Bộ tiền xử lý trong C

Là 1 cái khối nạp và là giai đoạn nào để biên dịch 1 code thành

Khi viết 1 application thì ta có 1 cái source code.c thì cái file .c này là đầu vào cho bộ tiền xử lý preprocessor, bộ này có nhiệm vụ expand cái code c của chúng ta ra, expand ở đây có nghĩa là xử lý 1 số cái chỉ thị tiền xử lý như là #include, #define, … là những line bắt đầu bằng ký hiệu # và bao gồm cả expand macro.

Macro

Được define theo kiểu #define → nó sẽ thay thế các giá trị này vào trong code.

Code của chúng ta sẽ được 1 file mở rộng của source code c. trong đó file source code c này đã được mở rộng rồi qua bộ tiền xử lý thì nó sẽ làm đầu vào cho complier, complier sẽ biên dịch file mở rộng ấy để tạo ra được những file object code.

Những **file object code** này thì làm đầu vào cho linker. Nếu application sử dụng 1 số thư viện standard chẳng hạn như là stdio.h hay math, string, 1 số thư viện hệ thống và những thư viện này được cung cấp trong trình biên dịch rồi, trong complier rồi thì nếu mà trình biên dịch của chúng ta, ứng dụng của chúng ta sử dụng những cái thư viện ấy thì nó cũng sẽ include những cái file hay thư viện static libraries vào để link chúng với nhau để tạo ra những file excutable code để nạp vào chip để chip có thể chạy được luôn.

Static libraries chỉ đơn giản là những file tập hợp object code thôi, thì tức là đã được biên dịch rồi, về sau là những phần linker đấy chỉ việc là include vào link thôi.

Và cái static libraries này có thể, 1 số thư viện đã được cung cấp bởi complier rồi nhưng 1 số thư viện thì cũng có thể là mình hoàn toàn tạo ra được, ví dụ như là .. và có thể biên dịch thành file lib.h và như vậy thì ngoài complier cung cấp những thư viện ra thì user cũng có thể tạo ra những file thư viện này.

Đó là 1 quá trình biên dịch từ 1 file source code thành file libraries, excutable code thì bộ tiền xử lý của chúng ta liên quan để macro, liên quan đến các chỉ thị tiền xử lý thì khối preprocessor là bước đầu tiên tiền xử lý thành file source code c.

Bộ tiền xử lý trong C là bộ tiền xử lý macro được sử dụng bởi trình biên dịch c để chuyển đổi source code của chúng ta trước khi nó có thể được complie. Ở bộ này cũng có thể cho chúng ta define các macro. Quá trình tiền xử lý này bao gồm xử lý những cái chỉ thị tiền xử lý và những cái macro và expand nó ra thì bao gồm những cái tiền xử lý về include, header rồi mở rộng macro ra và xử lý những cái chỉ thị điều kiện biên dịch như là #dif, #dif define,… và 1 số chỉ thị chuẩn đoán #bulling, #error, …

Chốt lại chỉ thị tiền xử lý luôn bắt đầu bằng #

Là ký hiệu báo hiệu 1 cái chỉ thị tiền xử lý.

Macro

là mảnh của code, nó có được cấp 1 cái tên để định danh thì khi mà quá trình tiền xử lý diễn ra thì nó sẽ xử lý bất kỳ chỗ nào của cái tên macro define đấy xuất hiện thì nó sẽ thay thế giá trị nội dung content của macro đó vào trong file source code và trong quá trình tiền xử lý đó

Và đối với macro thì chúng ta sẽ dùng cái ký hiệu là #define để có thể define 1 cái macro nào đó. Khi nào thì ta sử dụng, define những cái macro như thế này. Khi chúng ta tạo 1 hằng số hay 1 cái hằng string hay tạo 1 biểu thức.

Có 2 loại macro: 1 cái là **object like macro** (để define 1 số cái data, 1 cái giá trị hay data nào đó) và 1 cái là **function like macro** (được dùng giống như những cái function con của function, nó có những parameter).

Object like macro thường dùng để define 1 cái hằng số và cái body của nó thì sẽ kết thúc cái line đấy, là phần kết thúc của macro body.

Có thể define trên nhiều dòng với điều kiện kèm theo là thêm 1 cái ký hiệu \ để nối dòng thì ở đây là define theo kiểu nhiều dòng.

```
#define NUMBERS 1,\
                2
```

/*Tiền xử lý sẽ thay thế nội dùng của macro là 1,2 thành NUMBERS*/

```
#define UART0_BDH_SBR_MASK          (0x1FU)
#define UART0_BDH_SBR_SHIFT         (0U)
#define UART0_BDH_SBR(x)            (((uint8_t)(((uint8_t)(x))<<\
UART0_BDH_SBR_SHIFT))&UART0_BDH_SBR_MASK)
```

2 marco được define ở trên:

- Macro Mask của field bit sbr trong thanh ghi ghi bdh của TBNV UART0
- Macro Shift thể hiện vị trí của cái bit sbr này trong thanh ghi BDH của UART0 thì sbr này nằm ở vị trí bit thứ bao nhiêu trong thanh ghi ấy

Trong define function, để get được giá trị cho field sbr này thì khi chúng ta truyền giá trị x vào thì return của macro này sẽ luôn luôn trả về giá trị trả về 1 giá trị để đảm báo nó ghi được vào cái field sbr đó và giá trị của nó và giá trị của nó ko được vượt quá ngưỡng cho phép của field sbr và đồng thời nó ko ảnh hưởng đến các field khác trong thanh ghi bdh này khi viết vào thanh ghi bdh này.

Ta có thể thấy nội dung của nó: x dịch sang trái với giá trị số lần dịch bằng shift macro này và sau đó and với define mask này.

Khi and với mask này thì ta thấy với phép AND bit thì sẽ làm cho x nếu có giá trị vượt quá range của bit field của sbr này đi chăng nữa thì khi AND với cái mask này thì nó sẽ luôn luôn làm cho giá trị ở những field khác, field ko thuộc về sbr này có được clear hay ko và nó sẽ chỉ được giữ lại giá trị của field sbr này thôi.

Dùng để lọc ra được giá trị sbr chuẩn

Uint8_t là ép kiểu cho x: (unint8_t)(x), sau khi ép kiểu uint8_t thì dịch sang 0 lần sau đó là ép kiểu uint8_t để đảm bảo giá trị này của nó vẫn thuộc range uint8_t, sau đấy là AND với mask này

Dấu \ ở trước << là dấu nối dòng.

Ép kiểu x này để giá trị x này luôn luôn là đảm bảo range là uint8_t vì thanh ghi sbr này là thanh ghi 1 byte 8 bit thôi để đảm bảo x này là 8 bit thì như vậy khi mình dịch sang shift lần thì nó vẫn đảm bảo được rằng ko thể vượt quá 8bit, nếu lỡ vượt quá thì giá trị x cao (bit 8 trở đi) sẽ bị đẩy ra khỏi giá trị return của nó.

Đối với 1 phép dịch thì kết quả của phép dịch có thể khác kiểu, nó ko giống như kiểu mà mình expect.

Như vậy mình ép 1 lần nữa để đảm bảo dù có dùng phép dịch thì kết quả sau phép dịch vẫn là uint8_t. để khi dùng macro này có OR với 1 giá trị khác đi để ghi vào thanh ghi BDH này thì nó vẫn đảm bảo 8bit khi ghi vào thanh ghi BDH này thì ko bị vượt quá giá trị, cái range của thanh ghi này.

Thực ra phép dịch bit này chính là nhân. Phép nhân hay chia hay phép gì đi nữa thì nếu mà thao tác với 1 cái giá trị mà, 1 phép toán mà có 2 cái tham số, nó ghép kiểu nhau thì rất dễ làm thay đổi kiểu của giá trị trả về.

Thì nhân số uint8_t với 1 số uint8_t thì chắc chắn nó thành số 16, có thể giá trị của nó ko chứa vừa với kiểu ban đầu của nó là 8 là vì thế.

Đó là lý do khi mà define 1 function like macro hay là khi thao tác với 1 phép toán nào đấy chẳng hạn như dịch thì cần ép nó về uint8_t để đảm bảo rằng sau khi mình dùng function like macro này thì giá trị của nó luôn luôn là trong dải của mình chấp nhận được là uint8_t theo cái expect của mình.

Bản chất của phép dịch chính là nhân thôi.

Các thanh ghi BDH này chỉ có 8 thôi

Với việc **function like macro** ngoài là 1 function như bình thường thì có thể compare dc khác nhau giữa kiểu sử dụng này ko? Function thường thì được biên dịch sau giai đoạn tiền xử lý

Còn **function like macro** thì **được xử lý ở giai đoạn tiền xử lý, tăng size của chương trình trong trường hợp khi gọi macro này nhiều lần.**

Còn function mỗi lần dc gọi thì nhảy đến function đấy chứ ko thế cả nội dung của function đấy vào.

Thế trực tiếp macro vào nhưng tại sao **function like macro** lại nhanh hơn.

Function like macro sẽ **bỏ qua quá trình stacking**, còn function thì ko nên tốc độ của **function like macro** sẽ nhanh hơn.

Với **Function like macro** thì đoạn text mình lấy ra thì nó được paste vào cái mình gọi ra, còn function thì mình gọi đi gọi lại nhiều lần thì thêm quá trình call function đấy ra, sẽ có stacking. Cụ thể khi call thì nó làm thế nào.

Khi gọi hàm thì câu lệnh kế tiếp ngay sau hàm đó được lưu vào stackpointer và thanh ghi **PC** sẽ trỏ đến địa chỉ của hàm đó. Khi mình gọi hàm thì **PC** mới trỏ đến bộ nhớ lệnh của cái hàm đấy, sau khi thực hiện hàm đấy thì nó lấy trên đỉnh stack địa chỉ của ô lệnh tiếp theo đặt vào thanh ghi PC để thực hiện tiếp, thời gian đặt đó tốn thời gian.

Còn **function like macro** thì đặt nguyên câu lệnh của mình ở đó, ko phải gọi đi đâu nữa.

Sau khi kết thúc function đấy thì nó lại phải đặt giá trị của thanh ghi **stackpointer** vào thanh ghi PC để thực hiện tiếp câu lệnh tiếp theo.

Đối với function bình thường, mỗi lần gọi nó sẽ insert thêm các lệnh như là lệnh jump đến function đấy, có thể hiểu là move **PC** vào địa chỉ của function mình muốn gọi.

Đầu function đấy có những cái lệnh để lưu trữ vào trong stack. 1 số backup trong thanh ghi stack nó sẽ back ra. Đến cuối function đấy, để có thể return về cái địa chỉ gọi function trước đó gọi function đang thực thi đấy thì nó cần phải backup lại **stack pointer** sau đấy thì ..

Lúc đầu thì sẽ move giá trị thanh ghi vào stack, cuối function thì nó lại move ra, backup lại sau đó thêm lệnh nhảy về địa chỉ trước đó gọi function ấy. đó là 1 số câu lệnh khi gọi function bình thường (quá trình stacking)

Còn **function like macro** thì sẽ ko có những cái câu lệnh đấy thì nó sẽ làm cho chương trình sử dụng **function like macro** thì nó sẽ **excute nhanh hơn**