);

int dup2(int file\_descriptor\_one, int file\_descriptor\_two);

Hàm dup mở thêm một fd mới. Hàm này khác với open ở chỗ fd mới tạo bởi dup sẽ được trỏ đến cùng một file (hoặc pipe) với fd trước đó. fd mới này luôn là số nhỏ nhất khả dụng đối với hàm dup. Hàm dup2 có fd bằng hoặc số lớn hơn đầu tiên khả dụng so với file\_descriptor\_two.

Điểm mấu chốt ở phương pháp này là chúng ta luôn biết standard input có fd là 0 và fd mới sẽ có số nguyên dương khả dụng nhỏ nhất. Bằng cách đóng lại fd 0 và gọi hàm dup, fd mới sẽ có giá trị 0. Bởi vì fd mới sẽ là bản sao của fd cũ cho nên standard input sẽ truy xuất tới file hoặc pipe có fd được truyền vào hàm dup. Chúng ta tạo hai fd cùng trỏ tới cùng một file hoặc pipe, và một trong số chúng sẽ là standard input. Bằng cách quan sát Bảng 32, chúng ta sẽ hiểu được điều gì xảy ra khi chúng ta đóng fd 0 và gọi hàm dup.

Bảng 32: Thay đổi sau khi đóng fd và gọi hàm dup

|  |  |  |  |
| --- | --- | --- | --- |
| fd | Trước khi đóng fd | Sau khi đóng fd 0 | Sau khi gọi hàm dup |
| 0 | Standard input | Pipe file descriptor | Pipe file descriptor |
| 1 | Standard output | Standard output | Standard output |
| 2 | Standard error | Standard error | Standard error |
| 3 | Pipe file descriptor | Pipe file descriptor | Pipe file descriptor |

Các phương pháp ở trên chỉ sử dụng để truyền data giữa các chương trình có cùng nguồn gốc. Do đó phương pháp này sẽ bất tiện khi chúng ta cần truyền dữ liệu giữa những process không liên quan. Để giải quyết điều này chúng ta sử dụng phương pháp named pipes dựa trên ý tưởng FIFO. Một named pipe là một loại file đặc biệt có tên xuất hiện trong filesystem và không khác gì pipe đã được giới thiệu ở trên. Chúng ta có thể tạo một named pipe từ command line (dùng mknode *filename* p hoặc mkfifo *filename*) hay từ một chương trình sử dụng hàm mkfifo hoặc mknod:

int mkfifo(const char \*filename, mode\_t mode);

int mknod(const char \*filename, mode\_t mode | S\_IFIFO, (dev\_t) 0);

Một trong những đặc tính hữu ích của named pipe là nó có tên trong filesystem. Hạn chế duy nhất khi mở một FIFO là trong chương trình, chúng ta có thể không được mở một FIFO cho việc đọc/ghi dưới chế độ O\_RDWR. Nếu một chương trình vi phạm điều này, kết quả thu được sẽ không xác định. Thông thường, chúng ta sử dụng FIFO cho một chiều data duy nhất. Do đó chế độ O\_RDWR là rất nhạy cảm, một chương trường có thể đọc chính output của nó từ pipe nết nó được mở dưới chế độ read/write.

Nếu chúng ta không muốn truyền data theo hai chiều giữa các chương trình, tốt hơn hết hãy sử dụng một cặp pipe hoặc named pipe (FIFO), mỗi chiều một cái. Ngoài ra người ta vẫn có thể triển khai trao đổi dữ liệu 2 chiều sử dụng FIFO. Ví dụ như đóng rồi mở lại một FIFO để thay đổi chiều. Một điểm khác nhau nữa khi mở một FIFO và một file thông thường là ở chỗ sử dụng cờ open\_flag ở giá trị O\_NONBLOCK. Mode này không những thay đổi cách thực thi của hàm open mà còn thay đổi cách read và write yêu cầu fd trả về. Có 4 trường hợp hợp lệ được mô tả trong Bảng 33.

Bảng 33: Các chọn lựa hợp lệ

|  |  |
| --- | --- |
| Cách sử dụng | Hành vi |
| open(const char \*path, O\_RDONLY); | Hàm open sẽ bị block. Chỉ trả về khi một process mở FIFO này để ghi. |
| open(const char \*path, O\_RDONLY | O\_NONBLOCK); | Hàm open sẽ được thực thi và trả về ngay lập tức ngay cả khi FIFO đang được mở để ghi bởi một process nào đó. |
| open(const char \*path, O\_WRONLY); | Hàm open bị block cho tới khi một process mở FIFO này để đọc. |
| open(const char \*path, O\_WRONLY | O\_NONBLOCK); | Trả về ngay lập tức. Nếu không có process nào mở FIFO để đọc, hàm open sẽ trả về -1 và FIFO sẽ không được mở. Nếu một process đang mở FIFO để đọc, fd trả về sẽ được dùng để viết vào FIFO. |

Lưu ý rằng, sự bất đối xứng giữa việc sử dụng O\_NONBLOCK chung với O\_RDONLY hoặc O\_WRONLY là mở không khóa để đọc sẽ fail nếu không có process mở pipe để đọc nhưng đọc không khóa sẽ không fail. Hành vi của hàm close sẽ không bị ảnh hưởng bởi cờ O\_NONBLOCK.

read một empty blocking FIFO sẽ đợi đến khi có data để read. Ngược lại, read một empty non-blocking FIFO sẽ trả về 0 bytes.

write một full blocking FIFO sẽ đợi đến khi data có thể ghi được. write một FIFO không thể chấp nhận tất cả các byte ghi sẽ dẫn tới một trong hai trường hợp:

* + Thất bại nếu yêu cầu ít hơn PIPE\_BUF bytes và không có data nào được ghi.
  + Ghi được một phần nếu yêu cầu nhiều hơn PIPE\_BUF bytes và trả về số lượng bytes đã được viết.

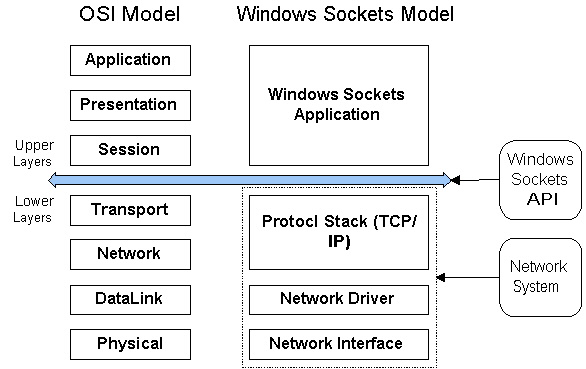
Kích thước của một FIFO rất quan trọng để lưu tâm. Có một số hệ thống sẽ đặt ra giới hạn về kích thước của FIFO. Giới hạn này được định nghĩa bởi macro PIPE\_BUF trong thư viện limits.h. Thông thường giới hạn này có kích thước 4096 bytes hoặc 512 bytes. Hệ điều hành Linux sẽ đảm bảo việc viết bằng hoặc ít hơn PIPE\_BUF bytes vào FIFO được mở ở mode O\_WRONLY (blocking) không bao giờ xảy ra hiện tượng viết được một phần (hoặc viết được hết, hoặc không viết được bytes nào). Mặc dù giới hạn này không quan trọng trong trường hợp chúng ta có một FIFO writer và một FIFO reader, việc sử dụng một FIFO cho nhiều chương trình cùng gửi request tới một FIFO reader rất phổ biết. Nếu nhiều chương trình cố gắng ghi vào FIFO cùng lúc, chúng ta phải đảm bảo rằng các write request phải khóa FIFO và có kích thước nhỏ hơn PIPE\_BUF.

### Socket

#### Khái niệm

Socket interface (phần mở rộng của pipe) là một phương pháp truyền thông giữa các process trên những máy tính khác nhau hoặc trên cùng một máy tính. Socket là một cơ chế truyền thông cho phép các hệ thống client/server được phát triển trên cùng một máy hoặc trên một mạng máy tính. Socket khác cơ chế pipe ở chỗ nó có sự phân biệt giữa client và server. Cơ chế socket có thể triển khai nhiều clients cùng với một server.

Socket có vị trí ở giữa tầng transport và tầng session trong mô hình OSI (Hình 20).

**

Hình 20: Vị trí của socket trong mô hình OSI

#### Kết nối

Đầu tiên, ứng dụng chạy trên server sẽ tạo một socket (fd) dùng syscall socket và đặt trên filesystem dùng syscall bind tại đường dẫn /tmp hoặc /user/tmp. Sau đó, syscall listen dùng tạo queue cho những truy cập đến socket và chấp nhận sử dụng syscall accepts. Khi một kết nối được chấp nhận, server sẽ tạo một socket mới với tên mới để phục vụ cho kết nối đó. Về phía client, chúng ta chỉ cần tạo socket dùng socket và thiết lập kết nối dùng connect. Sau đó, việc thông tin liên lạc hai chiều sẽ được thực hiện bằng cách đọc ghi đến fd đã được tạo.

Thuộc tính của socket bao gồm *domain*, *type*, và *protocol*:

* Họ giao thức (Domain)
* AF\_INET address family: Cho phép các process trong cùng hệ thống hay khác hệ thống giao tiếp với nhau. Bao gồm địa chỉ ip và cổng port
* AF\_INET6 address family: Tương tự như AF\_INET nhưng phục vụ cho Internet Protocol ver6.
* AF\_UNIX address family: Chỉ cho phép các process trong cùng hệ thống giao tiếp với nhau.
* Loại (Type)
* Stream (dựa trên TCP/IP)
* Datagram (Dựa trên UDP/IP)
* Giao thức (Protocol): Có thể chỉ định protocol cho socket, nếu mặc định 0 chương trình sẽ tìm protocol thích hợp.
* Local Socket Address: Địa chỉ của socket của process cục bộ.
* Remote Socket Address: Địa chỉ của socket của process từ xa.

Các syscall được sử dụng ở trên có API như :

* Hàm tạo socket: int socket(int domain, int type, int protocol); Hàm này tạo socket, tham số domain(ipv4-v6-UNIX), type(STREAM hoặc DGRAM), protocol(0 để socket chọn protocol thích hợp sử dụng), hàm trả về số tham chiếu file descriptor.
* Hàm khóa socket: int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen); Hàm này gán địa chỉ, cổng, giao thức vào socket. Sau đó sao chép my\_addr vào kernel space.
* Hàm lắng nghe: int listen(int sockfd,int backlog);(Chỉ TCP) Lắng nghe client, chuyển sang socket thụ động, chờ chấp nhận kết nối từ client, backlog là số lượng tối đa server có thể accept (tùy vào hệ điều hành backlog khác nhau thì số lượng tối đa client khác).
* Hàm kết nối: int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen); (Chỉ TCP) Thiết lập kết nối tới port và ip định sẵn.
* Hàm chấp nhận kết nối: int accept (int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen); (Chỉ TCP) chấp nhận kết nối từ client. Cấp 1 vùng nhớ socket khác(fd khác) dưới kernel space, các giao tiếp connect sẽ thông qua socket mới này.
* Một số hàm chuyển đổi địa chỉ: inet\_aton, inet\_addr, inet\_ntoa, inet\_pton, inet\_ntop, sock\_ntop:
* Hàm inet\_aton:int inet\_aton(const char \*cp, struct in\_addr \*inp);

Chuyển đổi địa chỉ của Host Internet cp từ địa chỉ IPv4 dạng dotted-decimal (string) sang dạng địa chỉ số(32bit) và lưu vào trong struct mà inp trỏ tới.

Trả về giá trị khác 0 nếu địa chỉ là hợp lệ, ngược lại trả về 0.

* Hàm inet\_addr:in\_addr\_t inet\_addr(const char \*cp);

Tương tự inet\_aton, trả về địa chỉ số(32bit) từ đối số cp, trả về -1 nếu địa chỉ không hợp lệ. Tuy nhiên, -1 lại là một địa chỉ hợp lệ (255.255.255.255) trong network -> tránh dùng, nên sử dụng inet\_aton.

* Hàm inet\_ntoa: char \*inet\_ntoa(struct in\_addr in);

Ngược lại với hàm inet\_aton, hàm này chuyển địa chỉ IP từ dạng số sang dạng dotted-decimal (string). Hàm trả về một chuỗi chứa địa chỉ.

* Hàm inet\_pton:int inet\_pton(int af, const char \*src, void \*dst); Chuyển số ip(v6 hoặc v4 dựa vào đối số af) dotted-decimal (string) sang dạng số binary.

Hàm trả về 1 nếu thành công, 0 nếu src chứa địa chỉ sai, -1 nếu af sai.

* Hàm inet\_ntop:const char \*inet\_ntop(int af, const void \*src, char \*dst, socklen\_t size); Chuyển số binary sang ip(v6 hoặc v4 dựa vào đối số af) dotted-decimal (string). Hàm trả về một pointer đến dst nếu thành công nếu sai trả về NULL.
* Hàm sock\_ntop:char \*sock\_ntop(const struct sockaddr \*sockaddr, socklen\_t addrlen); Nhận vào một địa chỉ dạng struct sockaddr và trả về pointer trỏ đến địa chỉ dạng dotted-decimal (string). Nếu hàm không thành công, trả về NULL.
* Các hàm chuyển đổi địa chỉ nằm trong thư viện *arpa/inet.h*
* Hàm đọc dữ liệu socket: *ssize\_t read(int filedes, void \*buf, size\_t nbytes);* Đọc nbyte từ filedes vào buf.
* Hàm ghi dữ liệu socket: *ssize\_t write(int filedes, void \*buf, size\_t nbytes);* Ghi nbyte từ buf vào filedes.
* Ngoài ra còn một số hàm đọc/ghi khác như: int sendto(int sockfd, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to, int tolen); Gửi data tới 1 socket(socket có thể chưa connect, thường dùng với kiểu UDP)
* int recvfrom(int sockfd, void \*buf, int len, unsigned int flags struct sockaddr \*from, int \*fromlen); Gửi nhận data từ 1 socket(socket có thể chưa connect, thường dùng với kiểu UDP)

#### Network information

Để lấy thông tin về network của host , chúng ta sử dụng những hàm sau:

#include <netdb.h>

struct hostent \*gethostbyaddr(const void \*addr, size\_t len, int type);

struct hostent \*gethostbyname(const char \*name);

Struct trả về có dạng:

struct hostent {

char \*h\_name; /\* name of the host \*/

char \*\*h\_aliases; /\* list of aliases (nicknames) \*/

int h\_addrtype; /\* address type \*/

int h\_length; /\* length in bytes of the address \*/   
char \*\*h\_addr\_list /\* list of address (network order) \*/

};

Tương tự, để lấy thông tin của server, chúng ta sử dụng những hàm sau:

struct servent \*getservbyname(const char \*name, const char \*proto);

struct servent \*getservbyport(int port, const char \*proto);

Struct trả về có dạng:

struct servent {

char \*s\_name; /\* name of the service \*/  
char \*\*s\_aliases; int s\_port; /\* list of aliases (alternative names) \*/  
int s\_port; /\* The IP port number \*/  
char \*s\_proton; /\* The service type, usually “tcp” or “udp” \*/

};

#### Datagram

Trong một số trường hợp, ví dụ việc cập nhật ngày giờ chỉ yêu cầu tạo socket, thiết lập kết nối, đọc một response rồi đóng kết nối. Nếu chúng ta thiết lập và duy trì kết nối trong trường hợp này thật lãng phí tài nguyên và không cần thiết. Dó đó đối với những trường hợp này, chúng ta dùng UDP. Dữ liệu truyền nhận sử dụng UDP có khả năng sai rất cao nhưng chúng ta vẫn có thể kiểm tra được. Để sử dụng UDP, chúng ta vẫn phải dùng syscall socket và close nhưng dùng sendto và recvfrom thay cho read và write.

int sendto(int sockfd, void \*buffer, size\_t len, int flags, struct sockaddr \*to, socklen\_t tolen);

int recvfrom(int sockfd, void \*buffer, size\_t len, int flags, struct sockaddr \*from, socklen\_t \*fromlen);

## FreeRTOS

### Giới thiệu hệ điều hành FreeRTOS

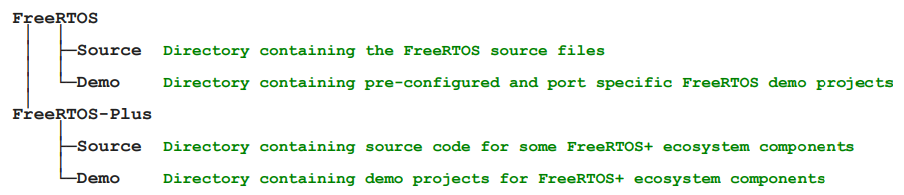
Hệ điều hành FreeRTOS được sử dụng cho những ứng dụng cần đặc tính real-time chạy trên các vi điều khiển hoặc vi xử lý đơn giản. Có hai loại yêu cầu về real-time: soft và hard. Ví dụ, sau khi bấm nút Esc 2s cửa sổ máy tính mới tắt đi tuy gây khó chịu nhưng vẫn chạy (soft). Trái lại, hệ thống nhúng sử dụng cảm biến va đập để kích hoạt túi khí trên xe hơi chậm 1s sau khi xảy va đập sẽ khiến hệ thống an toàn trở nên vô nghĩa (hard). FreeRTOS được thiết kế để đáp ứng yêu cầu hard real-time. FreeRTOS thường chạy trên các hệ thống chỉ có một nhân xử lý nên chỉ có thể xử lý một luồng tại một thời điểm. Do đó, hệ thống lập lịch của FreeRTOS gán độ ưu tiên cao cho các tác vụ cần hard real-time và ngược lại cho soft real-time. Hệ điều hành này miễn phí cho cả sản phẩm thương mại hóa.

Các tính năng tiêu chuẩn của FreeRTOS bao gồm:

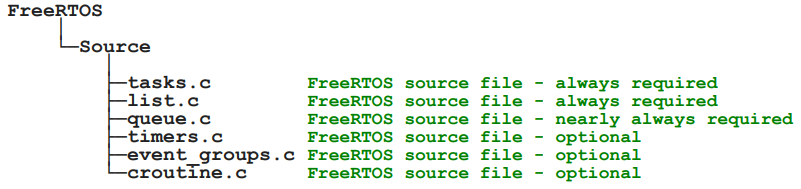
* + Pre-emptive/co-operation
  + Điều chỉnh linh hoạt độ ưu tiên của Task
  + Cơ chế truyền thông task nhanh, gọn, lẹ.
  + Queues
  + Semaphores (Binary và Counting)
  + Mutexes và Mutexes đệ quy
  + Software timer
  + Event group
  + Tick hook, idle hook
  + Kiểm tra tràn stack
  + Trace record
  + Thu thập thời gian chạy của các task
  + Interrupt nesting model
  + Tắt Tick cho ULPA (Ultra Low Power Application)
  + Software managed interrupt stack

Bảng 34: Một số khái niệm được dùng khi giới thiệu về hệ điều hành FreeRTOS

|  |  |
| --- | --- |
| FreeRTOS Port | FreeRTOS hỗ trợ khoảng 20 trình biên dịch và 30 kiễn trúc xử lý. Như vậy chúng ta có khoảng 600 bản code, mỗi bản gọi là một FreeRTOS port. Chúng ta nên tham khảo demo application cho mỗi port để include đúng các file source và header. |
| Build FreeRTOS | FreeRTOS được xem như các thư viện và cuối cùng sẽ được build chung thành một bare metal application. |
| FreeRTOSConfig.h | File này sử dụng để điều chỉnh các đặc tính của FreeRTOS để phù hợp với từng ứng dụng. |



Tất cả các FreeRTOS port có chung các file sau:



Các source file riêng của từng port sẽ nằm trong đường dẫn FreeRTOS/Source/portable/[compiler]/[architecture].

Có 3 đường dẫn cần được include vào include path của compiler. Thứ nhất là FreeRTOS core header file: FreeRTOS/Source/include. Tiếp theo là source file của một port nào đó. Cuối cùng là FreeRTOSConfig.h. Nếu source code của chúng ta sử dụng các API của FreeRTOS, chúng ta phải include FreeRTOS.h và các header file chứa các API cần dùng (task.h, queue.h, semphr.h, timers.h, event\_groups.h).

Để bắt đầu một FreeRTOS project, chúng ta cần làm theo một trong hai quy trình mà developer đề nghị. Quy trình thứ nhất là bắt đầu từ demo project:

1. Mở demo project tương ứng với port rồi build và chạy thử để kiểm tra
2. Xóa tất cả source code định nghĩa các task demo. Xóa hết các file trong Demo/Common khỏi project.
3. Xóa tất cả các lệnh gọi hàm trong main() trừ prvSetupHardware() và vTaskStartScheduler().
4. Build thử.

Quy trình thứ 2 là bắt đầu từ scratch:

1. Sử dụng tool chain đã chọn. Tạo project như bình thường (chưa include các FreeRTOS file).
2. Build và chạy thử project trên phần cứng
3. Thêm FreeRTOS source file
4. Copy FreeRTOSConfig.h của demo project trong port tương ứng vào project
5. Thêm path vào project để tìm header file:
   1. FreeRTOS/Source/include
   2. FreeRTOS/Source/portable/[compiler]/[architecture]
   3. Path chứa FreeRTOSConfig.h
6. Copy compiler setting từ demo project của port tương ứng
7. Cài đặt thêm một số FreeRTOS interrupt handler nếu cần thiết (xem trong demo project)

Để dễ dàng sử dụng FreeRTOS, chúng ta cần nắm các kiểu dữ liệu và coding style của nhà phát triển FreeRTOS được mô tả trong bảng 35.

Bảng 35: FreeRTOS convention

|  |  |  |
| --- | --- | --- |
| Convention/Macro/Types/... | | Mô tả |
| TickType\_t | | Trong FreeRTOS, tick interrupt dùng để cấu hình chu kỳ interrupt.  Số lượng tick interrupt xảy ra tính từ lúc ứng dụng bắt đầu chạy được gọi là tick count. Tick count sẽ được sử dụng để tính thời gian.  Thời gian giữa hai tick interrup được gọi là tick period. Thời gian sẽ được tính bằng tich count \* tick period.  TickType\_t là kiểu dữ liệu dùng để lưu tick count và thời gian.  TickType\_t có thể tùy chọn giữa uint16\_t hoặc uint32\_t bằng cách cấu hình macro configUSE\_16\_BIT\_TICKS trong FreeRTOSConfig.h.  uint16\_t sẽ có performance cao hơn khi chạy trên kiến trúc 8-bit và 16-bit hơn nhưng bù lại sẽ bị giới hạn số lượng chu kỳ tối đa. Tất nhiên kiến trúc 32-bit sẽ dùng uint32\_t. |
| BaseType\_t | | Kiểu dữ liệu này thường được định nghĩa theo cách rồi ưu nhất cho từng kiến trúc. Ví dụ 32-bit cho kiến trúc 32-bit.  BaseType\_t thông thường được sử dụng cho các kiểu trả về bị giới hạn về khoảng giá trị hoặc được sử dụng cho kiểu booleans pdTRUE/pdFALSE. |
| int | | Không bao giờ được sử dụng |
| char | | Phải dùng kèm theo signed hoặc unsigned (trừ trường hợp biến char dùng để lưu ký tự ASCII hoặc con trỏ tới char được sử dụng để trỏ tới một string) |
| Biến chứa tiền tố | c | char |
| s | int16\_t (short) |
| i | int32\_t (long) |
| x | BaseType\_t |
| u | unsigned |
| P | Pointer |
| Hàm chứa tiền tố | v**Task**PrioritySet() | Trả về kiểu void và được định nghĩa trong **task.c** |
| x**Queue**Receive() | Trả về kiểu BaseType\_t và được định nghĩa trong **queue.c** |
| pv**Timer**GetTimerID() | Trả về pointer kiểu void và được định nghĩa trong **timers.c** |
| Tab | | Bằng 4 spaces |
| Macros | | Được viết in hoa chứa tiền tố viết thường chỉ file header định nghĩa macros |
| Semaphore | | Semaphore API được viết theo kiểu define macros nhưng theo convention của hàm chứ không theo convention của macros. |

### Quản lý Heap

FreeRTOS các phiên bản dưới V9.0 thì phải include *heap manager*, còn lại chỉ cần include khi cờ configSUPPORT\_DYNAMIC\_ALLOCATION được set qua 1 trong file FreeRTOSConfig.h. Kể tử V9.0 trở đi, các ứng dụng FreeRTOS có thể được cấp phát tĩnh toàn bộ nên có thể không cần tới heap memory manager.

Có 5 loại heap manager được định nghĩa trong file heap\_1.c tới heap\_5.c. Ví dụ được cung cấp ở đường dẫn FreeRTOS/Source/portable/MemMang. Ở phần này, chúng ta sẽ quan tâm 3 vấn đề chính: khi nào thì FreeRTOS cấp phát trên RAM, cách chọn loại heap manager và 5 ví dụ kèm theo.

***Heap\_1***

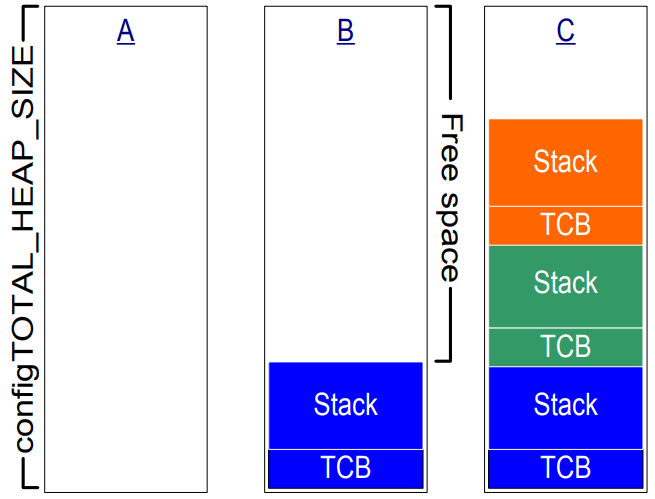
Trong các ứng dụng đơn giản, chúng ta chỉ cấp phát một số vùng nhớ bằng kernel trước khi ứng dụng bắt đầu. Những ứng dụng này không phải giải quyết các vấn đề như phân mảnh… Do đó, dùng heap\_1.c sẽ giúp code đơn giản và có kích thước nhỏ hơn.

Heap\_1.c implement phiên bản gọn nhất của pcPortMalloc() và không có hàm vPortFree(). Ứng dụng không bao giờ xóa task, kernel object sẽ dùng heap\_1. Các sản phẩm thương mại có yêu cầu an toàn cao, không cho phép cấp phát động cũng nên dùng heap\_1. Tổng kích thước của mảng được quy định bởi configTOTAL\_HEAP\_SIZE trong file FreeRTOSConfig.h. Sử dụng heap\_1 sẽ tốn RAM hơn bình thường do cách chi một mảng thành nhiều block nhỏ hơn. Từng task sẽ yêu cầu một task control block tương ứng và một vùng stack được cấp phát từ heap (Hình 21). Trong đó:

A: Trước khi task được tạo

B: Sau khi một task được tạo

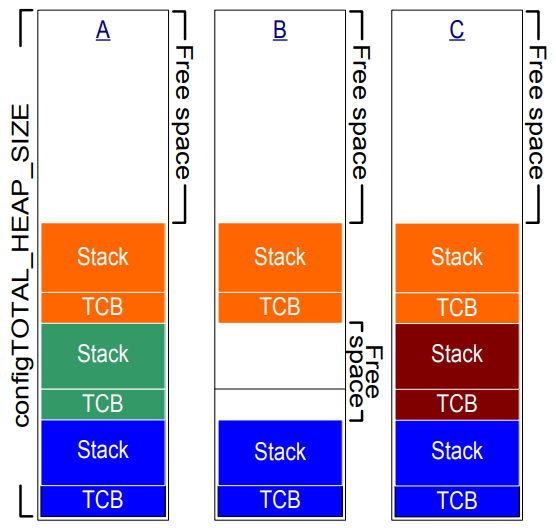
C: Sau khi 3 task được tạo



Hình 21: Cấp trúc mảng heap\_1 được cấp phát mỗi khi tạo một task

***Heap\_2***

Heap\_2 được phát triển cho việc tương thích ngược của các phiên bản FreeRTOS trước. Các sản phẩm mới nên dùng heap\_4 thay cho heap\_2. Điểm đáng chú ý của heap\_2 là nó sử dụng memory block trống có kích thước gần với yêu cầu cấp phát nhất và không nhập hai block trống sát nhau thành 1 block trống lớn hơn. Hình 22 thể hiện cách cấp phát của heap\_2 khi xóa và tạo một task.



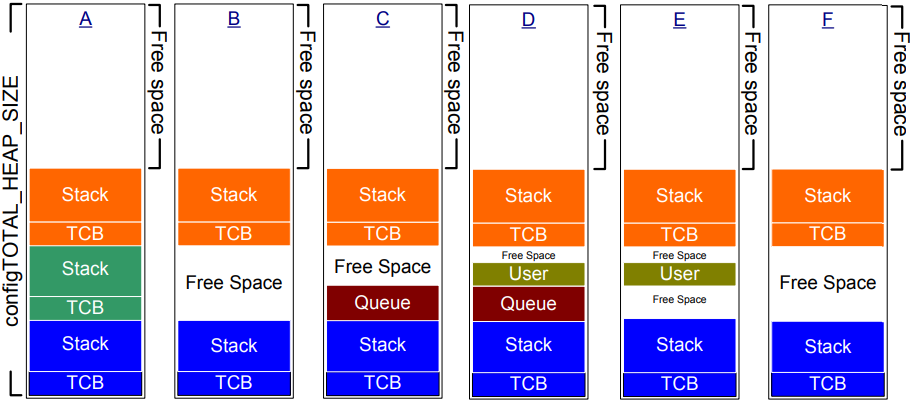
Hình 22: Heap\_2 xóa và tạo task

***Heap\_3***

Heap\_3 dùng malloc() và free() trong thư viện chuẩn. Do đó, kích thước heap sẽ được quyết định bởi cấu hình linker (configTOTAL\_HEAP\_SIZE sẽ không có tác dụng gì). Heap\_3 hỗ trợ malloc() và free() tính năng thread-safe bằng cách tạm dừng scheduler của FreeRTOS.

***Heap\_4***

Cũng giống như heap\_1 và heap\_2, heap\_4 cũng chia mảng thành những block nhỏ hơn. Nhưng điểm khác biệt là heap\_4 sát nhập các mảnh bộ nhớ thành một block lớn hơn làm giảm thiểu sự phân mảnh (Hình 23).



Hình 23: Quá trình cấp phát và thu hồi vùng nhớ của Heap\_4

***Heap\_5***

Tương tự heap\_4 nhưng heap\_5 không giới hạn việc cấp phát trong cùng một vùng tĩnh. Heap\_5 có thể cấp phát trên nhiều vùng nhớ tách biệt. Heap\_5 sẽ hữu dụng khi RAM trong hệ thống không xuất hiện trong system memory map dưới dạng một dải liên tục.

FreeRTOS hỗ trợ một số hàm hữu ích như:

size\_t xPortGetFreeHeapSize( void ); Hàm này trả về số lượng byte trống trong heap tại thời điểm gọi hàm. Hàm này hữu ích cho việc giảm kích thước của configTOTAL\_HEAP\_SIZE.

size\_t xPortGetMinimumEverFreeHeapSize( void ); Hàm này trả về số lượng byte trống thấp nhất kể từ khi ứng dụng bắt đầu chạy thười lúc gọi hàm. (Chỉ hỗ trợ trong heap\_4 và heap\_5)

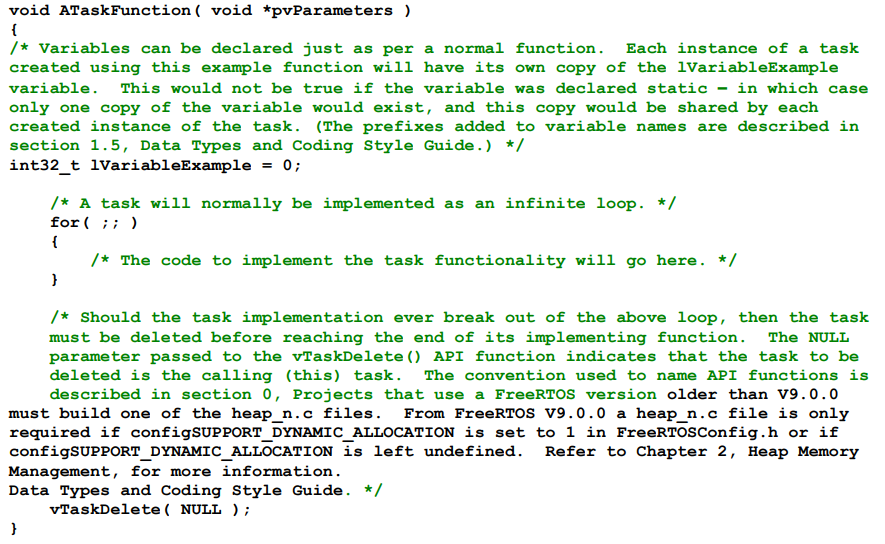
Hàm pvPortMalloc() có thể gọi trực tiếp từ code application hoặc từ FreeRTOS source mỗi lần một kernel object được tạo (task, queue, semaphores, và event groups). Hàm này trả về tương tự như hàm malloc(). Trong trường hợp trả về NULL, tất cả các heap\_\* đều hỗ trợ callback bằng cách set configUSE\_MALLOC\_FAILED\_HOOK thành 1 trong FreeRTOSConfig.h. Lúc này, ứng dụng chúng ta phải cung cấp một hàm với API là void vApplicationMallocFailedHook( void );

### Quản lý Task (Section quan trọng nhất của FreeRTOS)

FreeRTOS quản lý task trên những khía cạnh như thời gian task được thực thi, độ ưu tiên, trạng thái của task, ... Lập trình viên phải nắm được cách tạo một hoặc nhiều bản của một task, cách sử dụng các tham số của task, cách thay đổi độ ưu tiên task, xóa task, thiết lập chu kì cho task (dùng software timer) và sử dụng idle task.

***Task Function***

Task được implement như một hàm C với cấu trúc như Hình 24. Hàm task không được chứa return, exit và phải chứa vòng lặp vô hạn.



Hình 24: cấu trúc hàm task

***Top Level Task States***

Một task có hai trạng thái: running hoặc not running (trong thực tế trạng thái not running sẽ chi thành nhiều trạng thái hơn). Các trạng thái chi tiết hơn của not running sẽ giúp cho hệ thống quyết định task nào sẽ được chọn để thực thi và quay trở lại đúng vị trí nó đã dừng trong task và thực thi tiếp.

***Creating Tasks***

Để tạo task, chúng a có thể dùng xTaskCreate() hoặc xTaskCreateStatic():

BaseType\_t xTaskCreate( TaskFunction\_t pvTaskCode,

const char \* const pcName,

uint16\_t usStackDepth,

void \*pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \*pxCreatedTask);

Trong đó, pvTaskCode chính là con trỏ hàm trỏ tới hàm task đã được define. pcName là tên mô tả cho task để tiện cho việc đọc tên task khi debug (chiều dài tối đa được quy định bởi configMAX\_TASK\_NAME\_LEN). usStackDepth quy định kích thước stack cho mỗi task (đơn vị là word chứ không phải byte). Lưu ý, usStackDepth \* stack width phải chứa được trong range của unit16\_t. Kích thước stack của idle task sẽ được define bằng configMINIMAL\_STACK\_SIZE. Để tính kích thước stack tối đa cần thiết, chúng ta cần xem phần stack overflow. pvParameters là giá trị được truyền vào task. uxPriority đinh nghĩa độ ưu tiên của task (khoảng giá trị từ 0 – thấp nhất tới configMAX\_PRIORITIES – 1 – cao nhất). pxCreatedTask dùng để truyền hàm handle task được tạo. Hàm này trả về pdPASS hoặc pdFAIL (lúc này task không được tạo vì lý do không đủ bộ nhớ heap)

***Task Priorities***

Khi tạo task chúng ta đã set độ ưu tiên cho nó. Để thay đổi lại chúng ta dùng hàm vTaskPrioritySet(). Tuy nhiên FreeRTOS sử dụng một trong hai phương pháp để quyết định task nào tiếp theo sẽ được chạy thông qua việc điều chỉnh giá trị configMAX\_PRIORITIES bằng một trong hai phương pháp đó:

1. Generic method: Phương pháp này không giới hạn giá trị lớn nhất thành configMAX\_PRIORITIES. Tuy nhiên, chúng ta nên duy trì giá trị configMAX\_PRIORITIES thấp nhất có thể bởi vì nếu giá trị này càng cao thì càng tốn RAM và thời gian thực thi sẽ chậm hơn.

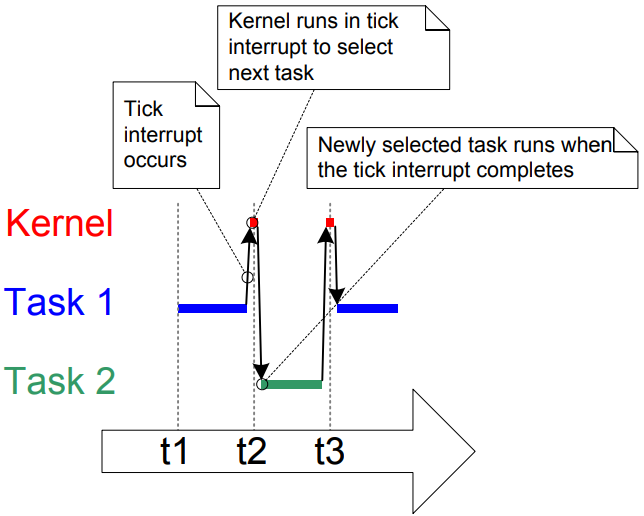
Phương pháp này sẽ được dùng nếu configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION được set về 0 hoặc undefine trong FreeRTOSConfig.h

1. Architecture Optimized Method: Phương pháp này sử dụng một lượng nhỏ code assembly và chạy nhanh hơn generic method. Lúc này, configMAX\_PRIORITES phải từ 32 trở xuống. Chỉ một số port support chức năng này.

FreeRTOS scheduler sẽ luôn đảm bảo task có độ ưu tiên cao nhất sẵn sàng chạy sẽ được chọn chạy. Nếu có nhiều task có cùng độ ưu tiên và sẵn sàng chạy, scheduler sẽ chạy thay phiên liên tục cho các task này.

***Time Measurement và Tick Interrupt***

Mỗi phiên chạy liên tục của các task có cùng độ ưu tiên là một time slice. Cuối mỗi time slice, scheduler sẽ thực thi để chọn task cho time slice tiếp theo. Hành động này được gọi là *tick interrupt* (Hình 25). Time slice sẽ được quy định bởi tick interrupt frequency dùng configTICK\_RATE\_HZ. Tùy ứng dụng mà sẽ có frequency tối ưu riêng nhưng thông thường người ta chọn 100 Hz.



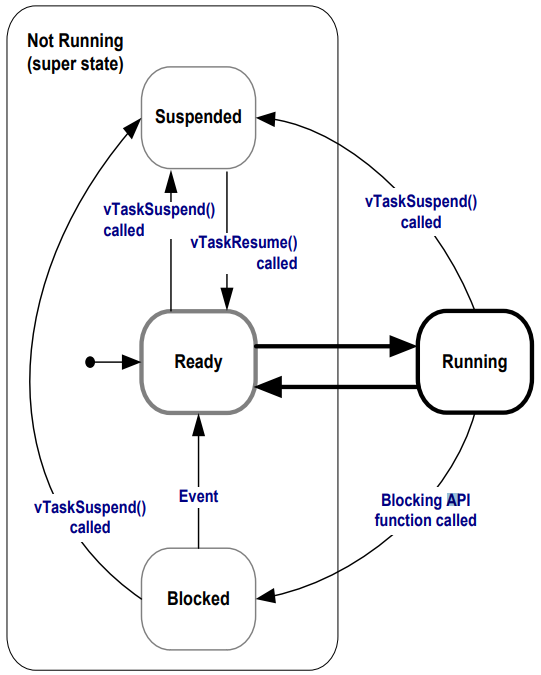
Hình 25: Tick interrupt

***Substate of Not-Running State*** (Hình 26)

*Blocked state*: Task đang đợi event. Có hai loại event mà task kiểu này đợi: temporal (ví dụ task được yêu cầu đợi 10ms để pass) và synchronization (đến từ task khác hoặc từ interrupt, ví dụ đợi data đến từ queue và thường được implement time out) FreeRTOS queues, semaphores, mutexes, event groups và task notification đều có thể dùng để tạo synchroniztion event.

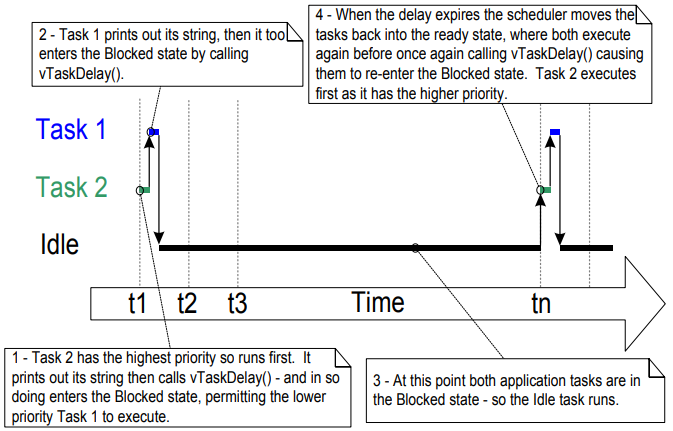
*Suspended state*: Scheduler sẽ không bao giờ sắp xếp cho task này chạy. Cách duy nhất để chuyên sang trạng thái này là gọi hàm vTaskSuspend() và kết thúc bằng hàm vTaskResume(). Trạng thái này hiếm khi được dùng.

*Ready state*: task không thuộc 2 trạng thái ở trên thì được xếp vào trạng thái này.



Hình 26: Task state

Thay vì sử dụng null loop, chúng ta sử dụng hàm vTaskDelay() để đưa task vào trạng thái blocked state với một lượng tick interrupt nhất định. Trong suốt thời gian này, chúng ta không sử dụng bất cứ thời gian thực thi nào của CPU (Hình 27).



Hình 27: execution sequence khi sử dụng vTaskDelay() thay cho null loop

***Idle Task và Idle Task Hook***

Khi không có task nào được thực thi, hệ thống sẽ tự động thực thi idle task. Task này được tạo tự động khi chạy hàm vTaskStartScheduler(). Task nào có độ ưu tiên thấp nhất (zero). Hàm này sẽ thực hiện một số công việc clean up sau khi một task được delete. Chúng ta có thể thêm một thàm nào đó vào idle task sử dụng idle hook. Hàm này sẽ được gọi mỗi idle task loop một lần. Idle hook thương được sử dụng để đo đạc khả năng tính toán của CPU cho một ứng dụng nào đó hoặc đưa CPU vào trạng thái tiết kiệm năng lượng (không tiết kiệm bằng cách dùng tick-less idle mode). Tuy nhiên, cần lưu ý trong idle hook, chúng ta không được block hay suspend.

***Thay đổi Task Priority***

Để thay đổi được độ ưu tiên của task, đầu tiên chúng ta cần set cờ INCLUDE\_vTaskPrioritySet lên 1 trong FreeRTOSConfig.h. Sau đó sử dụng hàm vTaskPrioritySet() để thay đổi và dùng uxTaskPriorityGet() lấy giá trị ưu tiên của task.

UBaseType\_t uxTaskPriorityGet( TaskHandle\_t pxTask );

UBaseType\_t uxTaskPriorityGet( TaskHandle\_t pxTask );

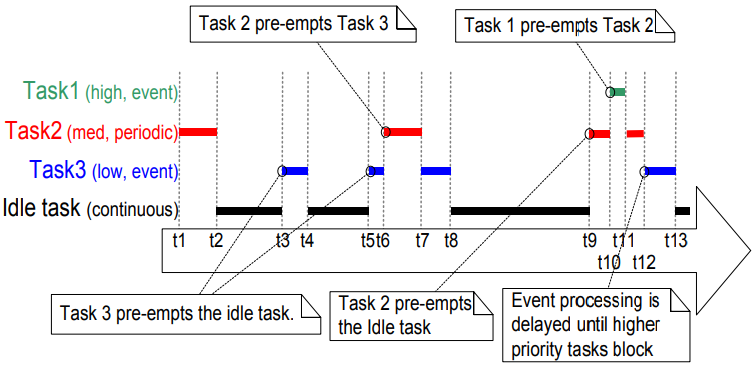
***Xóa Task***

Để xóa một task, đầu tiên chúng ta set cờ INCLUDE\_vTaskDelete lên 1 trong FreeRTOSConfig.h. Sau đó dùng hàm vTaskDelete(). Lưu ý rằng, khi trở lại idle task, chỉ có vùng nhớ được cấp phát bởi kernel sẽ được giải phóng, còn các vùng nhớ chúng ta cấp phát trong user space phải tự giải phóng.

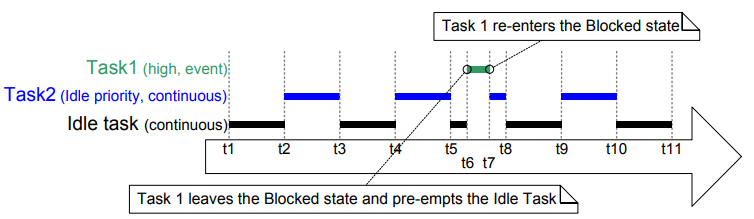
Để thay đổi thuật toán scheduler, chúng ta sử dụng hai macro configUSE\_PREEMPTION, configUSE\_TIME\_SLICING, và configUSE\_TICKLESS\_IDLE trong FreeRTOSConfig.h.

Bảng 36: Các mode hoạt động của FreeRTOS scheduler

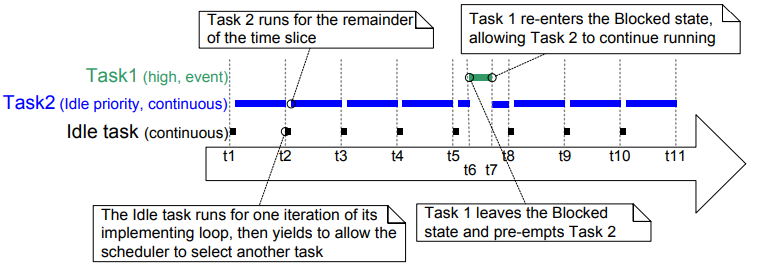
|  |  |  |  |
| --- | --- | --- | --- |
|  | Mode 1: Prioritized Pre-emptive Scheduling with Time Slicing | Mode 2: Prioritized Pre-emptive Scheduling (without Time Slicing) | Mode 3: Co-operative Scheduling |
| configUSE\_PREEMPTION | 1 | 1 | 0 |
| configUSE\_TIME\_SLICING | 1 | 0 | (any) |



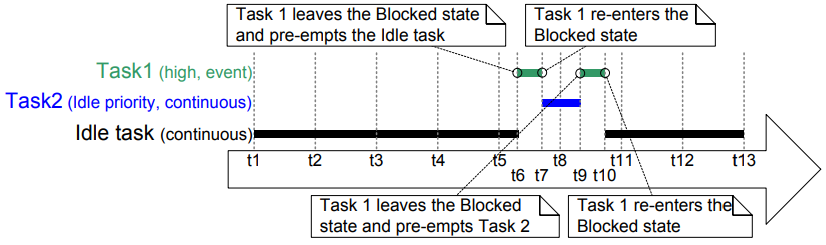
Hình 28: Mode 1



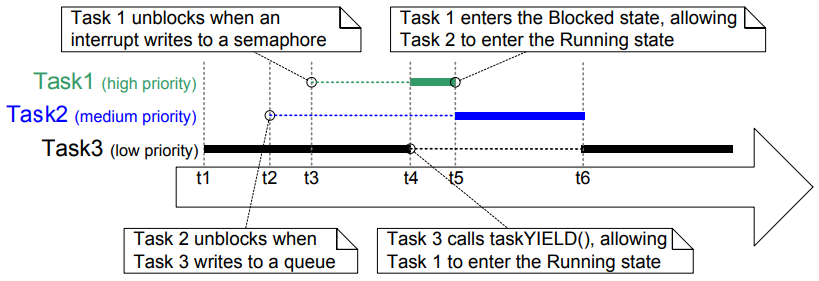
Hình 29: Mode 1 với 2 task có cùng priority



Hình 30: Mode 1 với 2 task có cùng priority nhưng cờ configIDLE\_SHOUD\_YIELD được set thành 1



Hình 31: Mode 2



Hình 32: Mode 3

### Quản lý Queue

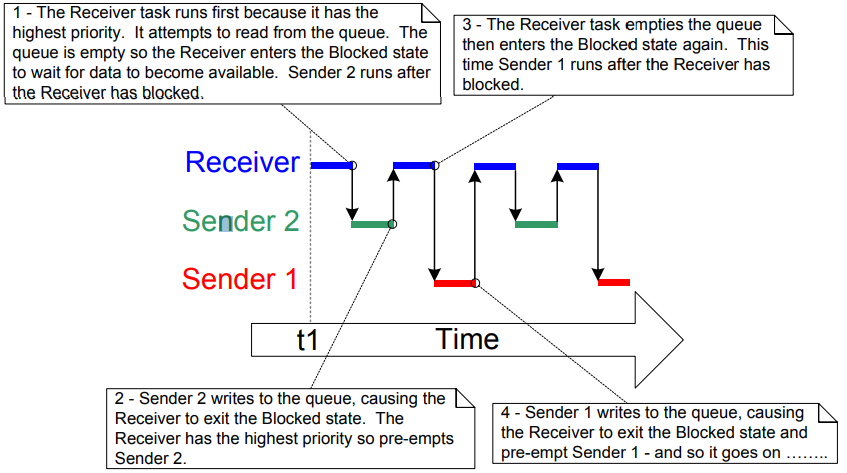
Queue trong FreeRTOS là cơ chế truyền thông giữa task với task, task với interrupt, và interrupt với task. Lập trình viên nên nắm được cách tạo queue, gửi nhận quản lý data với queue, block queue, ghi đè data trong queue, và xóa queue. Ngoài ra chúng ta nên hiểu được ảnh hưởng của việc đọc ghi queue lên độ ưu tiên của task. Trong phần này, chỉ có queue sử dụng cho task với task được trình bày.

Queue được access bởi tất cả task hay ISR nhìn thấy nó. Thông thường sẽ có nhiều writers và single reader. Khi queue trống, task muốn đọc queue sẽ nhảy vào blocked state đề chờ data. Nếu queue có multiple reader, có khả năng queue sẽ có nhiều task đang đợi data. Khi đó task có độ ưu tiên cao nhất sẽ nhảy ra khỏi blocked state. Ngược lại, nếu queue đang full và nhiều task đang muốn ghi vào queue, tất cả chúng sẽ nhảy vào blocked state và chỉ duy nhất task có độ ưu tiên cao nhất được phép chờ ghi vào vị trí trống sắp tới của queue. Nếu các task có cùng độ ưu tiên thì task đợi lâu nhất sẽ được unblock. Các queue có thể được nhóm thành từng nhóm nhỏ, điều này cho phép một task có thể chờ data từ bất kì queue nào trong nhóm.

Để tạo một queue, chúng ta dùng hàm xQueueCreate(). Trong đó uxQueueLength là số lượng phần tử tối đa mà queue có thể chứa. uxItemSize là số byte của mỗi phần tử trong queue. Hàm này trả về NULL cho trường hợp heap không chứa nổi và non-NULL trong trường hợp tạo queue thành công.

QueueHandle\_t xQueueCreate( UBaseType\_t uxQueueLength, UBaseType\_t uxItemSize );

Hàm xQueueReset() dùng để empty một queue. Hàm xQueueSendToBack() và xQueueSendToFront() dùng để ghi data vào tail và head của queue. Lưu ý trong ISR, chúng ta phải sử dụng cặp hàm xQueueSendToBackFromISR() và xQueueSendToFrontFromISR(). Để đọc giá trị từ queue, chúng ta sử dụng hàm xQueueReceive(). Trong ISR, chúng ta phải dùng hàm xQueueReceiveFromISR(). Hàm uxQueueMessagesWaiting() dùng để get số lượng message trong queue.



Hình 33: đọc ghi data tới queue

### Quản lý Software Timer

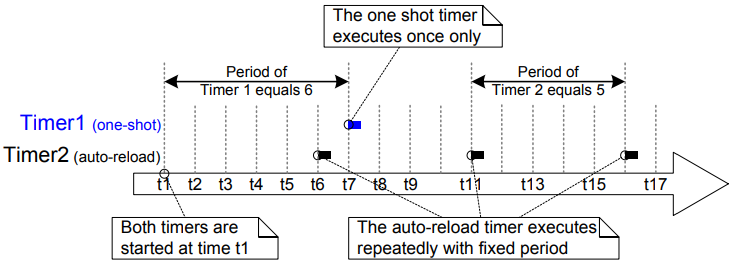
Software timer dùng để lập lịch thực thi cho một hàm được set sẵn ở một thời điểm trong tương lai hoặc tuần hoàn theo một chu kì nhất định. Hàm được thực thi bởi software timer được gọi là software timer callback function. Software timer không tốn processing time, hardware counter, hardware timer. Để sử dụng, chúng ta include FreeRTOS/Source/timers.c và set cờ configUSE\_TIMERS trong FreeRTOSConfig.h lên 1. Trong phần này, lập trình viên cần hiểu được sự khác nhau giữa software timer và task, RTOS daemon task, timer command queue, sự khác nhau giữa one shot software timer và periodic software timer, và cách sử dụng (create, start, reset, change period).

Các hàm software timer là các hàm C theo dạng API như sau:

void ATimerCallback( TimerHandle\_t xTimer );

Các hàm này nên ngắn gọn và không được chuyển về blocked state. Lưu ý, các hàm software timer không được gọi FreeRTOS API có thể dẫn đến việc task đang chạy bị block.

Đối với one-shot timer, hàm chỉ được chạy một lần và cần được restart nếu muốn chạy tiếp. Trong khi đó periodic timer sẽ tự restart (Hình34).

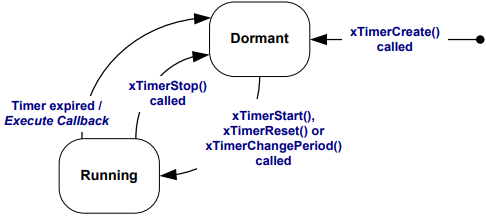
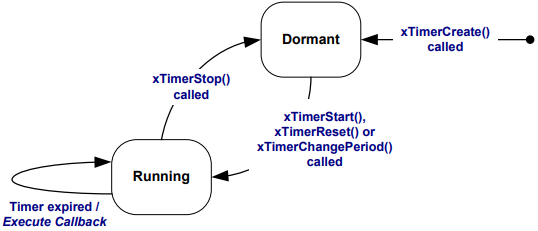


Hình 34: Sự khác nhau giữa one-shot và periodic software timer.

Một software timer có thể ở một trong hai trạng thái:

Dormant: Ở trạng thái này, software timer đang tồn tại vẫn có thể trược gọi bởi handle của nó nhưng không chạy nên callback function của nó cũng không chạy.

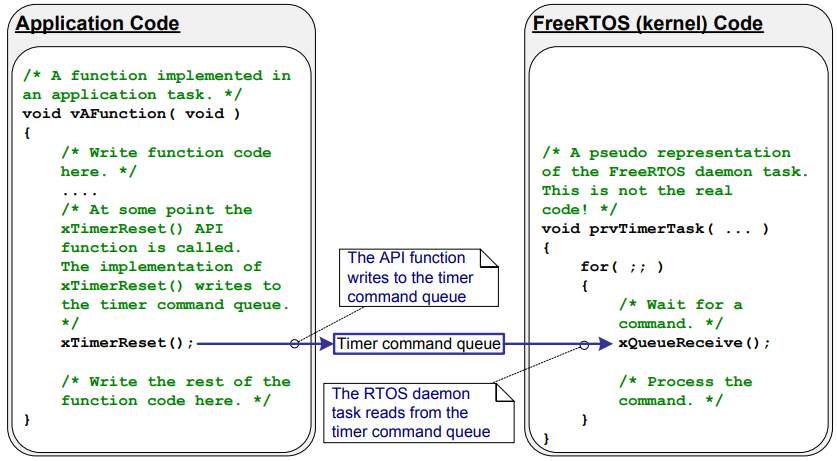
Running: Ở trạng thái này, software timer sẽ thực thi hàm callback của nó sau mỗi chu kì.



Hình 35: Các hàm chuyển đổi trạng thái cho periodic (trái) và one-shot (phải)

Tất cả software timer callback function đều được thực thi trong cùng một RTOS daemon task. Đây là một task tiêu chuẩn của FreeRTOS được tạo tự động khi scheduler bắt đầu. Độ ưu tiên và kích thước stack được set bởi configTIMER\_TASK\_PRIORITY và configTIMER\_TASK\_STACK\_DEPTH trong FreeRTOSConfig.h.

Timer command queue là một queue tiêu chuẩn của FreeRTOS được tạo tự động khi scheduler bắt đầu. Độ dài của queue được quyết định bởi configTIMER\_QUEUE\_LENGTH trong FreeRTOSConfig.h. Software timer API function sẽ gửi command (Ví dụ: start a timer, stop a timer, reset a timer) từ task gọi tới daemon task bởi queue này (Hình 36).



Hình 36: Software timer gửi command tới daemon task

Deamon task sẽ được lập lịch giống như những task khác của FreeRTOS. Nó chỉ xử lý command và thực thi hàm timer callback khi không có task nào có độ ưu tiên cao hơn nó sẵn sàng chạy.

Từ version 9.0 trở đi, FreeRTOS hỗ trợ hàm xTimerCreateStatic() để cấp phát tĩnh vùng nhớ cho timer ở thời điểm compile. Software timer phải được tạo để sử dụng (có thể tạo trước khi scheduler chạy).

TimerHandle\_t xTimerCreate(const char \* const pcTimerName,

TickType\_t xTimerPeriodInTicks,

UBaseType\_t uxAutoReload,

void \* pvTimerID,

TimerCallbackFunction\_t pxCallbackFunction );

Hàm xTimerStart() dung để bắt đầu một software timer đang ở trong trạng thái dormant hoặc dùng để reset một software timer đang ở trong trạng thái running.

BaseType\_t xTimerStart( TimerHandle\_t xTimer, TickType\_t xTicksToWait );

Mỗi một software timer có một ID riêng và được set bởi hàm vTimerSetTimerID(). Để get ID chúng ta dùng pvTimerGetTimerID(). Hai hàm này get trực tiếp, không gửi command vào timer command queue.

void vTimerSetTimerID( const TimerHandle\_t xTimer, void \*pvNewID );

void \*pvTimerGetTimerID( TimerHandle\_t xTimer );

Để thay đổi chu kì, chúng ta dùng hàm xTimerChangePeriod(). Dù cho đang ở trong trạng thái nào đi nữa thì nó cũng sẽ tự tính expiry time mới dựa trên chu kì mới. Hàm này có hỗ trợ version ISR-safe.

### Quản lý Ngắt

Độ ưu tiên của task hoàn toàn không liên quan tới phần cứng (do application cấp và cho chạy bởi scheduler) nhưng độ ưu tiên của interrupt thì có. ISR là một tính năng phần cứng nên phần cứng sẽ điều khiển ISR nào được chạy. Task chỉ được chạy khi không có ISR nào chạy. Trong phần này lập trình viên cần nắm được: các FreeRTOS API có thể gọi tử ISR, các hoãn ISR để task chạy, cách tạo và dùng semaphore, khác nhau giữa binary và counting semaphore, cách dùng queue để truyền nhận dữ liệu với ISR.

API đưa task về blocked state không gọi được trong ISR, bởi vì không có task gọi, do đó không có task nào được đưa về blocked state cả. Để giải quyết điều này, FreeRTOS tạo 2 version khác nhau cho một số API (một cho việc gọi từ task và một cho gọi từ ISR). Phiên bản thứ 2 được thêm hậu tố fromISR.

Nếu context switch do interrupt gây ra, task chạy sau khi interrupt kết thúc có thể sẽ khác task chạy trước khi nhảy vào interrupt (interrupt ngắt task này nhưng trả về task khác). Một số FreeRTOS API có thể chuyển task từ blocked qua ready ví dụ như xQueueSendToBack() mở khóa một task đang đợi trong trạng thái block để chờ data đến queue. Nếu độ ưu tiên của một task được mở khóa bởi một FreeRTOS API nào đó cao hơn độ ưu tiên của task đang chạy thì scheduler sẽ chuyển qua task này. Tuy nhiên điều này lại bị ảnh hưởng bởi một số trạng thái của nơi mà API đó được gọi:

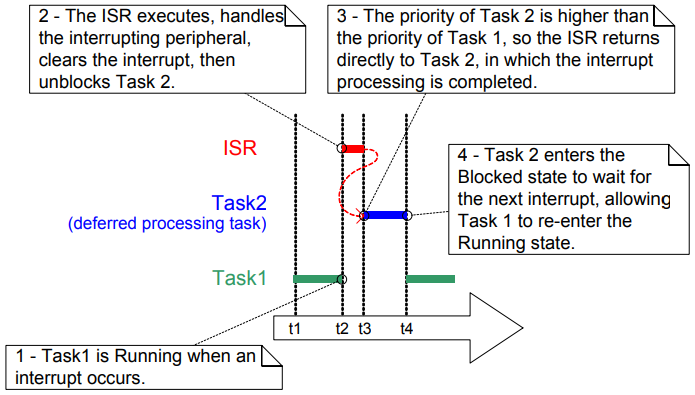
* + API được gọi từ một task. Nếu configUSE\_PREEMTION là 1 thì sẽ tự động nhảy qua task có độ ưu tiên cao hơn.
  + API được gọi từ ISR. Việc nhảy này có tự động hay không sẽ phụ thuộc vào pxHigherPriorityTaskWoken có được set lên pdTRUE hay không. Nếu không sẽ chờ ở trạng thái ready cho tới khi scheduler chạy tiếp hoặc tệ nhất là tick interrupt tiếp theo. Lý do mà việc nhảy này không tự động trong ISR là để tránh việc nhảy không cần thiết (ví dụ task xử lý string nhận từ UART interrupt, việc nhảy từ UART ISR qua task mỗi lần có ký tự được nhập vì task chỉ cần xử lý khi lấy xong một string), …

Chúng ta có thể dùng taskYIELD() để yêu cầu context switch từ một task. Để yêu cầu context switch từ một ISR, chúng ta dùng portYIELD\_FROM\_ISR() hoặc portEND\_SWITCHING\_ISR().

portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );

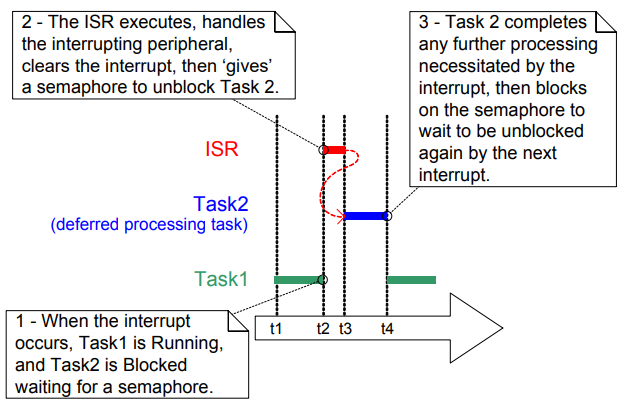
portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );

Khái niệm *deferred interrupt processing* chỉ việc các lệnh còn lại trong ISR được thực thi trong task (Hình 37). Điều này cho phép ISR kết thúc nhanh hơn. Chúng ta nên làm việc này khi những công việc còn lại của ISR khá cồng kềnh, hoặc không thể thực thi bên trong ISR (Ví dụ như ghi ra console, cấp phát bộ nhớ), hoặc khó biết trước được thời gian xử lý.

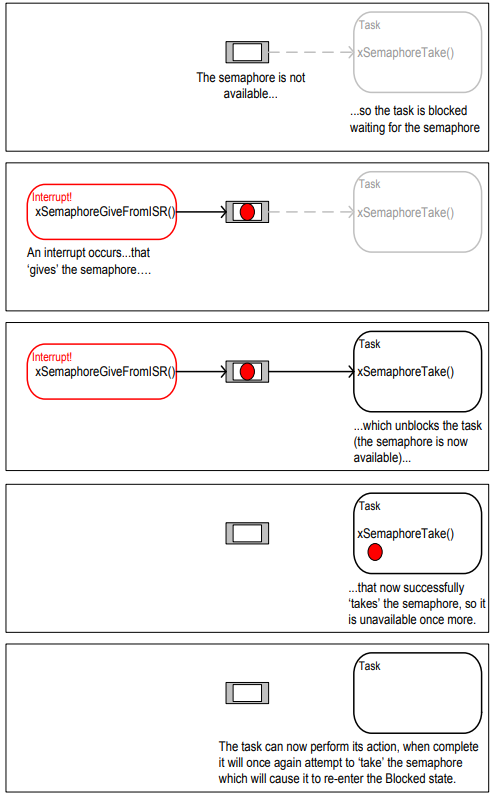


Hình 37: Task 2 có độ ưu tiên cao nhất sẽ hoàn thành công việc của ISR

Trong phần trước, chúng ta đã biết cách dùng binary semaphore để hoãn interrupt cho một task (Hình 38).



Hình 38: Sử dụng binary semaphore để implement deferred interrupt processing



Hình 39: Sử dụng binary semaphore để đồng bộ task với ngắt

(FreeRTOS từ V9.0 trở đi cho phép cấp phát tĩnh cho semaphore)

SemaphoreHandle\_t xSemaphoreCreateBinary( void ); Hàm này tạo binary semaphore.

BaseType\_t xSemaphoreTake( SemaphoreHandle\_t xSemaphore, TickType\_t xTicksToWait ); Hàm này không được dùng ở ISR.

BaseType\_t xSemaphoreGiveFromISR( SemaphoreHandle\_t xSemaphore, BaseType\_t \*pxHigherPriorityTaskWoken );

### Event group

Ở những phần trước, chúng ta đã biết cách FreeRTOS cho phép event tương tác với task Ví dụ dùng queue hoặc dùng semaphore cho phép một task chờ một event xảy ra ở blocked state và unblock một task khi một event xảy ra. Event group là một tính năng khác của FreeRTOS cho phép even tương tác với task theo một cách khác. Task có thể đợi một tổ hợp nhiều event xảy ra và unblock tất cả các task cùng chờ một/một nhóm event khi event đó xảy ra.

Tính năng này hữu ích khi đồng bộ nhiều task, truyền thông event cho nhiều task, cho phép task đợi một trong số các event xảy ra, và cho phép task đợi một số hành động được hoàn tất. RAM có thể được giảm xuống nhờ thay thế nhiều binary semaphore thành một event group.

EventGroupHandle\_t xEventGroupCreate( void ); tạo event group, trả về handle.

EventBits\_t xEventGroupSetBits( EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToSet ); Hàm này dùng để set bit để thông báo cho task tương ứng với bit được set.

EventBits\_t xEventGroupWaitBits( const EventGroupHandle\_t xEventGroup, const EventBits\_t uxBitsToWaitFor, const BaseType\_t xClearOnExit, const BaseType\_t xWaitForAllBits, TickType\_t xTicksToWait ); Hàm này cho phép một task đọc giá trị từ event group và chờ một/nhiều event bit được set.

### Truyền thông Task

Như chúng ta đã biết, các task trong FreeRTOS là hoàn toàn đọc lập. Để tương tác giữa các task, chúng ta dùng tính năng truyền thông task. Các phương pháp trước đây chúng ta sử dụng là tạo một communication object nào đó ví dụ như queue, event group, semaphore,... Khi sử dụng những phương pháp này, event và data không được gửi trực tiếp tới task và ISR. Truyền thông task là cách truyền trực tiếp giữa các task và để đồng bộ với ISR mà không cần semaphore. Bằng cách này, task hoặc ISR sẽ gửi event trực tiếp tới task nhận. Để sử dụng chúng ta bật cờ configUSE\_TASK\_NOTIFICATION. Lúc này mỗi task sẽ có một trạng thái thông báo (pending/not-pending) và một giá trị thông báo (tất cả là uint32\_t). Khi một task nhận thông báo, trạng thái thông báo sẽ được chuyển thành pending. Sau khi nó đọc thông báo này, trạng thái sẽ chuyển qua thành not-pending.

Sử dụng phương pháp này sẽ nhanh hơn queue, semaphore và event group, sử dụng ít RAM hơn (chỉ tốn 8 bytes overhead trên mỗi task). Bù lại, chúng ta không dùng được cách này trong các trường hợp:

* + Gửi data/event tới ISR (Chỉ gửi task với task hoặc ISR qua task)
  + Nhiều task nhận (communication object có thể được truy xất bởi nhiều task/ISR còn truyền thông task thì không)
  + Buffer data
  + Truyền thông tới nhiều task
  + Đang đợi ở blocked state để gửi để hoàn thành (nếu một task đang ở pending, task kia không thể đợi ở blocked state cho tới khi task này reset trạng thái thông báo của nó)

BaseType\_t xTaskNotifyGive( TaskHandle\_t xTaskToNotify ); Hàm này gửi thông báo tới một task khác và tăng một vào giá trị thông báo của task nhận.

uint32\_t ulTaskNotifyTake( BaseType\_t xClearCountOnExit, TickType\_t xTicksToWait ); Hàm này cho phép một task chờ ở blocked state tới khi nào giá trị thông báo của nó lớn 0.

### Tính năng mở rộng

Ở phần này, FreeRTOS cung cấp một loạt các hàm giúp chúng ta có thể hiểu được ứng dụng chúng ta có hành vi như thế nào, tìm ra những cơ hội để tối ưu và đặt bẫy những lỗi có thể xảy ra.

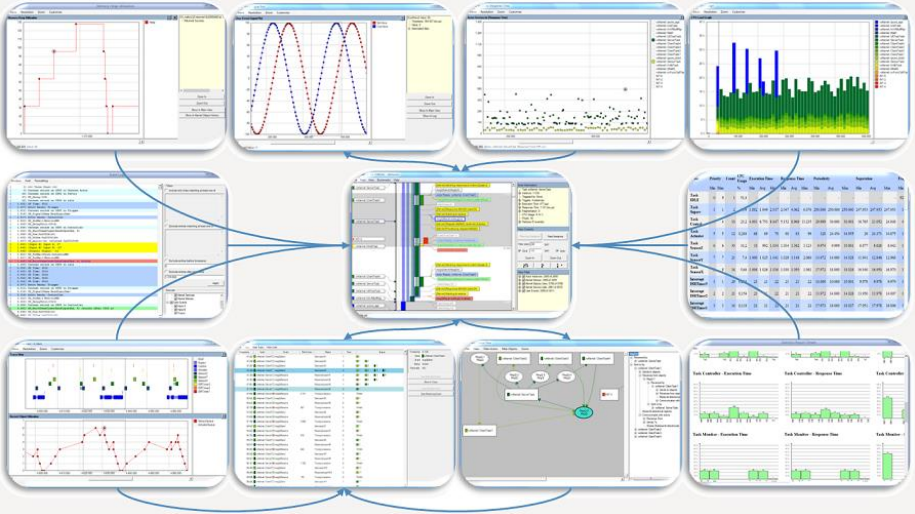
Hàm assert dùng để kiểm tra một giả sử nào đó trong quá trình debug. Khi release, chúng ta comment macro configASSERT() để giảm code size. Hàm này được khuyến khích dùng càng nhiều càng tốt. Khi assumption trong hàm này fail thì chương trình sẽ ngưng ngay tại đó. Hàm assert này được define trong FreeRTOSconfig.h như sau:

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE\_INTERRUPTS(); for(;;); }

Khi không debug nữa, chúng ta dùng định nghĩa sau để lưu lại những assumption fail trong quá trình chạy.

#define configASSERT( x ) if( ( x ) == 0 ) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

Phần mềm FreeRTOS+Trace được dùng để thu thập và phân tích các chương trình chạy FreeRTOS rồi biểu diễn dưới dạng đồ thị (Hình 40).



Hình 40: Các chức năng phân tích của FreeRTOS+Trace được đồ thị hóa

Hàm malloc failed hook thuộc phần quản lý Heap. Hàm stack overflow hook đảm bảo chúng ta sẽ được thông báo nếu tràn stack của một task nào đó:

UBaseType\_t uxTaskGetStackHighWaterMark( TaskHandle\_t xTask );

FreeRTOS có tích hợp hai cơ chế kiểm tra tràn stack được quy định bởi macro configCHECK\_FOR\_STACK\_OVERFLOW. Giá trị cờ này có thể 1 hoặc 2 để quy định cách define hàm:

void vApplicationStackOverflowHook( TaskHandle\_t \*pxTask, signed char \*pcTaskName );

Để thu thập thời gian chạy, chúng ta dùng các macro sau:

* + configGENERATE\_RUN\_TIME\_STATS : Nếu cờ này được bật, scheduler sẽ gọi các macro còn lại vào thời điểm thích hợp nhất.
  + portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() : Macro này phải được dùng để khởi tạo ngoại vi được dùng để cung cấp run-time statistic clock
  + portGET\_RUN\_TIME\_COUNTER\_VALUE() hoặc portALT\_GET\_RUN\_TIME\_COUNTER\_VALUE(Time) : Hai macro này phải được dùng để trả về giá trị clock. Đây là tổng thời gian ứng dụng đã được chạy.

Để thu thập thông tin trạng thái của các task đang được điều khiển bởi scheduler, chúng ta dùng hàm:

UBaseType\_t uxTaskGetSystemState( TaskStatus\_t \* const pxTaskStatusArray, const UBaseType\_t uxArraySize, uint32\_t \* const pulTotalRunTime );

Để hiển thị dưới dạng bảng đọc được, chúng ta sử dụng hàm:

void vTaskList( signed char \*pcWriteBuffer );

Để hiện thị thười gian chạy của các task (Hình 41), chúng ta dùng hàm vTaskGetRunTimeStats(). Hàm này xâm lấn code rất nhiều nên chỉ dùng khi debug.

void vTaskGetRunTimeStats( signed char \*pcWriteBuffer );



Hình 41: Kết quả của hàm vTaskGetRunTimeStats()

Một số macro khác:

traceTASK\_INCREMENT\_TICK(xTickCount)

traceTASK\_SWITCHED\_OUT()

traceTASK\_SWITCHED\_IN()

traceBLOCKING\_ON\_QUEUE\_RECEIVE(pxQueue)

traceBLOCKING\_ON\_QUEUE\_SEND(pxQueue)

traceQUEUE\_SEND(pxQueue)

# Tìm hiểu dự án S-Tracking

## Tổng quan dự án

S-TRACKING là thiết bị giám sát hành trình tàu cá có sử dụng vệ tinh và GPRS. Thiết bị hoạt động một cách độc lập với nguồn điện tàu nhờ tấm pin năng lượng mặt trời. Thiết bị được trang bị GPS, GPRS, module Iridium nhằm thực hiện chức năng Tracking, cảnh báo, lưu trữ nhật kí, nhắn tin về bờ.

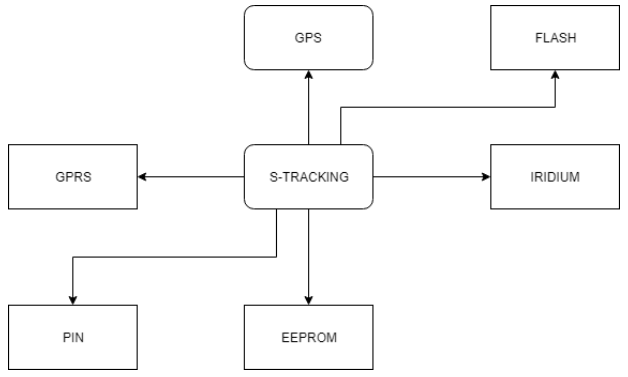
S-ALARM là thiết bị cho phép lấy dữ liệu về PIN, tin nhắn từ server, thông tin vượt biên từ thiết bị S-TRACKING qua chuẩn giao tiếp Bluetooth. Ngoài ra thiết bị còn cho phép người dùng yêu cầu gửi tin SOS lên server thông qua vệ tinh của outdoor.

Dự án S-Tracking ra đời trong bối cảnh từ tháng 10/2017, thủy sản Việt Nam bị Ủy ban châu Âu áp dụng “thẻ vàng” và có nguy cơ dẫn tới “thẻ đỏ” nếu tình trạng dư dân vi phạm hải phận các nước khác vẫn còn tiếp diễn. Trước đó, giải pháp chống “thẻ vàng” hay “thẻ đỏ” với các thiết bị định vị mua của nước ngoài cho các tàu cá bị vô hiệu bởi nhiều lý do. Hai dự án được tài trợ với các thiết bị của Nhật và Pháp hầu hết đều bị ngư dân tắt nguồn hoặc rút ăng ten để tùy nghi đánh bắt và bản thân thiết bị cũng có những hạn chế. Đứng trước nguy cơ mất đi 20% thị phần xuất khẩu thủy sản, sản phẩm S-Tracking của Viettel sẽ là lời giải cho lĩnh vực đánh bắt thủy sản của Việt Nam chúng ta hiện nay.

S-tracking được thiết kế để hạn chế sự bất hợp tác của người dân: tất cả mọi linh kiện, kể cả ăng ten, đều được đặt trong một hộp kín chống nước, chống bụi và sử dụng ốc vít rất khó mở, thiết bị sử dụng nguồn năng lượng mặt trời và kể cả khi đặt trong bóng tối cũng có pin dự trữ có thể duy trì thiết bị trong 30 ngày – bằng thời gian một chuyến biển. Cứ 90 phút một lần, S-tracking lại tự động gửi tọa độ của tàu về cho các cơ quan chức năng. Trên màn hình của các cơ quan quản lý, Viettel thiết kế phần mềm để họ có thể thấy bản đồ vị trí của tất cả các tàu và nhấp chuột vào tàu nào trên màn hình thì thông tin của tàu đó lại hiện ra và sẽ có còi báo hiệu khi tàu vượt hải phận hoặc khi thiết bị sắp cạn nguồn. Sản phẩm của Viettel cũng ra đời vào “đúng thời điểm” tỉnh Cà Mau đưa ra một quyết định cứng rắn: tàu nào không lắp thiết bị giám sát hành trình sẽ không cho ra khơi hoặc đăng ký, đăng kiểm. Cũng không có gì ngạc nhiên khi S-tracking ra đời từ lời đặt hàng của Viettel Cà Mau đồng thời thử nghiệm ở tỉnh này đầu tiên, với 300 thiết bị đã được bán ra.

## Thiết kế thiết bị S-Tracking (outdoor)

### Tổng quan thiết kế



Hình 42: Sơ đồ tổng quát thiết bị outdoor

Thiết bị outdoor sử dụng dòng chip Bluetooth NRF51822 của hãng Nordic với lõi vi điều khiển ARM-Cortex M0 và phần mạch RF Bluetooth. Các ngoại vi trên thiết bị bao gồm:

- GPS L80 của Quectel.

- Module GSM/GPRS M95 của Quectel.

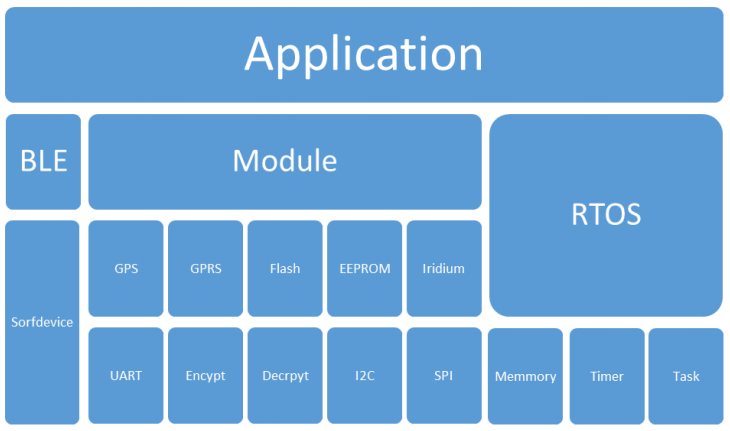
- Flash ngoại W25Q128JV.

- EEPROM CAT24C512WI.

- Module vệ tinh Iridium 9603.

- Pin năng lượng mặt trời, pin sạc, pin không sạc.

Phần mềm thiết bị theo dõi tàu cá (S-tracking) có kiến trúc sử dụng gồm ba tầng, trong đó, RTOS sử dụng mã nguồn mở freeRTOS và stack BLE sử dụng softDevice của hãng Nordic. Thiết bị còn sử dụng thêm module ngoài như GPS, GPRS, Iridium, Flash ngoại, EEPROM, mỗi module sẽ được tổ chức thành 1 phần riêng biệt.



Hình 43: Kiến trúc phần mềm

Nền tảng hardware dựa trên chip chính là nrf51822 của Nordic, chip là 1 SoC nhỏ gọn, tích hợp giữa lõi ARM cotex M0, một số ngoại vi cơ bản, 256Kbytes Flash nội (ROM) và 32Kbytes RAM, cùng phần cứng RF tích hợp bên trong tạo thành chức năng BLE đi kèm. Các module đi kèm như GPS, GPRS, Iridium được kết nối với chip chính và giao tiếp thông qua UART. Để khắc phục nhược điểm hạn chế về bộ nhớ chip được gắn thêm 1 chip. EEPROM, 1 Flash ngoài. Thiết bị cũng sử dụng nguồn điện pin và năng lương mặt trời, nên 1 trong các Hạn chế về bộ nhớ (256Kbytes Flash + 32Kbytes RAM) và phải chứa thêm 2 phân vùng

Bootloader và Softdevice (sẽ giải thích kĩ hơn ở phần sau). Dẫn đến kiến trúc phần mềm của thiết bị cần phải được tối ưu triệt để nhằm có khối lượng nhỏ nhất có thể. Để dễ bề quản lý và phát triển, phần mềm sử dụng thêm freeRTOS, hệ điều hành thời gian thực miễn phí và phổ biến nhất hiện nay.

### Chi tiết API

Bảng 37: UART driver

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Input | Detail |
| void | init | uartDriverClass\_t \* | Khởi động module, liên kết các object, cài đặt giá trị khởi đầu. |
| bool | changePort | uartDriverClass\_t \* portName | Thay đổi port của UART, các port là M95,9603 và GPS. (có thể mở rộng nhờ file myConfig.h) |
| bool | putChar | uartDriverClass\_t \* const char | Truyền 1 char qua uart. |
| bool | putDatas | uartDriverClass\_t \* const char \* uint16\_t | Truyền chuỗi data với độ dài biết trước. |
| bool | putStringUntil | uartDriverClass\_t \* const char \* uint8\_t | Truyền chuỗi data đến khi gặp kí tự kết thúc. |
| bool | getData | uartDriverClass\_t \*uint8\_t \*uint16\_tbool | Nhận tín hiệu từ chân Rx, data  được chứa ở trong  (uartDriverClass\_t \*)->RxBuffer, index chạy từ  (uartDriverClass\_t \*)->pRxRead đến  (uartDriverClass\_t \*)->pRxWrite. |

Bảng 38: SPI driver

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Input | Detail |
| void | init | SPIDriverClass\_t\* | Khởi động module, liên kết các  object, cài đặt giá trị khởi đầu. |
| bool | spiTransfer | SPIDriverClass\_t\* const char \*data uint32\_t length | Truyền data và nhận data, chưa data mới tại  (SPIDriverClass\_t\*)->rxBuffer. |
| bool | spiTransferCMD | SPIDriverClass\_t const char \* uint16\_t const char \* uint32\_t | Truyền data gồm CMD và dữ liệu thô, CMD và chiều dài ứng với prag 2-3, data thô và chiều dài ứng với prag 4-5 |
| bool | enableCS | void | Chân CS ứng với eeprom xuống mức thấp. |
| bool | disableCS | void | Chân CS ứng với eeprom lên mức cao. |
| byte | spiWriteByte | SPIDriverClass\_tuint8\_t | Truyền 1 byte và nhận lại 1 byte |

Bảng 39: I2C driver

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Return | Name | Input | Detail | |
| void | init | I2CDriverClass \_t\* | | Khởi động module, liên kết các object, cài đặt giá trị khởi đầu. |
| bool | transferI2C | I2CDriverClass\_t \* uint8\_t uint8\_t \* uint8\_t bool | | Truyền data, dựa vào address (tham số thứ 2) để biết là đọc hay ghi. Data sau khi nhận nằm trong  Vùng nhớ con trỏ (tham số thứ 3) đang trỏ vào. |
| bool | writeI2C | I2CDriverClass \_t const char \* uint16\_t const char \* uint32\_t | | Ghi data |
| bool | readI2C | I2CDriverClass \_t const char \* uint16\_t const char \* uint32\_t | | Đọc data |

Bảng 40: GPS API

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Input | Detail |
| void | init | GPSClass\_t \* uartDriverClass\_t \* | Set up module, cài đặc thông số ban đầu, kết nối uart cho module (Module có thể tải sử dụng dễ dàng hơn nếu đổi sang chip khác, ta cần viết lại các API driver uart là được). |
| void | enableGPS | GPSClass\_t \* | Bật nguồn cấp cho GPS (Để GPS hoạt đồng cần cấp nguồn Vin và nguồn Vbat, xem sche để |
|  |  |  | hiểu chi tiết) |
| void | disableGPS | GPSClass\_t \* | Tắt nguồn cấp cho GPS |
| bool | gpsToStandbyMode | GPSClass\_t \* | Đưa GPS standby mode |
| bool | gpsToBackUpMode | GPSClass\_t \* | Đưa GPS vào backup mode |
| bool | gpsHotStart | GPSClass\_t \* | Hot start GPS |
| bool | GpsWarmStart | GPSClass\_t \* | Warm start GPS |
| bool | gpsColdStart | GPSClass\_t \* | Cold start GPS |
| bool | configNMEA | GPSClass\_t \* | Giảm các gói tin NMEA, hàm này được viết với thông số cứng. |
| bool | updateDataGPS | GPSClass\_t \* | Lấy thông tin từ GPS, cho đến khi nhận được data có ý nghĩa.  Thông số về Latitude  Longtitude được cập nhật trong  GPSClass\_t \* |
| void | onBatPWR | GPSClass\_t \* | Bật nguồn vBat(xem sche) |
| void | offBatPWR | GPSClass\_t \* | Tắt nguồn vBat |

Bảng 41: Iridium API

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| return | | name | input | Detail |
| void | | Init | IRIDIUMClass\_t \* uartDriverClass\_t \* | Cài đặc thông số, GPIO.  Kết nối uart. |
| void | | on9603 | IRIDIUMClass\_t \* | Bật chân on/off trên module |
| void | off9603 | | IRIDIUMClass\_t \* | Tắt chân on/off trên module |
| void | onPower | | IRIDIUMClass\_t \* | Bật nguồn nuôi module  (nguồn cấp module và SuperCap). |
| void | offPower | | IRIDIUMClass\_t \* | Tắt nguồn nuôi module ( nguồn cấp module và  SuperCap ). |
| bool | getIMEI | | IRIDIUMClass\_t \* | Cập nhật IMEI thiết bị,  IMEI mới cập nhật nằm trong IRIDIUMClass\_t |
| bool | writeStringToISU | | IRIDIUMClass\_t \*  uint8\_t \* | Đẩy data vào buffer ISU, data sẵn sàng để chuyển thông qua  sendMOandGetMT |
| bool | writeBinaryToISU | | IRIDIUMClass\_t \*  uint8\_t \* uint32\_t | Đẩy data vào buffer ISU, data có độ dài biết trước. |
| bool | sendMOandGetMT | | IRIDIUMClass\_t \* | Data trong buffer ISU sẽ được chuyển lên vệ tinh và nếu có data gửi về, nó sẽ được chưa trong buffer ISU. |
| bool | getMessageFromSat  ellite | | IRIDIUMClass\_t \* | Lấy gói tin trong ISU và chưa trong  IRIDIUM\_Rxbuf |
| bool | getBinaryFromSatell  ite | | IRIDIUMClass\_t \* | Lấy gói tin trong ISU  (dạng binary ) chứa trong  IRIDIUM\_Rxbuf |
| bool | getStatusExtended | | IRIDIUMClass\_t \* | Đọc status thiết bị để biết còn bao nhiêu gói tin đang chờ, MT’s buffer trống hay không. |
| bool | resignSatellite | | IRIDIUMClass\_t \* | Thực hiện lên  AT+SBDREG để đk thiết bị vào mạng, khi có gói tin mới, vệ tinh có thể cảnh báo thống qua RI. |
| bool | detachSatellite | | IRIDIUMClass\_t \* | Thực hiện lên  AT+DETACH để thoát khỏi mạng. Ngược lại với  resignSatellite |
| bool | sendCommandString  Return | | IRIDIUMClass\_t \* const char \* const char \* uint32\_t | Gửi lệnh module vệ tinh,  thông tin trả về là String |
| bool | sendCommandBinar uReturn | | IRIDIUMClass\_t \* const char \* const char \* uint32\_t | Gửi lệnh module vệ tinh,  thông tin trả về là binary |
| void | onLinear | | IRIDIUMClass\_t \* | Bật module linear |
| void | offLinear | | IRIDIUMClass\_t \* | Tắt module linear |

Bảng 42: M95 API

|  |  |  |  |
| --- | --- | --- | --- |
| void | init | M95Class\_t \* uartDriverClass\_t \* | Kết nối M95, kết nối uart |
| void | turnPowerON | M95Class\_t \* | Mở nguồn cấp |
| void | turnPowerOFF | M95Class\_t \* | Tắt nguồn cấp |
| void | pwrKeyON | M95Class\_t \* | Mở nguồn powerKey |
| void | pwrKeyOFF | M95Class\_t \* | Tắt nguồn powerKey |
| bool | sendCommand | M95Class\_t \* const char \* uint32\_t const char \* const char \* uint32\_t | Gửi AT command  Đối số 1: lệnh  Đối số 2: độ dài lệnh  Sau khi gửi lệnh, hàm sẽ chờ để nhận được phản hồi từ module, hàm sẽ chờ đến khi gặp head và tail  Đối số 3: head  Đối số 4: tail  Đối số 5:thời gian timeout theo ms. |
| bool | sendCommandUntil | M95Class\_t \* const char \* uint8\_t const char \* const char \* uint32\_t | Gửi AT command  Đối số 1: lệnh  Đối số 2: kí tự kết thúc lệnh  Sau khi gửi lệnh, hàm sẽ chờ để nhận được phản hồi từ module, hàm sẽ chờ đến khi gặp head và tail  Đối số 3: head  Đối số 4: tail  Đối số 5:thời gian timeout theo ms. |
| bool | sentCommandWait  Length | M95Class\_t \* const char \* uint32\_t const uint32\_t uint32\_t | Gửi AT command  Đối số 1: lệnh  Đối số 2: độ dài lệnh  Sau khi gửi lệnh, hàm sẽ chờ đến khi nhận được đủ data với độ dài biết trước  Đối số 3: độ dài  Đối số 4: thời gian timeout theo ms. |
| bool | sendCommandUntil  WaitLength | M95Class\_t \* const char \* uint8\_t const uint32\_t uint32\_t | Gửi AT command  Đối số 1: lệnh  Đối số 2: ký tự kết thúc lệnh  Sau khi gửi lệnh, hàm sẽ chờ đến khi nhận được đủ data với độ dài biết trước  Đối số 3: độ dài  Đối số 4: thời gian timeout theo ms. |
| bool | getIEMI | M95Class\_t \* | Lấy IMEI thiết bị |
| bool | isActive | M95Class\_t \* | Kiểm tra thiết bị mở hay chưa |
| bool | checkIsViettelSim | M95Class\_t \* | Kiểm tra Sim có phải vietel hay ko, nếu ko có Sim hay Sim khác viettel thì trả về false. |

Bảng 43: GPRS API

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Input | Detail |
| bool | init | GPRSClass\_t \*  M95Class\_t \* | Cài đặc các thông số ban đầu, trỏ tới đối tượng M95 |
| bool | openPort | GPRSClass\_t \* const char \* const char \* | Mở port với 2 thông số đầu vào là domain và port |
| bool | sendDataViaTCP | GPRSClass\_t \*  const char \* const char \* const char \* uint32\_t | Gửi data qua TCP với domain và port mở sẵn, gửi và chờ nhận lại data  Sau khi gửi lệnh, hàm sẽ chờ để nhận được phản hồi từ module, hàm sẽ chờ đến khi gặp head và tail  Đối số 3: head  Đối số 4: tail  Đối số 5:thời gian timeout theo ms. |
| bool | sentDataViaTCP waitLength | GPRSClass\_t \*  const char \* uint32\_t uint32\_t | Gửi data qua TCP với domain và port mở sẵn, gửi và chờ nhận lại data.  Sau khi gửi lệnh, hàm sẽ chờ đến khi nhận được đủ data với độ dài biết trước  Đối số 3: độ dài  Đối số 4: thời gian timeout theo ms. |
| bool | isPortOpen | GPRSClass\_t \* | Kiểm tra xem có port nào đang mở ko |
| bool | getRSSI | GPRSClass\_t \* | Cập nhật RSSI. |

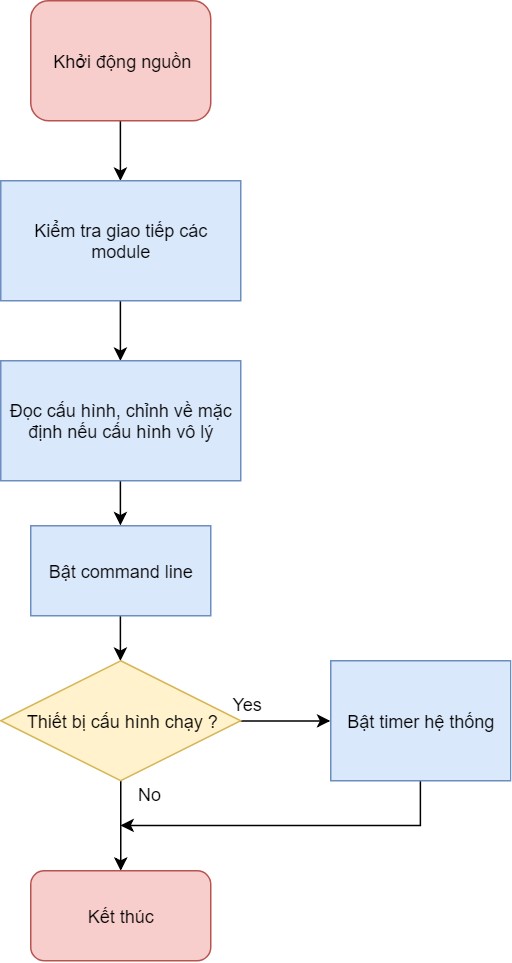
Bảng 44: Tracking API

|  |  |  |  |
| --- | --- | --- | --- |
| Return | Name | Input | Detail |
| void | init | TrackingClass\_t \*  EEPROMClass\_t \* | Cài đặc thông số ban đầu, link với eeprom |
| bool | savePoint | TrackingClass\_t \* uint32\_t vec int16\_t | Lưu 1 điểm trong đường biên |
| bool | savePoints | TrackingClass\_t \* uint32\_t vec \* int16\_t int16\_t | Lưu nhiều điểm trong đường biên |
| bool | readBorder | TrackingClass\_t \* uint32\_t vec \* | Lấy đường biên giới |
|  |  | int32\_t \* |  |
| bool | saveBorder | TrackingClass\_t \* uint32\_t vec int32\_t | Lưu đường biên giới |
| bool | isInside | TrackingClass\_t \*  Vec Vec \*  int | Kiểm tra có nằm trong hình bất kì hay ko |
| bool | isInside  3kmBorder | TrackingClass\_t \* vec | Kiểm tra có cách đường biên 3km hay ko |
| bool | isInside  Border | TrackingClass\_t \* vec | Kiểm tra có nằm trong đường biên giới hay ko |
| uint32\_t | getTime  ToTracking | TrackingClass\_t \* | Lấy thời gian Tracking |
| uint32\_t | getPERIOD | TrackingClass\_t \* | Lấy chu kì Tracking |
| bool | saveTime  ToTracking | TrackingClass\_t \* uint32\_t | Lưu thời điểm Tracking |
| bool | savePERIOD | TrackingClass\_t \* uint32\_t | Lưu chu kì Tracking |

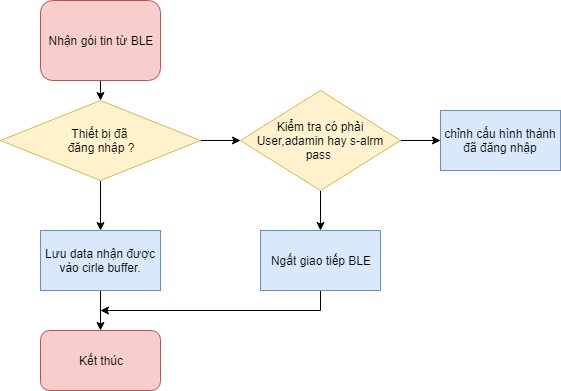
Bảng 45: User API

|  |  |  |
| --- | --- | --- |
| void | init | UserClass\_t \* |
| char\* | getAdminPassword | UserClass\_t \* |
| char\* | getUserPassword | UserClass\_t \* |
| char\* | getOutdoorPassword | UserClass\_t \* |
| char\* | getDeviceName | UserClass\_t \* |
| char\* | getVersion | UserClass\_t \* |
| char\* | getFOTADeviceKey | UserClass\_t \* |
| char\* | getSeriNumber | UserClass\_t \* |
| uint32\_t | getFOTA\_UTC | UserClass\_t \* |
| uint32\_t | getIDMax | UserClass\_t \* |
| char\* | getMessage | UserClass\_t \* |
| bool | setAdminPassword | UserClass\_t \* char \* |
| bool | setUserPassword | UserClass\_t \* char \* |
| bool | setOutdoorPassword | UserClass\_t \* char \* |
| bool | setDeviceName | UserClass\_t \* char \* |
| bool | saveVersion | UserClass\_t \* char \* |
| bool | saveFOTADeviceKey | UserClass\_t \* char \* |
| bool | saveSeriNumber | UserClass\_t \* char \* |
| bool | saveFOTA\_UTC | UserClass\_t \* char \* |
| bool | saveIDMax | UserClass\_t \* char \* |
| bool | saveMessageFromSat | UserClass\_t \* char \* |
| bool | getMode | UserClass\_t \* |
| bool | saveMode | UserClass\_t \* char \* |
| bool | getState | UserClass\_t \* |
| bool | saveState | UserClass\_t \* char \* |
| bool | saveIMEI | UserClass\_t \* char \* |
| char\* | getIMEI | UserClass\_t \* |

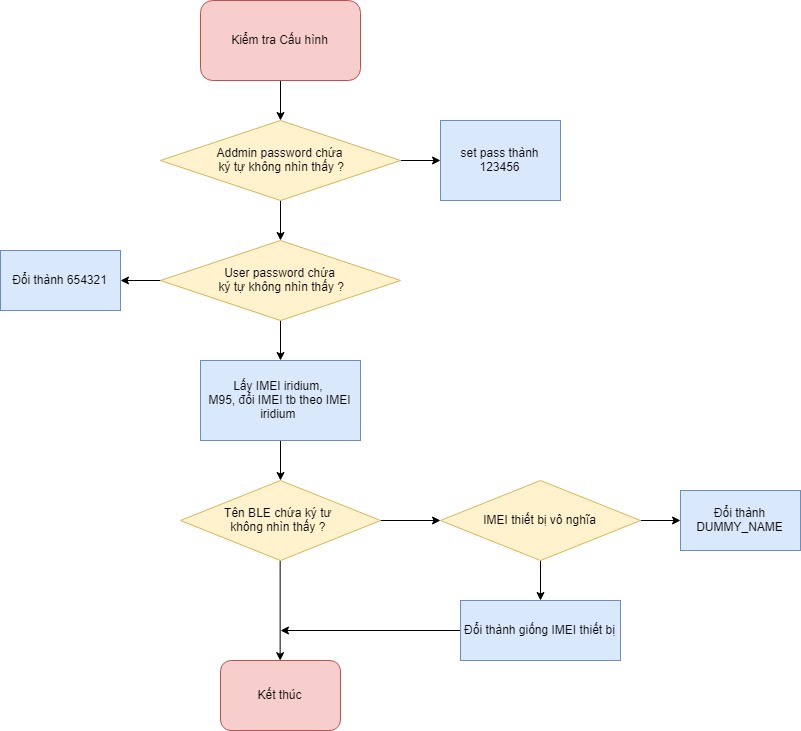
### Luồng xử lý



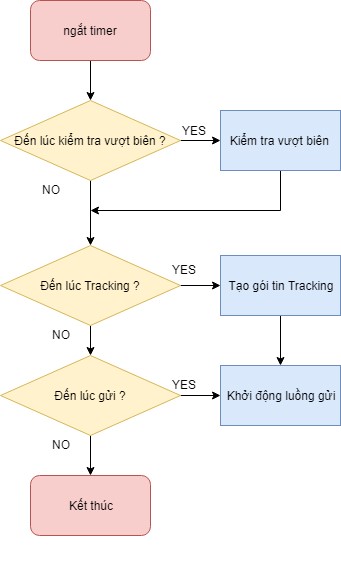
Hình 44: Luồng khởi động



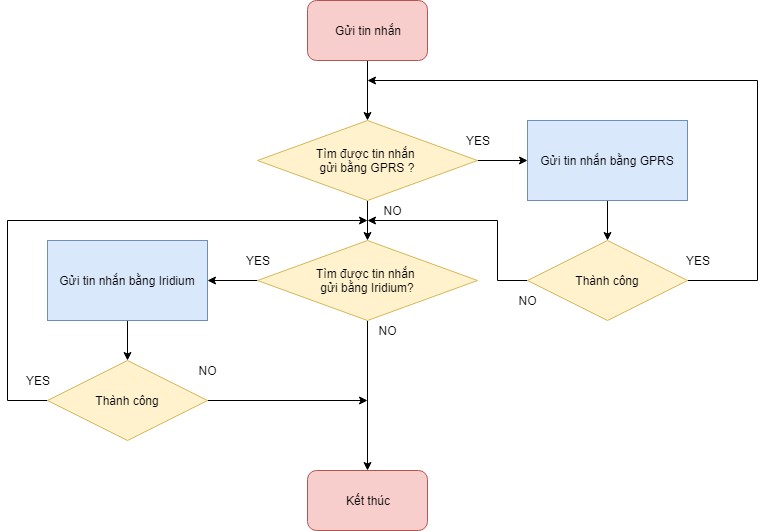
Hình 45: Luồng đăng nhập



Hình 46: Luồng kiểm tra phần cứng

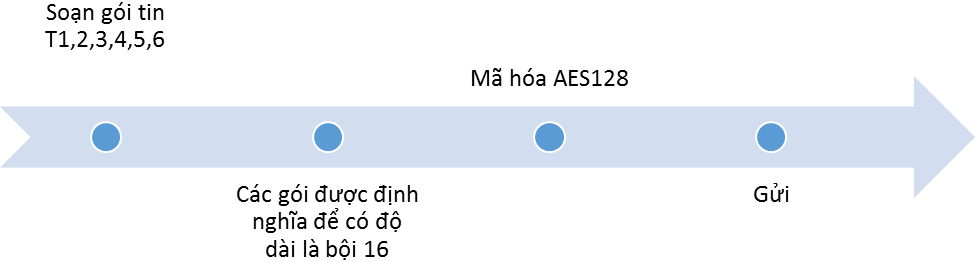


Hình 47: Luồng chạy chính



Hình 48: Luồng gửi thiết bị

### Mã hóa gói tin



Hình 49: Mã hóa gói tin

Bảng 46: Các gói tin

|  |  |  |
| --- | --- | --- |
| Gói tin | Độ dài | Chú thích |
| T1 | 18 bytes | Gói tin Tracking |
| T3 | 18 bytes | Gói tin khởi đầu chuyến đi |
| T4 | 34 bytes | Gói tin mẻ cá |
| T5 | 18 bytes | Gói tin kết thúc chuyến đi |
| T6 | 18 bytes -> 50 bytes | Gói tin người dùng |

## Các chế độ debug

Bộ tập lệnh này được viết dưới dạng command line giúp người lập trình, tester kiểm tra hoạt động của các module trên thiết bị, cũng như cấu hình các thông số ban đầu cho thiết bị.

Tập lệnh này được sử dụng trong hai chế độ là UART debug và BLE debug.

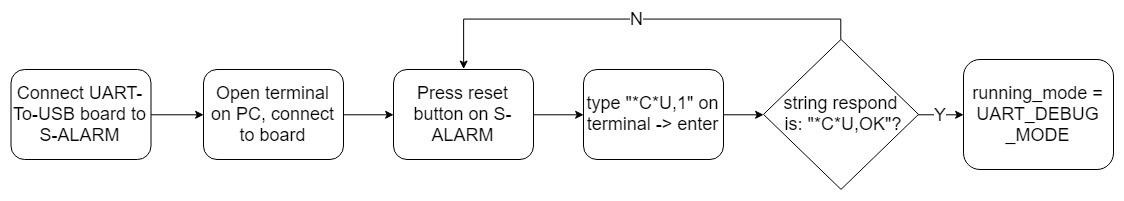
### UART debug

Dùng board UART-To-USB kết nối đến board S-ALARM với chân tương ứng như sau

Bảng 47: Kết nối chân giữa board UART-To-USB với S-ALARM

|  |  |
| --- | --- |
| Board UART-USB | S-ALARM |
| RX | P0.15 |
| TX | P0.14 |
| 3V3 | 3V3 |
| GND | GND |

Dùng một số phần mềm COM terminal như MobaxTerm, Teraterm,… để giao tiếp, chọn cấu hình giao tiếp: COM tương ứng, Baudrate: 115200, Data: 8 bit, Parity: None, Stop bit: 1 bit.



Hình 50: Lưu đồ giải thuật để vào chế độ UART debug

Bảng 48 tập lệnh được liệt kê dưới bảng sau:

Bảng 48: Tập lệnh hỗ trợ kiểm tra module và cấu hình thiết bị

|  |  |  |
| --- | --- | --- |
| Câu lệnh | Chức năng | Trả về |
| on\_led | Bật tất cả các LED | "ON LED OK" |
| off\_led | Tắt tất cả các LED | "OFF LED OK" |
| test\_led | Chớp tắt tất cả các  LED 3 lần chu kỳ 1s | "TEST LED OK" |
| on\_3v3 | Bật nguồn 3v3 | "POWER ON 3V3" |
| off\_3v3 | Tắt nguồn 3v3 | "POWER OFF 3V3" |
| on\_5v | Bật nguồn 5v | "POWER ON 5V0" |
| off\_5v | Tắt nguồn 5v | "POWER OFF 5V0" |
| enable\_amplifier | Cho phép khuyếch đại ngõ ra audio | "ENABLE AMPLIFIER OK" |
| disable\_amplifier | Tắt khuếch đại ngõ ra audio | "DISABLE AMPLIFIER OK" |
| test\_sine | Kiểm tra VS1011E với sóng sine 1KHz | "TEST SINE OK" |
| read\_vs | Đọc thanh ghi SCI\_MODE của  VS1011E | "READ VS1011E OK" "SCI\_MODE value: [ ]" |
| enable\_spi\_vs | Chọn VS1011E là thiết bị giao tiếp  SPI | "ENABLE SPI FOR VS1011E  OK" |
| disable\_spi | Tắt module SPI | "DISABLE SPI OK" |
| test\_flash | Kiểm tra đọc ghi Flash  (Manufacturer, đọc 256 byte dữ liệu từ địa chỉ 0x0004F000) | "TEST FLASH "  "Manufacturer: []"  "OK" |
| read\_wf | Đọc 1 wave file [0..42] trong Flash | "Select a number [0..42]: "  "num read: []"  "start address read: [] -> "  "[Data read]" |
| test\_wf | Phát tất cả wave file ra loa | "TEST WAVE FILE "  "test wave file: []"  "test wave file OK" |
| test\_storm | Phát mẫu câu thoại bão ra loa | "TEST STORM OK" |
| test\_cross\_over | Phát mẫu câu thoại vượt biên | "TEST CROSS OVER OK" |
| test\_bt | Kiểm tra hoạt động của nút nhấn | "TEST BT OK" |
| disable\_uart | Tắt module UART | "DISABLE UART OK" |
| reset\_board | Reset board | "reset\_board … " |
| get\_adc | Đọc điện áp PIN indoor | "POWER: []V" |
|  |  |  |

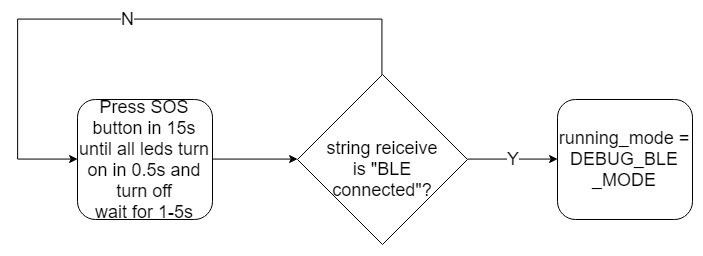
Bảng 49: Tập lệnh hỗ trợ kiểm tra module và cấu hình thiết bị (tiếp theo)

|  |  |  |
| --- | --- | --- |
| check\_adc | Đọc điện áp và tính % PIN indoor còn lại | "POWER: []V"  "Battery percent: []%] |
| bass\_enhancement | Điều chỉnh Bass Treble cho audio | Enter parameters:[treble\_amp]  [treble\_lower\_limit\_freq]  [bass\_amp] [bass\_lower\_freq\_limit]  "Set bass enhancement OK" |
| read\_per\_peer\_address | Đọc địa chỉ MAC mà thiết bị indoor muốn kết nối đến | "peripheral peer address: %02x:%02x:%02x:%02x:%02 x:%02x" |
| change\_per\_peer\_address | Đổi địa chỉ MAC mà thiết bị indoor muốn kết nối đến | "peripheral peer address:  <user type the peripheral:  %02x:%02x:%02x:%02x:%02x:%02x  >  "peripheral peer address read from buffer:  %02x:%02x:%02x:%02x:%02x:%02x"  "Change peripheral peer address  SUCCESS"  "reset\_board …" |
| peer\_address\_flag | Kiểm tra cờ check ghi địa chỉ MAC | "peer address flag: [%02x]" |
| change\_oudoor\_pass | Đổi password đăng nhập thiết bị  Outdoor(độ dài tối đa 32 bytes) | "Outdoor password:"  <user type the new password: >  Change outdoor password SUCCESS |
| outdoor\_pass | Đọc password đăng nhập thiết bị  Outdoor hiện tại | Outdoor password currently:  <password> |
| manufactory | Đưa các thông số cấu hình về mặc định | "Manufactory Ok"  "reset\_board …" |
| read\_good\_pin | Đọc trạng thái chân GOOD của  BQ24073 | "good char pin: []" |
| read\_char\_pin | Đọc trạng thái chân CHARG của  BQ24073 | "char char pin: []" |
| check\_charge | Kiểm tra trạng thái sạc PIN | "CHARGING" hoặc "CHARGE  DONE",  "NOT CHARGE" hoặc "CHARGE  FAULT" |
| disconnect\_ble | Ngắt kết nối Bluetooth | "Disconnect BLE …" |
| disable\_charge | Không cho phép sạc | "disable charge OK" |
| enable\_charge | Cho phép sạc | "enable charge OK" |
| erase\_id\_mess | Xóa ID mess về 0 | "Erase ID message OK" |
| test\_wave\_file | Kiểm tra toàn bộ dữ liệu wavefile ghi vào | "WF OK" |

### BLE debug

Chế độ này yêu cầu có thiết bị hỗ trợ cấu hình kèm theo được cung cấp cho các cơ quan quản lý cũng như nhà máy sản xuất.

Thiết bị hỗ trợ cấu hình cũng sử dụng chip NRF51822 đã được nạp sẵn firmware BLE debug S-ALARM với ngõ ra là USB-To-UART chỉ cần thực hiện cắm thiết bị vào cổng USB của máy tính.



Hình 51: Lưu đồ giải thuật để vào chế độ BLE debug

# Kết quả đạt được trong thời gian Thử việc/Tập nghề

|  |  |  |
| --- | --- | --- |
| **STT** | **Nội dung công việc đã thực hiện (nêu các đầu việc đã thực hiện, kết quả đạt được)** | **Đánh giá của người được giao kèm cặp (nếu có) hoặc của người quản lý trực tiếp (đối với từng nội dung tương ứng)** |
| **I** | **Công việc chính** |  |
| 1 | * + Tìm hiểu về cơ cấu tổ chức, văn hóa, các giá trị cốt lõi của Viettel.   + Tìm hiểu về chức năng, nhiệm vụ của Tổng công ty Công nghiệp Công nghệ cao Viettel |  |
| 2 | * + Coding style sử dụng Linux kernel coding style   + Tìm hiểu GCC, GDB, makefile   + Ngôn ngữ C   + Cấu trúc dữ liệu (linked list, stack, queue)   + Các giải thật sắp xếp trong C   + Lập trình C trong Linux (process, thread, IPC, socket)   + Hoàn thành các bài tập lập trình được giao   + Tìm hiểu thiết bị S-Tracking   + Tìm hiểu hệ điều hành free   + Tham gia một số task của dự án S-Tracking (thử nghiệm chip sạc cho phiên bản mới, debug klocword) |  |
| **II** | **Công việc phụ (hoặc đột xuất)** |  |
| 1 | Không có công việc phụ (hoặc đột  xuất). |  |
| **III** | **Tính phối hợp, tinh thần trách nhiệm, khả năng sáng tạo** |  |
| 1 | Tham gia các buổi seminar của dự án. |  |
| 2 | Tham gia các khóa học đào tạo cho nhân viên mới của tập đoàn |  |
| **IV** | **Đề xuất, kiến nghị** |  |
|  |  |  |

# Các Tài liệu tham khảo đã đọc, học trong thời gian Thử việc/Tập nghề

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Tên tài liệu** | **Tóm tắt nội dung** | **Tác dụng đối với công việc đang thực hiện** |
| 1 | Linux Kernel Coding Style | Các quy chuẩn về lập trình linux để có một chương trình tường minh và có tính thừa kế cao | Sử dụng các tiêu chuẩn đã học để viết các chương trình |
| 2 | GCC/GDB manual/ GNU Makefile manual | Bộ công cụ lập trình C trên linux bao gồm trình biên dịch, trình tìm lỗi và makefile | Sử dụng để lập trình C trong môi trường Linux |
| 4 | The C programming language – 2nd edition | Các kiến thức căn bản về lập trình ngôn ngữ C | Cung cấp toàn bộ các kiến thức lập trình căn bản C cho lập trình nhúng |
| 5 | Algorithm in C – 3rd | Các giải thuật trong lập trình C | Hiểu được các giải thuật sắp xếp |
| 6 | Data structure in C – 2nd | Các cấu trúc dữ liệu trong C | Cung cấp kiến thức căn bản về linked list, queue, stack |
| 7 | Beginning Linux Programming – 4th | Cung cấp các khái niệm liên quan đến lập trình C trên Linux | Hiểu được các khái niệm process, thread, IPC |
| 8 | BM.30.QT.VTTEK.QLCL.01. Thiet ke chi tiet\_S\_TRACKING\_&\_S\_ALARM | Thiết kế về phần mềm phần cứng, các luồng xử lý của thiết bị outdoor của dự án S-Tracking | Hiểu được cấu tạo và nguyên lý hoạt động của thiết bị outdoor |
| 9 | Mastering the FreeRTOS Real Time Kernel | Các khái niệm trong hệ điều hành thời gian thực | Để lập trình cho thiết bị outdoor của dự án S-tracking |

# Nhận xét của các cấp

# Nhận xét, đánh giá của người quản lý trực tiếp:

|  |  |  |
| --- | --- | --- |
| Ngày | Nhận xét  Đề xuất: | CHỮ KÝ |

# Nhận xét của người quản lý chức năng (nếu có)

|  |  |  |
| --- | --- | --- |
| Ngày | Nhận xét  Đề xuất: | CHỮ KÝ |

# Xét duyệt của Ban giám đốc

|  |  |  |
| --- | --- | --- |
| Ngày | Nhận xét  Đề xuất: | CHỮ KÝ |

Ghi chú: Người quản lý chức năng là người không quản lý về mặt hành chính nhưng  
quản lý về mặt nghiệp vụ.  
Ví dụ: nhân viên Tài chính cửa hàng:

+ Người quản lý trực tiếp là Cửa hàng trưởng.  
+ Người quản lý chức năng là nhân viên kế toán chuyên quản (phòng Tài chính Chi nhánh)