

Unidad 3 DISPOSITIVOS DE IN/OUT MAPEADOS EN MEMORIA

CLASE 13

Martes 19/2024

CARACTERÍSTICAS DEL SW PARA SE.

- En sistemas de cómputo de propósito general el SW se distribuye en formato binario y puede instalarse en varias máquinas
 - También se distribuye en código fuente, pero es menos común (Linux - open source)
 - El instalador ofrece más alternativas de configuración
- En ESS el software lo llamamos FIRMWARE, ya que no es modificable por el usuario
 - No se distribuye, viene pre-programado en el E.S.
 - Algunos más modernos pueden tener capacidad de actualizarse (Ej FW en Impresora 3D)
- El FW es escrito para trabajar dentro de las restricciones de cómputo, memoria, dispositivos I/O, etc. de un dispositivo en particular (MCU o familia de MCUs)
- Para facilitar la portabilidad del F.W., el SW de ESS se escribe sobre una serie de capas de SW que abstracten las particularidades del hardware
 - Propiedades → Portabilidad
 - Backward compatibility (+ en syst. de prop. gen.)
 - La capacidad de dividir el trabajo.
 - Puede ser que un producto evolucione tanto que ya no cabe en el MCU para el que se desarrolló originalmente.

Es en estos como estos en el que ya no todo el F.W. VA A FUNCIONAR!

(todo lo que viene en el contexto → va a caer en el contexto → $\frac{M2}{M3}$ pero no hacia otras compatibilidades hacia arriba, ↓)

Para garantizar portabilidad en el código el F.W. se diseña por capas.

Modelo de Capas del SW de ESS:

Permitir básicos controlar los pocos recursos
que son la memoria y el procesador

Ej: Un driver para un GPS + middleware!
Un driver para un módulo LTE de señales)

Capa Aplicación	
API - I/F de programación de Aplicación	
OS Embeded/RTOS	Middleware
HAL - Capas de abstracción del HW	DRIVERS
	LIBRERÍAS DE H.W.

→ es lo que NO VA EN EL OS. Ej: ...

El OS y el middleware están construidos sobre una capa de abstracción de H.W. Esta a su vez está construida sobre los drivers y estos a su vez, están construidos sobre los lenguajes de hardware

A) Capa de Aplicación

- Es la capa de mayor abstracción del F.W.
- El main en una aplicación bare-metal
- Estamos hablando de cómo especifica la aplicación al conjunto de apps del sistema embedido
- En esta capa de app es donde vamos a especificar cómo funciona el sistema.. Donde vamos a implementar las características funcionales en base a los requisitos funcionales del sistema.
- En sist. embed. que no tienen O.S. (lo que se llama Bare-metal), esa capa de aplicación se convierte en el "main".
- Pero si hablamos de un sistema que sí tiene S.O. entonces corresponde al conjunto de Tareas que "describen la aplicación" = Allí está reflejada esta capa
- En esta capa se implementa la especificación completa del E.S.
 - Requisitos Funcionales
 - Requisitos no funcionales
- Las capas inferiores proporcionan la abstracción necesaria para que el código de la aplicación sea lo más portable posible.
 - Es decir... que sea "Independiente del H.W."
- Si bien los E.S. están diseñados para un conjunto de aplicaciones definido e TIEMPO DE DISEÑO, la portabilidad del SW sigue siendo muy importante
 - Migración a otro MCV
 - Reuso del S.W en otros productos de la compañía.
- ¿Cuál es el objetivo de la capa de aplicación?
- En esta capa no debería haber ningún detalle del hardware

el mundo real es muy complejo, entonces una forma de mitigar la complejidad es quitarle detalles, i.e. crear modelos

Ej: una planta la "veo" a través de un más sencillo modelo matemático. (se "abstiene")

se hace lo mismo en SW y H.W.
tratar de reducir la cantidad de detalles a proporcionar para que algo funcione!!

que nos evita las particularidades del H.W.

Y de alguna forma, los otros apps, nos tienen que esconder esas particularidades del H.W.

Ej) Cuando estamos transmitiendo datos por un serial..

setup: - Un dispositivo embedido que se comunica con un computador.

- Esta comunicación hoy día se puede dar ppal/ a través de USB

- Pero todavía hay PCs con puente serial.

- Supongamos que esa comunicación es parte fundamental de la APP; pero si yo changeo las particularidades del módulo serial en la aplicación y el día de mañana nos toca migrar de serial a USB.

- lo que haremos inicialmente es que pudiendo haber hecho el código de aplicación independiente del esos I/Fs seriales particulares ("del HW") y entonces cualquier modificación, nos toca ir allá y modificar todo el código (código spaghetti) → "hay de todo en todos los niveles".

(B) Interfaz de Programación de Aplicación - (API)

• El API define todos los métodos, funciones disponibles, los protocolos, y todas las herramientas necesarias para escribir el código de la aplicación.

- UNO puede definir unas funciones...
EJ. → UNO puede definir una función que se encargue de definir si un pin está activo o no.

→ UNO puede definir una función que genere un LED todo momento

- Y empleo a construir todo eso que necesito... y ya en la aplicación digo:

`ReadButtonStart` ← (eso es como un API)

solo posiblemente modifiqué la fcn { si el día de mañana ya no es el pin 5 sino el 8! }
yo no modifiqué la aplicación. (Incluso cambió el pin)
podría estar en una app de abstracción más arriba)

El ideal → Proteger la aplicación y que no dependa tanto del H.W.
→ (aunque no siempre es posible!)

- UNO puede pensar en API's genéricos para funciones (cosas) MUY RECURRENTES
(y servir a muchas aplicaciones)

Ej: • El manejo del TIEMPO } SON MUY RECURRENTES EN LAS APLICACIONES
• SLEEP }
• ALARMS } "Y se puede tener un API que le sirve a muchas
aplicaciones y productos!"

-- Y el código de la aplicación, pues si es particular de un producto"

- UN API define la interfaz de alto nivel para acceder a los componentes del MCV
 - Entradas y Salidas
 - Cómputo
 - Aceleraciones
 - Esta capa permite que los cambios en H.W. no afecten la aplicación
 - Si el H.W. cambia se modifica el API y NO la aplicación
- ¿Qué es lo que le da valor al producto?
- No es el detalle de que el pin esté en "1" o en "0"
 - Es la funcionalidad que está en la aplicación y que se ofrece al usuario.

C) AMBIENTE OPERATIVO → TIENE 2 Opciones de implementación

C.1 BARE-METAL → es lo que se verá este cuadro!

- El más simple de todos donde NO se usa ningún tipo de O.S.
- El F.W. se desarrolla con el H.W. para optimizar la operación y realizar algunas de las funciones de un O.S.
- La ppal función será administrar el acceso a los recursos de cómputo del MCV

C.2 USAR UN O.S. ENBEDIDO → ¿Qué ofrecen? → UN API para...

- Aquí lo que se tiene es un código de un TERCERO.
- Hay varios O.S. que están disponibles. y nosotros trabajamos con el FREERTOS en sistemas enbedidos. [lo compró Amazon]
- SON desarrollados específicamente para el ambiente de MCUs
- Comúnmente nos referimos a ellos como SISTEMAS OPERATIVOS DE TIEMPO REAL (RTOS)
- Se encarga de distribuir el tiempo de CPU entre las diferentes tareas (Scheduling) y proporciona un API para comunicación y sincronización de las tareas.

UN O.S. → Tiene un componente ppal → "el scheduler" (oplificador de tareas) que tiene como función: determinar qué tareas se ejecutarán en qué instante de tiempo

- En este ambiente uno como desarrollador, cada tarea es como si fuese un proc ppal que tiene un pedazo de inicialización y después tiene un loop (o) infinito
- Y uno debe hacer que esas tareas indepedientes contribuyan a la aplicación completa!
- Y por eso el S.O. proporciona API para comunicación y sincronización entre las tareas!

con el objetivo último de que las tareas puedan trabajar de forma cooperativa en un propósito. (3)

- || "LOS RTOS superan el desempeño de las implementaciones BASE-METAL PERO REQUIEREN MÁS MEMORIA"
 - Esto es "entre comillas" → Son los RTOS más rápidos que las apps BASE-METAL.
RTA/ depende de las aplicaciones
 - SI SON MUY SEÑILLAS → es difícil que con el O.S. se tenga un código más simple o más rápido.
 - CON UN S.O. → hay un "OVERHEAD" → hay que ESTAR haciendo TRANSICIONES ENTRE TAREAS ("context switching")
 - CUANDO EL "Scheduler" → Toma la decisión que la Tarea A NO VA MÁS Y QUE EL OS → LA TAREA B → se debe ejecutar...
 - Guarda entonces el "Estado arquitectónico del procesador" en la Tarea A (previa), antes de poder saltar a la siguiente.
 - de propósito específico (stack pointer, program counter, etc) → Todo lo que te permita luego reanudar la ejecución en el punto donde FUE interrumpida!
 - ALMACENAR ESTE CONTEXTO → TIENE MUCHAS IMPLICACIONES DESDE EL P.V. DEL DESEMPEÑO (ESO HAY QUE meterlo al stack, no se hace en "TIEMPO ZERO" y cuando regreso a REANUDAR HAY QUE VOLVER A RESTAURAR EL "ESTADO DEL PROCESADOR" ANTES DE SEGUIR EJECUTANDO TAREAS. SWITCHEAR DE UNA TAREA A OTRA TIENE UN COSTO!!)
 - ESE COSTO NO ESTÁ EN EL BASE-METAL
 - PERO, entonces porque ES MEJOR el O.S.?
 - RI porque... → PERMITE MANEJAR PRIORIDADES Y → ~~PERMITE ENTREPRENDER~~ → ~~realmente hacer~~ TIEMPO REAL (REALTIME)
 - NO es que se haga muy rápido sino...
 - que se haga en el Tiempo destinado.
 - "QUE LAS TAREAS SE COMPLAN EN EL TIEMPO QUE ESTÁN DESTINADAS PARA EJECUTARSE"
 - Es AGREGARLE A LAS TAREAS EL "Deadline"

En tiempo real → Duro → alguien se puede morir

→ suave → la imagen en el TV se distorsiona un poco

- Quién define las prioridades de las tareas?

Yo como desarrollador → NO puedo poner con más prioridad las tareas equivalentes

- TRABAJAR CON O.S. SÍ ES MÁS SENCILLO

- Uno hace la división ^{entre} de TAREAS (Usted hace esto... y yo hace esto...)
- Uno lanza las TAREAS, las comunica ... y ya!
- El O.S. se encarga de hacer que ellos se ejecuten.

- CUANDO TRABAJO UNA APLICACIÓN EN BARE-METAL, SOY YO EL QUE TENDRÉ QUE HACER ESA PRIORIZACIÓN DE TAREAS DENTRO DEL CÓDIGO (Dependiendo de una aplicación de igual complejidad!). SOY YO EL QUE TENDRÉ QUE TENER DIREKTAMENTE LAS ESTRATEGIAS DE CÓMO HACER QUE LA PRIORIDAD FUNCIONE Y ESTO NO ES FÁCIL!!

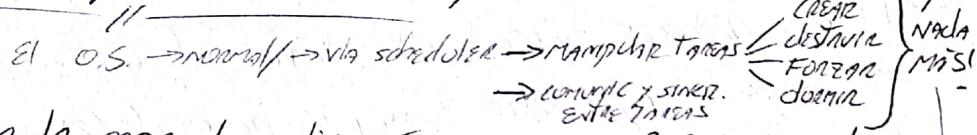
C.3 Algunos E.S. pueden tener un O.S. completo!

- Algunos MCUs con suficiente cómputo y memoria pueden implementar O.S. como Linux

Ej.: Los Raspberry Pi, los SBC (Single Board Computers), Algunas FPGAs que combinan procesadores + lógica programable (tienen capacidad de correr un sistema Linux completo)

- En estos casos ese O.S. completo, se va a encargar también de las funciones del middleware (y se supone que es todavía más fácil!).

D) MIDDLEWARE



- Proporciona servicios a la capa de aplicación que no proporciona el ambiente operativo. → Todo lo de comunicaciones, sistema de archivos}
- Entre los servicios proporcionados por el middleware tenemos:
 - Almac. de datos y sistemas de archivos
 - Conectividad (Wi-Fi, Bluetooth, LoRaWAN, ZigBee, etc.)
 - Funciones específicas del dispositivo (Encriptación, compresión, etc.)
 - Drivers de periféricos externos al MCU. → ej un GPS.

(E) CAPA DE ABSTRACCION DEL H.W. (HAL)

- Proporciona una interfaz común a las capas de OS embedido y Middleware con los drivers del hardware
- Es como lo que hace la API para la aplicación... lo hace...
- El HAL se puede trabajar a nivel de los drivers ó como un WRAPPER que proporciona una interfaz común a capas superiores del software (S.W.)

→ Abstrae el hardware a la APP
+ (4)

→ Abstrae los drivers al

Ej: Abstace los módulos de GPS o Corundum

... la HAL para el < O.S. > el Middleware

Ej: CUANDO trabajamos los VART en MCUs

- VART de la RPP → para TX! → probablemente esa función sea pareada
- VART de un ESP32 → !! !! ! → otras en un...
- VART de un microchip

Todos tienen señales! → pero la función propia para TX un serial es diferente en C/U! → porque?

- porque el driver es diferente!

⇒ Aquí lo que hace la HAL es:

- Si es el MCU1 → entonces llame esta función
- Si es el MCU2 → !! llame esta otra función

QUE podrían ser
código de pre-compilador
#ifdef_RPP
llame esta función!
#else if#define_ESP32
llame esta otra...

!! UN WRAPPER !!

→ Es tratar como de crear una IF para el O.S. ó el Middleware
tratando de abstractar los detalles de H.W.

- Si cambia el H.W. ó se hacen modificaciones en los drivers, cambia la HAL pero NO el O.S. ó el Middleware.

(F) DRIVERS (el driver y las librerías de HW si son 100% dependientes del H.W.)

- Capa 100% dependiente del H.W
- Implementa la funcionalidad de los diferentes módulos I/O, y sistemas del MCU
- Utilizan las librerías de H.W. para acceder a los registros de configuración de los módulos.
- Normalmente un driver define un conjunto de funciones para manipular el módulo...
 - Inicialización
 - Lectura de DATOS
 - Escritura de DATOS

⑥ LIBRERÍAS DE H.W.

- ESTAS SON LAS QUE NOS DAN ACCESO A LOS REGISTROS DE CONFIGURACIÓN → DE LOS MÓDULOS I/O:

Servicios tienen una librería de H.W para:
" " " " " " " " " " PARA el UART

" " " " " " " " " " PARA SPI,
" " " " " " " " " " PARA I2C,
" " " " " " " " " " PARA ADC, etc

- Hay Módulos del sistema (del MCU) como:
Que también tienen opciones de configuración
Y que por tanto también necesitan un driver.
" " " " " " " " " " PARA CLK
" " " " " " " " " " PARA Watchdog
" " " " " " " " " " PARA TIMERS, etc

Habrá una librería de H.W para CLK.

En la clase pasada vimos el ejemplo de las librerías de H.W. y cómo se presentan para la RP2040.

- Allí se hablaba de una librería de estructuras y ...
... de una librería de registros

- Aquí se definen una serie de ...

- MACROS
- ESTRUCTURAS

PARA ACCEDER A LOS DIFERENTES REGISTROS DE CONFIGURACIÓN

- También es común definir ...

- MACROS

PARA LA MANIPULACIÓN A NIVEL DE OPERACIONES BITWISE

- OTRA ALTERNATIVA ES DECLARAR LOS REGISTROS USANDO ...

- UNIONES Y
- ESTRUCTURAS DE BITS

ESTO DE TRABAJAR POR CAPAS → ES UNA BUENA ESTRATEGIA PARA EL TRABAJO ENTRE LOS COMPAÑEROS DEL EQUIPO.

→ DE CIERTA MANERA EL SDK ESTÁ CONSTRUIDO DE LA MISMA FORMA: (LIB. ALTO NIVEL, LIB. DE HW, LIB. DE ESTADOMS... DE REGS)
API Drivers Lib de Hardware

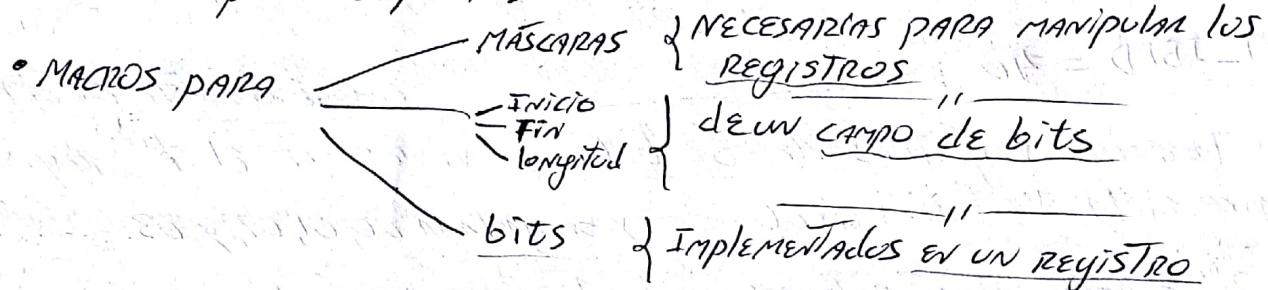
Y el Middleware?

El FREERTOS → TIENE COSAS PARA MANEJAR MEMORIAS SD, SISTEMA DE ARCHIVOS, PROCESAMIENTO DE SEÑALES. QUE SE METEN AQUÍ POR EL MIDDLEWARE. COMPRESIÓN, DECODIFICACIÓN, INTELIGENCIA ARTIFICIAL. HAY LIBRERÍAS PARA ESTO ADICIONALES A LO QUE PROPORCIONA LA RPP Y QUE PERMITIRÍAN HACER LAS VECES DE MIDDLEWARE EN PARTICULAR ARM PROPORCIONA UNA LIBRERÍA QUE SE LLAMA ARM CMSIS → es como el SDK OFICIAL de ARM DSP → podrían ser útiles en proyecto final!

El SDK NO TRAE SISTEMA OPERATIVO. ESTE LO TENDRÁS UYO QUE COMPILARLO APARTES
PERO NO SE HACE ESTE SEMESTRE. SE HACE EN EL CURSO DE ENBEVIDOS.

I) LIBRERÍAS 100% MACROS

- Definen, por medio de macros, los registros de todos los periféricos
- Opcionalmente pueden definir:



- Su principal ventaja es que cada dirección usada en los MACROS debe almacenarse en Flash (Por cada registro que yo declare hay que suministrar un apuntador a una dirección en memoria flash.)

EJ: LIBRERÍA DE H.W. DE UN MÓDULO UART (EN ARM PRIMECELL PL011)

- LOS REGISTROS DE CONFIGURACIÓN DEL PUERTO SERIAL ESTAN EN ...
`#define UART_BASE 0x40003000` → UN MACRO CON LA DIR. (DÓNDE APARECEN LOS REGISTROS)
- #define UART_DATA (*((volatile unsigned long *) (UART_BASE + 0x00)))
Y LUEGO SE DEFINEN UNOS APUNTADORES → CASTING.
Y LO ESTOY DICHIENDO: ESTO ES UN APUNTADOR A UN ENTERO DE 32 BITS SIN SIGNO!
--- Y ¿EN DONDE ESTÁ? → ESTÁ EN UART_BASE + 0x00
(ES DECIR, ESTE ES EL 1º REGISTRO!)
- Y, PORQUÉ HAY QUE HACERLE ESTE CASTING A ESA DIRECCIÓN?
UN APUNTADOR TIENE DOS COSAS?

- UNA DIRECCIÓN
- UN TIPO DE DATO

Y ALLÍ SOLAMENTE HAY UNA DIRECCIÓN
Y PARA QUE SEA UN APUNTADOR HAY QUE DARLE UN TIPO DE DATO

- Y ESO ES LO QUE SE LE DICE ALLÍ: "QUE ES UN POINTER A UN volatile unsigned long *"
VOLATILE → porque puede cambiar en cualquier momento. Se dice al compilador → NO SE META CON ESO VOLTE!
→ PUEDE CAMBIAR EXTERIAMENTE EJ: VALOR RECIBIDO COMO DATOS DE TX/RX → PUEDE CAMBIARLO UNA INTERRUPCIÓN
- QUE EL COMPILEADOR NO INTENTE HACER OPTIMIZACIONES, ETC. CON ESA VOLTE!

Realmente esos apuntadores son constantes y yo no necesito manipularlos (ni sumarle, ni restarle, nada!). Significa que se va a utilizar tal cual! Y entonces pongo el ~~*~~ ^{aquí} ~~((volatile ...))~~
Si no se pusiera el *, entonces cuando voy a usar los chicos tengo que poner ~~*UART_DATA~~ pero...
NO SE VA A USAR ~~*UART_DATA !!!~~ Y por eso se deja el * allí.
--- Y ya le puedo dar valores a UART_IBRD como si fueran variables normales!

UART_IBRD = 40;

Cómo se interpreta ese offset?

- Aquí tenemos registros de 32 bits y vemos que el 1º registro aparece en la dir base + 0x00. Y se guarda la 00, 01, 02, y 03.
 - El 2º registro estará entonces en la dir base + 0x04.
- #define UART_RSR (*((volatile unsigned long *)(UART_BASE + 0x04)))
- Y el 3º registro está en la dir base + 0x18 → Porque? (^{XQué} ~~se dejan esos espacios?~~)

#define UART_FLAG (*((volatile unsigned long *)(UART_BASE + 0x18)))

Es común en muchos MCUs que hay unos espacios y que los registros no siempre son continuos en memoria. Porque? → en la RPP → es solo la RPP7040 no hay más; pero en otras compañías si tienen varios modelos → ej en un modelo se tiene un puñado de canales y en otro modelo se tienen ocho. Es el mismo hardware pero se dejan los espacios de los canales que no están implementados.

- Estos saltos hay que tenerlos en cuenta dependiendo de cómo esté implementando mi librería. En este caso al ser una librería de macros NO ES TAN IMPORTANTE porque aquí se está poniendo la dirección exacta y lista!
- Pero en otras estrategias que vamos a ver, si hay que tenerlos en cuenta. Es UNA RESTRICCIÓN A TENER EN EL MISMO MAPA DE MEMORIA Y QUE NO LO PUEDO CAMBIAR!

MÁS ABAJO SE MUESTRAN QUE A LOS DIFERENTES REGISTROS DE CONFIGURACIÓN DEL UART PUES LE DAMOS LOS VALORES QUE ^{SE} REQUEREN PARA ~~LA~~ CONFIGURACIÓN.

UART_FBRD = 11; (o ^{DARLE} UN NÚMERO EN HEXADECIMAL!) → QUE NO DICEN NADA!
→ CUÁL ES EL PROBLEMA CON ESOS NÚMEROS?

- Ej: $\text{UART_IBRD} = 40$; - primero hay que pasar el 40 a binario
 - luego ir a mirar cómo está declarando el registro y cuáles corresponden a "1s" y cuáles no, para entender la configuración del periférico. Mira la hoja de datos y entiende qué se modifica con esa asignación.
- Por eso es que la mayoría de las veces estas asignaciones se hacen con máscaras ← porque ellos como tienen textos legibles, son más fáciles de entender!
- De todas formas allí el comentarista intenta decir qué está haciendo:
 $\text{ibrd} = 25\text{MHz}/38400/16 = 40$

II

LIBRERÍAS USANDO ESTRUCTURAS

- En lugar de declarar el acceso a cada registro se declara, por medio de UNA MACRO, el acceso a una estructura que tiene idéntico mapa de memoria que el módulo.

Ahorra memoria Flash
 - YANO un apuntador para el registro
 - SINO un punto para el módulo

Aquí son importantes esos saltos en el mapa de mem y que también se deben incluir en la estructura!

- Vamos a tener un macro de acceso pero va a ser solamente a la dirección base
 - Tenemos que declarar una estructura para poderle decir: "El apuntador tiene esta forma" y esa estructura va a tener el mismo mapa de mem que los registros en el módulo (sino los accesos a los registros quedarían desalineados y no funcionaría, y posiblemente modificando el registro que NO es!)

- Opcionalmente se puede declarar (opcional a la estructura de arriba)
 - los registros combinando estructuras de bits y uniones
 - Macros con valores típicos de los campos de bits
 - Máscaras para manipular el registro completo
 - Macros para simplificar el nombre de estructuras, registros y campos de bits

EJ: Cómo declarar la estructura de un módulo? (para el mismo UART!)

```
typedef struct { // BASE ON ARM PRIMECELL PL011A
    volatile unsigned long DATA;           // 0x00
    volatile unsigned long RSR;            // 0x04
    unsigned long RESERVED0[4];           // 0x08 - 0x14
    volatile unsigned long FLAG;          // 0x18
    unsigned long RESERVED1;              // 0x1C
    volatile unsigned long LPR;            // 0x20
    volatile unsigned long IBRD;           // 0x24
    :
    :
} VART_TypeDef;
```

SE DECLARA COMO UN TIPO DE DATOS

AQUÍ ERA DONDE ANTES TENÍAMOS UN SALTO EN MEMORIA!
OJO: HAY QUE METER ESOS SALTOS!!!

tipo de datos \Rightarrow VART_TypeDef \rightarrow QUÉ HACEMOS CON ESO?

```
#define UART0 ((VART_TypeDef *) 0x40003000)
```

```
#define UART1 ((VART_TypeDef *) 0x40004000)
```

```
#define UART2 ((VART_TypeDef *) 0x40005000)
```

YA NO SE ADEPONE
EL * INICIAL, SE HACE SÍ EL CASTING!

Y Puedo hacer cosas como - -

UART0 \rightarrow IBRD = 40; // ibrd: $25\text{MHz}/38400/16 = 40$

→ Cuando Tengo varios UART, como Todos tienen la MISMA ESTRUCTURA, yo los PUEDO "DECLARAR" todos por LA MISMA VÍA

- En la estrategia 100% MÍCROS, yo los TENDRÍA que DECLARAR TODOS INDEPENDIENTES!

→ Ya para UTILIZARLO, veo que se DECLARÓ el VARTX como un MÍCRO Y LO PODEROS USAR EN CUALQUIERA PARTE DE FORMA GLOBAL.

UART1 \rightarrow IBRD = 40; - Notación de APUNTADORES PARA ACCEDER A LOS CAMPOS DE LA ESTRUCTURA

- Si AÚN HUBIÉSEMOS AGREGADO EL * INICIAL:
 $\#define \text{UART1} *((VART_TypeDef *) 0x40004000)$

\Rightarrow YA NO ES FLECHA SINO UN PUNTO (-)

UART1.IBRD = 40;

¿POR QUÉ CREAR MIS PROPIAS LIBRERÍAS?

- Puede ser que no me guste el SDK que me ofrece el FABRICANTE DEL MCU PARA ACCEDER A LOS PERIFÉRICOS. Yo PUEDE ESTAR ACOSTUMBRADO A TRABAJAR DE UNA FORMA Y ESTE NUEVO MCU ME OFRECE UNA LIBRERÍA QUE NO ES COMO YO ESTOY ACOSTUMBRADO

- La empresa ya tiene una experiencia y una forma de trabajar y hay que acostumbrarse a ese estilo de trabajo, codificación, etc. NO ACOSTUMBRAR todo el equipo de trabajo al SDK, sino que el SDK se acostumbre al equipo! ⑦
- A nivel de hardware normal el SDK "Todo lo define" → Todos los registros, todas las máscaras y bits. (en las librerías de HW). [YA en los drivers los SDK → Pueden no expponer todos los funcionalidades/modalidades de uso de los módulos]. y 100% dependientes del MCU
- Productividad:
 - El uso de máscaras es más propenso a errores y más complejo de utilizar
 - El código usando estructuras de bits es más compacto y menos propenso a errores
 - Siempre y cuando el compilador lo soporte
 - La librería esté bien construida.

EJ.: en el SDK de la RPP no se encuentra cómo hacer funcionar un PWM como Timer.. [No hay ninguna función del SDK que lo permita!] //
 - Si yo lo quiero usar de esa forma, entonces a mí me toca hacer el driver!

CUÁL ES EL [OBJETIVO] DE LAS LIBRERÍAS DEL CURSO?

- ← FACILITAR EL ACCESO A PERIFÉRICOS DE UNA FORMA EFICIENTE Y FÁCIL DE RECORDAR
- DEFINIR ESTRUCTURAS de REGISTROS DE CONFIGURACIÓN y ESTADO de los periféricos
 - DEFINIR MACROS para acceder directamente a los ---
 - CAMPOS DE BITS } de registros de configuración
 - DEFINIR CONSTANTES → con los valores que pueden asignarse a los ---
 - CAMPOS DE BITS } de los registros de configuración de los periféricos
 - DEFINIR MACROS → para acceder de forma rápida y fácil a ---
 - ESTRUCTURAS DE MÓDULOS
 - REGISTROS Y
 - CAMPOS DE BITS

EJ: 175 LIBRERIAS DE LA RPP2040 } CADA PERIFERICO TIENE UN NOMBRE + DE REGISTROS

- TIENE 29 ARCHIVOS DE LIBRERIA (LOS ESTA CONSTRUYENDO EL PROFESOR!)

Ej: SÓLO 3 REYES POR GPIO	IOBANK0_LIB 97 REG.	IOQSPI_LIB 22 REG.	PADS BANK0_LIB 33 REG.
	PADSQSPI_LIB 7 REG	VART_LIB 22 REG	SPI_LIB 18 REG.

- Si: UN REGISTRO ESTÁ REPETIDO → NO LO DECLARO DOS (2) VECES

→ 2 REGISTROS FISICOS CON LA MISMA ESTRUCTURA DE BITS!

→ Y YO LO QUE DECLARO ES ESA ESTRUCTURA DE BITS!

Y LUEGO SE REUSAN ESTADOS LOS REGISTROS QUE TENGAN ESA MISMA

OJO

→ NO HAY QUE DECLARAR LOS 97 REGS PARA LOS GPIOS. PERO AUNQUE FUNCIONA NO ES NECESARIO PORQUE LOS REGISTROS, ESTRUCTURAS → SE DECLARAN CON "TYPEDEF" Y LUEGO SE UTILIZA ESE "TIPO DE DATOS" VARIAS VECES CAMBIANDO SOLO NOMBRE, Y NO DECLARAR 97 VECES LA MISMA COSA.

— — —

LA ESTRUCTURA DE LA LIBRERIA DEL CURSO

1. VAMOS A DECLARAR CADA TIPO DE REGISTRO DE UN MÓDULO PERIFÉRICO POR MEDIO DE UNA UNIÓN Y ESTRUCTURA DE BITS
 - Declaración de constantes → los valores que pueden tener los campos de bits
 - Declaración de máscaras → para modificar campos de bits en registros
 - Registros con la misma estructura de bits se declaran solo una vez.
2. Declaración de ESTRUCTURA DE REGISTROS
 - Definir macro de estructura asociado a la dirección del módulo
 - Definir macros de registros
 - Definir macros de campos de bits
 - LAS ESTRUCTURAS MÓDULOS CON MÚLTIPLES INSTANCIAS (EJ. SPI) SE DECLARAN UNA SOLA VEZ, pero...
 - LOS MACROS DE ESTRUCTURA Y LOS RESPECTIVOS } SE DEBEN DECLARAR
 - MACROS DE REGISTROS } POR CADA INSTANCIA !

Resulta que...

Cuando yo creo estructuras de registros que son uniones, eso se va haciendo:

VARCΦ - ^{NOMBRE}_{REGISTRO} • TIPO (word o bit). campo de bits } Y de pronto esto se vuelve muy LARGO
ACCESO } } => lo que se hace es definir unos
MACROS que simplifican eso !

NOMENCLATURA DE LAS LIBRERIAS DEL CURSO

- ESTRUCTURA MÓDULO ⇒ S + NOMBRE MÓDULO + [NÚMERO MÓDULO] → S de estructura
Ej: SADCΦ, SPIT, STPM1 si pongo S + CTRL + SPACE } va a mostrarme estructuras que existen en el MCV
- REGISTROS ⇒ S + NOMBRE MÓDULO + [NÚMERO MÓDULO] + _ + NOMBRE REGISTRO
Ej: SADCΦ_SCIA, SPIT_MCR SADC_ + CTRL SPACE } lista todos los registros para ese periférico
la cosa sigue --
- CAMPO DE BITS EN REGISTRO ⇒
b + NOMBRE MÓDULO + [NÚMERO MÓDULO] + _ + NOMBRE ~~REGISTRO~~
CAMPO BITS } NO SIEMPRE SE PUEDE HACER ESTO!
Ej: bADCΦ_ADICLK, bPIT_MDIS

porque si uno tiene un campo de bits que se repite en varios registros
pues todo meter el "nombre del Registro". Pero en la mayoría de
los casos los campos de bits son "uno solo" → por ejemplo que se llama
ADICLK → para ese módulo (ADC0), entonces no se requiere poner el nombre registro
ya que la idea es "hacerlo más corto".
Pero pueden existir casos donde un campo de bits aparece varias veces en
varios registros → allí no sirve esta nomenclatura!

- **Constantes** → son los valores que le podemos dar a los campos de bits

$k + \text{NOMBREmódulo} + \dots + \text{NOMBREcampo bits} + \dots + \text{NOMBREconstante}$

Ej: $k\text{ADC_ADCH_DADP0}$, $k\text{TPM_PS_DIVIDE_BY_1}$

- **Macros de campos de bits**

$m + \text{NOMBREmódulo} + \dots + \text{NOMBREregistro} + \dots + \text{NOMBREcampo bits} + (\text{VALOR})$

Ej: $m\text{TPM_SC_PS}(z)$

VER Ejemplos:

Ej: ADC_LIB para KL27Z

BUSCTRL_LIB para RPP

LPOART_LIB para KL27Z

GPIO_LIB para KL27Z.