Propuesta del curso de

“DISEÑO DE SISTEMAS CON PROCESADORES” - DISPRO

a partir del primer semestre de 2023

Versión de documento a Enero 2 de 2023

Ing. JUAN-CARLOS GIRALDO, Ph.D.

Página dejada en blanco de manera intencional.

COMPETENCIAS AL FINAL DE "DISEÑO DE SISTEMAS CON PROCESADORES"

COMPETENCIA 1: Representar cualquier dato numérico y alfabético mediante una codificación binaria.

Comentario: Antes de desarrollar la siguiente competencia 2, el estudiante entiende que cualquier dato se codifica en binario, incluyendo las instrucciones en lenguaje máquina. En el capítulo 2, a partir del Lenguaje C generará lenguaje de ensamble y su equivalente código máquina.

COMPETENCIA 2: Hacer ingeniería inversa a un código máquina para entender su código fuente con propósitos académicos, considerando sus implicaciones éticas.

Comentario: Para lograr esta competencia debe primero entender el flujo de compilación de Lenguaje C y las herramientas usadas en el "*toolchain*". Describe los tipos de instrucciones fundamentales de un lenguaje de ensamble, así como explica la arquitectura básica de un procesador. Determina el uso de variables en memoria. Entiende y usa el ambiente en tiempo de ejecución.

COMPETENCIAS 3: Analizar y diseñar un sistema de computador embebido.

Comentario: Entiende y usa el "*datasheet*" de la arquitectura de un microcontrolador. Entiende y elabora diagramas de bloques, diagramas de tiempos y diagramas esquemáticos. A partir de la información del "*datasheet*", configura un periférico. Para enfrentar un diseño, entiende las generalidades de las modernas metodologías de diseño de sistemas de computador embebido.

COMPETENCIAS 4: Diseñar, adaptar e implementar en software cualquier algoritmo mediante una "*Finite State Machine*".

Comentario: A diferencia de la programación tradicional en PC, el estudiante entiende que la programación a nivel de sistema puede ser poco o nada estructurada. Las *FSM* son más aptas para esta aproximación, en contraste con una aproximación mediante el uso de "*flowcharts*".

COMPETENCIA 5: Justificar la elección de una arquitectura de software embebido, acorde a los requerimientos del diseño.

Comentario: De las cuatro arquitecturas comúnmente usadas en la práctica, a saber:

- "*Round-robin*"
- "*Round-robin*" con interrupciones
- "*Function-Queue-Scheduling*"
- Sistema Operativo en Tiempo-Real (*RTOS*)

… el estudiante justifica, implementa y usa la más conveniente según requerimientos del diseño.

GRADUALIDAD DE COMPETENCIA PARA “DISEÑO DE SISTEMAS CON PROCESADORES”

COMPETENCIA 1: Representar cualquier dato numérico y alfabético mediante una codificación binaria.

Usa codificación binaria para:

- Representar y operar números de cualquier tipo (negativos, punto fijo y flotante)
- Representar códigos alfanuméricos *ASCII*, secuencias de escape, Unicode, etc.
- Leer instrucciones en código máquina

COMPETENCIA 2: Hacer ingeniería inversa a un código máquina para entender su código fuente con propósitos académicos, considerando sus implicaciones éticas.

- Escribe un código en lenguaje de ensamble con supervisión y ayuda
- Explica un código en lenguaje de ensamble
- Usa el “*stack frame*” para desarrollar funciones en lenguaje de ensamble
- Explica la arquitectura básica de un procesador
- Usa los comandos de Linux para compilar en el “*toolchain*” de C

COMPETENCIAS 3: Analizar y diseñar un sistema de computador embebido.

- Diseña el “*driver*” para cualquier periférico
- Configura cualquier periférico de un micro-controlador
- Comunica componentes (p.e. memoria y microcontrolador por *I2C*) y/o sistemas
- Explica una arquitectura al igual que sus periféricos
- Entiende, analiza y desarrolla diagramas para hacer diseños en ingeniería
- Usa metodologías para diseñar sistemas embebidos

COMPETENCIAS 4: Diseñar, adaptar e implementar en software cualquier algoritmo mediante una “*Finite State Machine*”.

- Diseña *FSM* para detección de borde
- Diseña *FSM* para medición de características de una señal digital periódica
- Diseña *FSM* para generación de señales *PWM*

COMPETENCIA 5: Justificar la elección de una arquitectura de software embebido, acorde a los requerimientos del diseño.

- Extrae los requerimientos de sistemas embebidos
- Explica Condición atómica, tareas apropiativas (“*preemptive*”)

PROGRAMA GENERAL DE “DISEÑO DE SISTEMAS CON PROCESADORES”

CAPÍTULO 1. CODIFICACIÓN DE BAJO NIVEL

1.1. Sistemas de Numeración
1.2. Representación de Números en un Sistema Digital
1.3. Formatos Estándar IEEE-754 de Punto Flotante
1.4. Codificación Binaria
1.5. Secuencias de Conteo

CAPÍTULO 2. PROGRAMACIÓN EN BAJO NIVEL

2.1. Herramientas de Desarrollo en C
2.2. Control de Flujo en Bajo Nivel
2.3. Casos Especiales de Control de Flujo
2.4. Operaciones de la ALU: *Expresiones en C*
2.5. Tipos de Datos en Memoria: *Declaraciones en C*
2.6. Llamado con Retorno y Contexto en Pila: *Funciones en C*

CAPÍTULO 3. METODOLOGÍAS DE DISEÑO

3.1. Ingeniería de Software y de Hardware
3.2. Elaboración y Lectura de Diagramas
3.3. Sistemas de Computador Embebido
3.4. Configuración de Periféricos
3.5. Protocolos de Comunicación

CAPÍTULO 4. MÁQUINAS DE ESTADOS FINITOS

4.1. Diagramas de Estado
4.2. Tipos de Máquinas de Estados Finitos
4.3. Implementación de Máquinas de Estados Finitos
4.4. Máquinas de Estados Finitos Concurrentes

CAPÍTULO 5. INTRODUCCIÓN A RTOS

5.1. Interrupciones Hardware
5.2. Arquitecturas de Software
5.3. Servicios de una Arquitectura RTOS
5.4. Otros Servicios del Sistema Operativo
5.5. Uso de un Diseño Básico con RTOS

PROGRAMA DETALLADO DE "DISEÑO DE SISTEMAS CON PROCESADORES"

CAPÍTULO 1. CODIFICACIÓN DE BAJO NIVEL

1.1. Sistemas de Numeración
- 1.1.1. Historia de los sistemas de numeración
- 1.1.2. Sistema numérico posicional
- 1.1.3. Conversión entre bases numéricas

1.2. Representación de Números en un Sistema Digital
- 1.2.1. Precisión y rango dinámico
- 1.2.2. Representación de números negativos
- 1.2.3. Representación en punto fijo y punto flotante
- 1.2.4. Representación logarítmica
- 1.2.5. Aritmética binaria

1.3. Formatos Estándar IEEE-754 de Punto Flotante
- 1.3.1. Justificación, tipos de formatos y actualizaciones
- 1.3.2. Codificación de los formatos
- 1.3.3. Valores y operaciones especiales
- 1.3.4. Operaciones aritméticas

1.4. Codificación Binaria
- 1.4.1. Justificación y usos
- 1.4.2. Códigos ponderados, no-ponderados, auto-complementados
- 1.4.3. Códigos alfanuméricos
- 1.4.4. Códigos detectores y correctores de error

1.5. Secuencias de Conteo
- 1.5.1. Secuenciadores hardware
- 1.5.2. Contadores binario y Gray
- 1.5.3. Contadores de anillo y Johnson
- 1.5.4. Contadores pseudo-aleatorios

CAPÍTULO 2. PROGRAMACIÓN EN BAJO NIVEL

2.1. Herramientas de Desarrollo en C
- 2.1.1. Sistema operativo por consola
- 2.1.2. El pre-procesador de C
- 2.1.3. Etapas de compilación en C
- 2.1.4. Herramientas software
- 2.1.5. Herramientas hardware

CONCEPTOS CLAVE DE CADA SECCIÓN DEL CURSO

CAPÍTULO 1. CODIFICACIÓN DE BAJO NIVEL

1.1. Sistemas de Numeración

1.1.1. Historia de los sistemas de numeración
1.1.2. Sistema numérico posicional
1.1.3. Conversión entre bases numéricas

Conceptos clave:

Tomados del resumen de la sección 1.1: "*Number System*" [Giraldo, 2023].

- Un **sistema numérico** es un sistema de escritura para representar números.
- Según Charles Petzold, el sistema numérico hindú-árabe difería de los sistemas numéricos anteriores en tres aspectos fundamentales: (1) El sistema numérico hindú-árabe **es un sistema numérico posicional**, (2) Todos los primeros sistemas numéricos tienen algo que **el hindú-árabe sistema no tiene**, que es el símbolo del número diez, (3) Todos los primeros sistemas numéricos **carecen de algo que tiene el sistema hindú-árabe tiene**, y es el número cero.
- La invención del cero es uno de los conceptos abstractos más fundamentales de la aritmética no solo para representar cantidades sino para facilitar las operaciones aritméticas como el producto de dos números.
- Un **guarismo** es un signo gráfico simple que expresa un número en un sistema numérico.
- Un **sistema de numeración posicional** es un sistema de expresión de números en el que los dígitos están ordenados en sucesión, la posición de cada dígito tiene un valor posicional y el número es igual a la suma de los productos de cada dígito por su lugar.
- En un sistema de numeración posicional, la **raíz o base** es el número de dígitos únicos, incluido el dígito cero, que se utilizan para representar números.

1.2. Representación de Números en un Sistema Digital

1.2.1. Precisión y rango dinámico
1.2.2. Representación de números negativos
1.2.3. Representación en punto fijo y punto flotante
1.2.4. Representación logarítmica
1.2.5. Aritmética binaria

Conceptos clave:

Tomados del resumen de la sección 1.1: "*Number System*" [Giraldo, 2023].

- El rango dinámico especifica el rango de valores posibles o aceptables que una señal dinámica puede asumir cuando es entregada o producida por un sistema dado.
- El rango dinámico se define como la relación entre los valores más grandes y más pequeños que un ADC puede medir de manera confiable. Para un ADC, el rango dinámico está relacionado con la cantidad de bits que se utilizan para digitalizar la señal analógica. Considere un ADC ideal de N bits. El valor mínimo que se puede detectar es un bit menos significativo (LSB). El valor máximo es $2^N - 1$ veces el valor LSB. Por lo tanto, en términos de decibeles, el rango dinámico del ADC será

$$20\log_{10}\left[\frac{(2^N - 1)LSB}{LSB}\right] \approx 6.02 \times N(dB)$$

  Por lo tanto, con un ADC de 10 bits, esperaríamos un rango dinámico de 60,2 dB. Esto significa que el ADC podría resolver amplitudes de señal desde x hasta aproximadamente 1000x, donde x es el mínimo que se puede detectar. https://www.allaboutcircuits.com/technical-articles/understanding-the-dynamic-range-specification-of-an-ADC/
- Se conocen cuatro métodos principales para extender el sistema binario para representar números con signo:
  - Signo y magnitud.
  - Complemento a uno.
  - Complemento a dos.
  - Exceso de K, donde K generalmente es igual a $b^{n-1} - 1$.
- **Complemento a uno** es transformar dígitos de uno a cero y de cero a uno.
- **Complemento a dos** es hacer complemento a uno con adición de la unidad.

1.3. Formatos Estándar IEEE-754 de Punto Flotante

1.3.1. Justificación, tipos de formatos y actualizaciones
1.3.2. Codificación de los formatos
1.3.3. Valores y operaciones especiales
1.3.4. Operaciones aritméticas

Conceptos clave:

Tomados del resumen de la sección 1.2: "*Real Number Representation*" [Giraldo, 2023].

- Los dos tipos de aritmética que se realizan en las computadoras son la aritmética de enteros y la aritmética real.
- El número de bits necesarios para codificar 8 dígitos decimales es de aproximadamente 26, ya que log(2) 10 = 3,32 bits se necesitan en promedio para codificar un solo dígito decimal.
- La mantisa o significando es el componente de un número finito de punto flotante que contiene sus dígitos significativos. La mantisa se puede considerar como un número entero, una fracción o alguna otra forma de punto fijo, eligiendo una compensación o sesgo de exponente apropiada. Un significado binario decimal o subnormal también puede contener ceros a la izquierda [IEEE754].
- El exponente es el componente de una representación finita de punto flotante que representa la potencia entera a la que se eleva la base para determinar el valor de esa representación de coma flotante. El exponente e se usa cuando la mantisa se considera un dígito entero y un cuerpo fraccionario, y el exponente q se usa cuando la mantisa se considera un número entero; $e = q + p - 1$ donde p es la precisión del formato en dígitos [IEEE754].
- Los números de punto flotante se utilizan para representar números fraccionarios no enteros en la mayoría de los cálculos técnicos y de ingeniería. Los números binarios de punto flotante comenzaron a usarse a mediados de la década de 1950.
- El estándar IEEE para la aritmética de punto flotante (referido como IEEE 754) es un estándar técnico establecido en 1985 por el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE). Una vez que fue adoptado por todos los fabricantes de computadoras, hizo posible portar programas de una computadora a otra sin que los resultados fueran diferentes. Hoy en día, la mayoría de las unidades de coma flotante (FPU) en hardware utilizan el estándar IEEE 754.
- El IEEE 754-1985 fue la primera definición del estándar. El IEEE 854-1987 es el estándar para la aritmética de punto flotante independiente de Radix. El IEEE 754-2008 incluye casi todo el original más el IEEE 854-1987. El IEEE 754-2019 proporcionó una revisión menor del anterior, incorporando aclaraciones.

- El exponente sesgado es la suma del exponente y una constante (sesgo) elegida para hacer que el rango del exponente sesgado no sea negativo [IEEE754].
- Una forma compacta de representar el número es la siguiente: R=(−1)^B ×2^(E-127)×(1·m)2
- "*Not a Number*" es un dato simbólico de punto flotante. Hay dos tipos de representaciones de NaN: silenciosas y de señalización. La mayoría de las operaciones propagan NaN silenciosos sin excepciones de señalización y señalan la excepción de operación no válida cuando se les proporciona un operando de NaN de señalización [IEEE754].
- En un formato particular, un número de punto flotante distinto de cero con una magnitud menor que la magnitud del número normal más pequeño de ese formato. Un número subnormal no usa la precisión completa disponible para los números normales del mismo formato [IEEE754].

1.4. Codificación Binaria

1.4.1. Justificación y usos
1.4.2. Códigos ponderados, no-ponderados, auto-complementados
1.4.3. Códigos alfanuméricos
1.4.4. Códigos detectores y correctores de error

Conceptos clave:

Tomados del resumen de la sección 1.3: "*Binary Codes*" [Giraldo, 2023].

- Un código numérico es un sistema de codificación cuyo alfabeto objetivo contiene solo dígitos y/o cadenas de dígitos.
- Los códigos numéricos se pueden clasificar en: (1) códigos ponderados y no ponderados y (2) códigos autocomplementarios y secuenciales.
- Los caracteres alfanuméricos son una combinación de caracteres alfabéticos y numéricos.
- Los códigos alfanuméricos proporcionan dos funciones principales: (1) mostrar información en dispositivos de salida, (2) representar símbolos alfanuméricos en un texto.
- Los tres principales tipos de códigos alfanuméricos con ASCII, EBCDIC y Unicode.
- ASCII (abreviado de "*American Standard Code for Information Interchange*") es una codificación y un juego de caracteres desarrollado por ASA en la década de 1960.
- El código de intercambio decimal codificado en binario extendido (EBCDIC) es un código binario de 8 bits para caracteres numéricos y alfanuméricos. Fue desarrollado y utilizado por IBM. Es una representación de codificación en la que símbolos, letras.
- Unicode y el conjunto de caracteres universales (UCS) ISO/IEC 10646 tienen una gama mucho más amplia de caracteres y sus diversas formas de codificación han comenzado a suplantar rápidamente a ISO/IEC 8859 y ASCII en muchos entornos. Mientras que ASCII está limitado a 128 caracteres, Unicode y UCS admiten más caracteres al separar los conceptos de identificación única (usando números naturales llamados puntos de código) y codificación (a formatos binarios de 8, 16 o 32 bits, llamados UTF-8). , UTF-16 y UTF-32).

1.5. Secuencias de Conteo

1.5.1. Secuenciadores hardware
1.5.2. Contadores binario y Gray
1.5.3. Contadores de anillo y Johnson
1.5.4. Contadores pseudo-aleatorios

Conceptos clave:

Tomados del resumen de la sección 1.4: "*Counting*" [Giraldo, 2023].

- Contar es la acción de encontrar el número de elementos de un conjunto finito de objetos. La forma tradicional de contar consiste en aumentar continuamente un contador de una unidad para cada elemento del conjunto, en cierto orden, mientras marcan esos elementos para evitar visitar el mismo elemento más de una vez, hasta que no queden elementos sin marcar. Si el contador se estableció en uno después del primer objeto, el valor después de visitar el objeto final proporciona el número deseado de elementos.
- Los principales sistemas de conteo son contador binario de N-Bit, contador con código Gray, contador de anillo y contador Johnson.
- Los contadores de código binario y Gray de N-bit se pueden extender naturalmente a la longitud de varios bits n, también se pueden usar para contar y representar secuencias que permiten determinar el siguiente valor a partir del valor de la actual.
- Un número binario representado en un contador de anillos también se conoce como un código único. El código único consiste en un grupo de bits entre los cuales las combinaciones legales de valores son solo aquellas con un solo bit en uno y todos los demás con cero. Una implementación similar en la que todos los bits son uno excepto el cero único en la secuencia a veces se llama un solo plato.
- Un contador de Johnson es un contador de anillo modificado, donde la salida invertida del último bit está conectada a la entrada del primero. El registro se dispara a través de una secuencia de patrones de bits. El mod del mostrador Johnson es $2n$ si se usan n bits. La principal ventaja del mostrador Johnson es que solo necesita la mitad del número de bits en comparación con el contador de anillo estándar para el mismo mod.

CAPÍTULO 2. PROGRAMACIÓN EN BAJO NIVEL

Conceptos clave sobre el lenguaje C:

Tomados del resumen del capítulo 1, titulado en inglés "*C Refresher*", de [Anderson & Anderson, 1988], pp. 42-43.

- En el tipo de dato entero "unsigned" sin signo el compilador trata el bit más significativo como dato.
- Los datos de tipo entero tienen formatos diferentes a los datos de tipo en punto flotante. Si el compilador interpreta un número entero como punto flotante (o viceversa), los programas pueden producir resultados incorrectos.
- C no convierte los enteros a punto flotante ni viceversa, en los parámetros de las funciones.
- Si una función retorna algo que no sea un número entero, debe informar al compilador del tipo de dato de retorno de la función antes de usarla.
- C proporciona operadores para expresiones aritméticas, relacionales y lógicas. Los operadores bit a bit manipulan bits y los operadores de asignación compuesta proporcionan expresiones compactas. Los operadores condicionales son útiles en parámetros de funciones y macros.
- C proporciona construcciones de flujo de control para decisiones de múltiples opciones, bucles y bifurcaciones. Las declaraciones "if-else" evalúan expresiones sucesivas hasta que una, si alguna, se evalúa como distinta de cero. Un "switch" con una gran cantidad de etiquetas "case" puede ser más eficiente que su equivalente que usa "if-else" ya que el compilador produce tablas de salto en código de ensamble.
- "break" y "continue" proporcionan control sobre el procesamiento de bucles. "goto" debe usarse con moderación, pero puede ser útil para salir rápidamente de una construcción profundamente anidada.
- C admite arreglos simples y multidimensionales. Los arreglos pueden contener tipos agregados así como tipos de datos simples.
- Los apuntadores son variables que contienen direcciones de objetos en memoria. C proporciona los operadores *, &, ->, ++ y -- para manipular apuntadores en los programas.
- Los programas en C comienzan a ejecutarse en "main()", que es una función con dos argumentos. El primer argumento es el número de argumentos en la línea de comando que invocó el programa, y el segundo argumento es un arreglo de apuntadores a caracteres que contienen la dirección de cada argumento, incluyendo el nombre del archivo ejecutable.
- Las estructuras agrupan diferentes miembros en un solo objeto. El tamaño de una estructura es la cantidad de espacio requerido para almacenar todos los miembros. Las uniones son como estructuras, pero el compilador asigna solo el

espacio suficiente para contener el miembro más grande de la unión. Las estructuras y uniones tienen la misma sintaxis.

- C proporciona clases de almacenamiento para que los programas puedan controlar dónde residen las variables en el entorno en tiempo de ejecución. Las cuatro clases de almacenamiento son automático (“auto”), estático (“static”), de registro (“register”) y externo (“extern”).
- Una función o bloque debe declarar primero las variables de registro más importantes y colocar cada declaración de registro en una línea separada para efectos de portabilidad.
- Las directivas del preprocesador permiten la compilación condicional de líneas en un archivo fuente. El preprocesador tiene una sintaxis distinta de la de C.
- Las macros se ejecutan más rápido que las funciones, pero ocupan más espacio de código. No todas las funciones se convierten en macros.
- La conversión de tipo con “type cast” cambia los tipos de datos en tiempo de ejecución. Los programas usan “type cast” para parámetros de funciones, valores devueltos y conversiones aritméticas.

2.1. Herramientas de Desarrollo en C

2.1.1. Sistema operativo por consola
2.1.2. El pre-procesador de C
2.1.3. Etapas de compilación en C
2.1.4. Herramientas software
2.1.5. Herramientas hardware

Conceptos clave para las secciones 2.1.2, 2.1.3 y 2.1.4:

Tomados del resumen del capítulo 5, titulado en inglés "*C Debugging Techniques*", de [Anderson & Anderson, 1988], pp. 318-319.

- La opción de pre-procesamiento -D del compilador de C es útil para la depuración. Esta opción de línea de comando permite que los programas compilen condicionalmente código de depuración en archivos fuente.
- Los manejadores ("*drivers*") de test integrados (dentro del código fuente) permiten que los módulos se prueben a sí mismos.
- En C "*signals*" permiten a los usuarios escribir caracteres del teclado y hacer que los programas en ejecución descarguen la salida para depuración de programas.
- Una aserción ("*assertion*") es una expresión booleana que se espera que sea verdadera en tiempo de ejecución.
- Las aserciones detectan errores de tiempo de ejecución. Si las aserciones fallan, los programas muestran mensajes de error y terminan.
- Las aserciones ayudan a detectar errores fuera de los límites de los arreglos y comprueban si hay valores de apuntadores no válidos.
- Las aserciones son macros. Los usuarios pueden crear sus propias aserciones.
- La opción del compilador -DNDEBUG elimina las aserciones de los programas.
- El preprocesador sustituye el nombre del archivo fuente actual y el número de línea actual por los símbolos __FILE__ y __LINE__ respectivamente. El nombre del archivo es una constante de cadena y el número de línea es una constante entera. Los programas de usuario pueden utilizar esta información en los mensajes de error.
- La impresión de depuración selectiva permite que los programas muestren la salida de depuración sin volver a compilar los módulos.
- Los "front-ends" brindan verificación de errores adicional para las rutinas de la biblioteca C.

Conceptos clave para las secciones 2.1.4 y 2.1.5:

Tomados del resumen del capítulo 9, titulado en inglés "*Embedded Software Development Tools*", de [Simon, 1999], pp. 280-281.

- El desarrollo de software embebido generalmente se realiza en una máquina "*host*", diferente de la máquina de destino en la que el software finalmente se despacha a los usuarios.
- Una cadena de herramientas para desarrollar software embebido normalmente contiene:
  - Un compilador cruzado.
  - Un ensamblador cruzado.
  - Un encadenador/localizador.
  - Un método para cargar el software en la máquina de destino.
- Un compilador cruzado entiende el mismo lenguaje C que un compilador nativo (con algunas excepciones), pero la salida usa el conjunto de instrucciones del microprocesador de destino.
- Un ensamblador cruzado entiende un lenguaje ensamblador específico para el microprocesador de destino y genera instrucciones para ese microprocesador.
- Un encadenador/localizador combina módulos compilados y ensamblados por separado en una imagen ejecutable. Además, coloca código, datos, código de inicio, cadenas constantes, etc., en direcciones adecuadas en ROM y RAM.
- Los encadenadores/localizadores usan segmentos para decidir dónde colocar las diferentes partes del código y los datos.
- Los encadenadores/localizadores producen resultados en varios formatos. Depende del desarollador asegurarse de que la salida de el encadenador/localizador sea compatible con las herramientas que utiliza para cargar software en el objetivo.
- Debe encontrar una manera de cargar el software en el sistema de destino para realizar pruebas. Las formas más comunes incluyen programadores de PROM, emuladores de ROM, emuladores en circuito, memoria flash y monitores.

2.2. Control de Flujo en Bajo Nivel

2.2.1. Diagramas de flujo
2.2.2. Instrucciones, condiciones y secuencias
2.2.3. Bloque de instrucciones (delimitadores "{" y "}")
2.2.4. Secuencias lineales (separador ";")
2.2.5. Secuencias de selección ("if-else", "if")
2.2.6. Secuencias de repetición ("for", "while", "do-while")

Conceptos clave:

- Mientras que un programador piensa en el **control de flujo** mediante las secuencias básicas de la **programación estructurada** en los lenguajes de alto nivel, el ingeniero electrónico que programa en lenguaje de ensamble piensa en **saltos** que rompen el flujo de ejecución de las instrucciónes máquina a ciertas posiciones de memoria diferentes de la siguiente instrucción a ejecutar.
- El paradigma de la **programación estructurada** fue creado para abstraer (o esconder) los **saltos** en algunos lenguajes que proporcionan la instrucción “goto-label” ó el mnemónico “JMP” de los lenguajes de ensamble.
- Las tres secuencias fundamentales para control de flujo en la **programación estructurada** son:
  - **Secuencia lineal**: sigue la ejecución de instrucciones según las encuentra en una secuencia ordenada en memoria una tras otra.
  - **Secuencia de selección**: salto condicionado hacia adelante para seleccionar bloques de instrucciones que han de ejecutarse en lugar de otros, según se cumpla una condición booleana.
  - **Secuencia de repetición**: salto condicionado hacia atrás que permite repetir la ejecución de bloques de instrucciones por los que ya se ha pasado previamente, mientras se cumpla cierta condición booleana.
- El lenguaje C ofrece dos construcciones básicas para implementar secuencias de selección:
  - Secuencia de **selección binaria** mediante el uso de “if-else”.
  - Secuencia de **selección con “by-pass”** mediante el uso de “if”.
- El lenguaje C ofrece tres construcciones básicas para implementar secuencias de repetición ó iteración:
  - Secuencia de **repetición con conteo controlado** con “for”.
  - Secuencia de **repetición de pre-condición con sentinela** con “while”.
  - Secuencia de **repetición de post-condición con sentinela** con “do-while”.
- El registro “*Program Counter*” (PC) de la unidad de control de la CPU es el encargado de indicar la dirección de memoria de la siguiente instrucción máquina a ejecutar.

- Las instrucciones de salto en lenguaje máquina modifican la dirección de memoria guardada en el "*Program Counter*" (PC), que corresponde a la dirección de la siguiente instrucción por defecto a ejecutar.
- En lenguaje de ensamble, y su equivamente código máquina, existen cuatro formas de desviar el flujo lineal de un programa:
  - Saltos **incondicionales** (JPM).
  - Saltos **condicionados** (JNZ).
  - Saltos **con retorno** (CALL, RET) guardando el contenido del "*Program Counter*" (PC) en el "stack", previo al salto y recuperando luego el PC del "stack" para retorno a la instrucción original posterior al salto.
  - Saltos **por interrupción** ocasionados por eventos externos.

2.3. Casos Especiales de Control de Flujo

2.3.1. Árboles de decisión
2.3.2. Selección con "switch-case"
2.3.3. Optimización de bucle
2.3.4. Saltos locales ("goto-label", "break", "continue", "return")
2.3.5. Saltos no-locales (setjmp.h)
2.3.6. Resumen: "Secuencias en Lenguajes de Ensamble"

Conceptos clave:

- Un árbol de decisión es un método analítico a través del cual una representación esquemática de las alternativas disponibles facilita la toma de decisiones.
- La secuencia de selección “switch-case” es una estructura que evalúa más de un caso de una expresión entera y se caracteriza por:
  - Selección de una opción entre varias.
  - El “switch” recibe un “case” y lo evalúa hasta encontrar el caso que corresponda.
  - Se puede usar la opción “default” para cuando no se encuentra el caso dado.
- En la teoría de diseño de compiladores, la optimización de ciclos de repetición es un proceso para aumentar la velocidad de ejecución y reducir el costo computacional asociados a la ejecución de ciclos de iteración. Estas técnicas desempeñan un papel fundamental en la mejora del desempeño de la memoria caché y en el uso eficaz de las capacidades de procesamiento paralelo.
- Los saltos locales son construcciones en lenguaje C para desviar el flujo de control dentro de las funciones.
- El lenguaje C permite hacer saltos no-locales a otras funciones mediante el uso de la biblioteca “setjmp.h”
- Los saltos no locales mediante “setjmp/longjmp” se usan para implementar mecanismos de excepción que con “longjmp” restablecen el estado del programa o hilo, incluso a través de múltiples niveles de llamadas a funciones. Un uso menos común de “setjmp” es crear una sintaxis similar a corrutinas.

2.4. Operaciones de la ALU: Expresiones en C

2.4.1. Operandos y operadores
2.4.2. Precedencia
2.4.3. Asociación
2.4.4. Evaluación
2.4.5. Promoción
2.4.6. Puntos de secuencia garantizada

Conceptos clave:

- Una **expresión** es una secuencia de **operadores** y **operandos** que especifica el cálculo de un valor, o designa **objeto** o **función**, o genera **efectos laterales** o realiza combinación de lo anterior [ISO/IEC 9899] pag. 67.
- Los **operadores** fundamentales de las **expresiones** en lenguaje C que calculan un valor pueden ser considerados como el conjunto de posibles operaciones que una ALU puede realizar en un procesador.
- La ALU ("*Aritmetic Logic Unit*" o en español Unidad Aritmético-Lógica) es un bloque fundamental del procesador que hace posible la realización de cada una de las operaciones de un computador.
- No solo es suficiente saber los **operadores** nativos del lenguaje C sino los conceptos de **precendencia**, **asociación**, **evaluación**, **promoción** y **puntos de secuencia garantizada** para una completo dominio de la evaluación de las expresiones en C.

2.5. Tipos de Datos en Memoria: Declaraciones en C

2.5.1. Atributos de tipos de datos
2.5.2. Tipos escalares, agregados y derivados
2.5.3. Apuntadores
2.5.4. Estructuras
2.5.5. Arreglos
2.5.6. Declaraciones complejas

Conceptos clave:

- Los **tipos aritméticos** guardan y operan datos de tipos entero y de punto flotante: “int” y “float”.
- Un apuntador no es más que un tipo de dato que guarda la dirección de una posición de memoria.
- Los tipos aritméticos junto a los apuntadores conforman los **tipos escalares**.
- Los arreglos y estructuras corresponden a **tipos agregados**. Los arreglos agrupan datos de un mismo tipo, mientras que las estructuras agrupan datos de diferente tipo.
- Los apuntadores, funciones y arreglos conforman los **tipos derivados**.

Conceptos clave para la sección 2.5.5:

Tomados del resumen del capítulo 3, titulado en inglés “*An Array of Choices*”, de [Anderson & Anderson, 1988], pp 200-202.

- C convierte todas las referencias para indexar arreglos en saltos ó desplazamientos (“*offsets*”) de apuntadores respecto a una dirección base.
- La regla básica establece que “a[i]” es equivalente a “(*(a + i))” para un arreglo “a” de cualquier tipo de dato.
- La regla básica ayuda a descifrar las referencias de arreglos con direcciones indirectas de apuntadores. Recuerde conservar los paréntesis exteriores si el orden de evaluación de C cambia cuando los omite en una expresión.
- C usa escalares para calcular los desplazamiento ("*offsets*") de los apuntadores y sustraer los apuntadores que apuntan a tipos de datos similares.
- Las expresiones de apuntador compacto utilizan autoincremento (++) y autodecremento (--) con direccionamiento indirecto de apuntador (*). Las expresiones de apuntador compacto se dividen en dos grupos. El primer grupo de expresiones de apuntador compacto es

*p++ *p-- *++p *--p

Estas expresiones cambian el apuntador y no el objeto apuntado. Si los apuntadores apuntan a arreglos, se pueden usar en lugar de referencias a arreglos. Estas expresiones mejoran la velocidad de ejecución dentro de los bucles. El segundo grupo de expresiones de apuntadores compactos es

++*p --*p (*p)++ (*p)--

Estas expresiones incrementan o decrementan el objeto apuntado. El apuntador no se ve afectado. Las expresiones de apuntador en este grupo son más restrictivas. Los objetos no pueden ser estructuras o uniones. Si el objeto es un apuntador, las expresiones del segundo grupo mejoran la velocidad de ejecución dentro de los bucles.

- Los subíndices de arreglos negativos son legales. C no verifica las referencias fuera de los límites del arreglo.
- Los arreglos bidimensionales tienen filas y columnas. C asigna el almacenamiento por filas, por lo que el segundo subíndice varía más rápido.
- Si “a” es una matriz bidimensional de números enteros con “n” filas e “i” es un número entero, entonces “a[i]” es un apuntador a un número entero (apunta al comienzo de la fila “i”).
- Las ecuaciones de mapas de almacenamiento bidimensionales tienen un paso de multiplicación. Dependiendo de la máquina, las referencias de matriz con una variable para el subíndice de fila generalmente hacen que el compilador genere una instrucción de multiplicación o cambio y suma en código ensamblador.
- Cuando se pasa la dirección de un arreglo bidimensional a una función, el compilador pasa un apuntador a la primera fila.
- Los arreglos tridimensionales tienen grillas, filas y columnas. El almacenamiento es en forma de grillas, luego en forma de filas, por lo que el tercer subíndice varía más rápido.
- Suponga que “a” es un arreglo tridimensional de números enteros con grillas “g”, filas “r” y columnas “c”. Suponiendo que “i” y “j” son enteros, “a[i]” es un apuntador a un arreglo de “c” enteros. “a[i][j]” es un apuntador a un número entero.
- Las ecuaciones del mapa de almacenamiento tridimensional tienen dos pasos de multiplicación. Dependiendo de la máquina, las referencias de arreglos con variables para la grilla y los subíndices de fila generalmente hacen que el compilador genere instrucciones de multiplicación o cambios y sumas en código ensamblador.
- Cuando se pasa la dirección de una arreglo tridimensional a una función, el compilador pasa un apuntador a la primera grilla.
- En la mayoría de las máquinas, las expresiones de apuntador compacto se ejecutan más rápido que las referencias de matrices bidimensionales o tridimensionales dentro de los bucles.

- La declaración de arreglos de apuntadores en lugar de arreglos bidimensionales y tridimensionales elimina las ecuaciones del mapa de almacenamiento para las referencias de arreglos.
- Se puede convertir programas con declaraciones de arreglos bidimensionales y tridimensionales en arreglos de apuntadores. Esto no afecta la notación de arreglo y el compilador usa desplazamientos de apuntadores en lugar de ecuaciones de mapa de almacenamiento. Las declaraciones dentro de las funciones que acceden a la matriz deben cambiar al igual que las declaraciones externas con “extern”.
- Los arreglos en tiempo de ejecución usan almacenamiento en el "heap" de memoria. Los programas realizan una llamada de biblioteca al administrador de almacenamiento dinámico para un arreglo unidimensional. Los arreglos bidimensionales requieren una llamada para asignar los datos y una llamada adicional para un arreglo de filas de apuntadores. Las arreglos tridimensionales requieren una tercera llamada para una matriz de grilla de apuntadores. Conectar las matrices de apuntadores a los datos hace que el almacenamiento en el "heap" asignado parezca un arreglo multidimensional. El compilador usa desplazamiento de apuntadores para las referencias de arreglos.
- Los programas pueden crear matrices multidimensionales de cualquier tamaño y cualquier tipo en tiempo de ejecución con una macro o una función.
- Las funciones de C deben usar almacenamiento en el "heap" de memoria y arreglos de apuntadores para imitar la convención de llamada a subrutinas usada en FORTRAN y BASIC. Para acceder a elementos de arreglos de diferentes tamaños declarados en tiempo de compilación, una función debe configurar arreglos de apuntadores en el "heap". Las funciones pueden acceder a los elementos de los arreglos directamente, porque los arreglos de apuntadores ya están en su lugar.
- Suponga que los apuntadores "p", "q" y "r" apuntan a los arreglos "a", "b" y "c", respectivamente. Las siguientes líneas muestran referencias equivalentes de arreglos y apuntadores para arreglos unidimensionales, bidimensionales y tridimensionales.

a[IMAX]	a[i]	*p	p[i]
b[IMAX][JMAX]	b[i][j]	**q	q[i][j]
c[IMAX][JMAX][KMAX]	c[i][j][k]	***r	r[i][j][k]

Las referencias a arreglos pueden usar apuntadores en lugar de nombres de arreglos. Los apuntadores usan un nivel de direccionamiento indirecto para cada dimensión de matriz. Los arreglos y los apuntadores pueden apuntar a cualquier tipo de datos.

2.6. Llamado con Retorno y Contexto en Pila: Funciones en C

2.6.1. Parámetros y argumentos
2.6.2. Paso de parámetros por valor y por referencia
2.6.3. Tipos de almacenamiento ("auto", "static", "register", "extern")
2.6.4. Retorno de valores de la función
2.6.5. Uso del "*stack*" y el "*stack frame*"

Conceptos clave:

Tomados del resumen del capítulo 2, titulado en inglés "*The Run Time Environment*", de [Anderson & Anderson, 1988], pp. 107-108.

- Los programas C utilizan las áreas de programa de **código**, **pila**, **datos** y "***heap***" del entorno en tiempo de ejecución. Es posible que un programa no use los datos o las áreas del "***heap***".
- El **área de código** normalmente está protegida contra la escritura de un programa en ejecución. Los apuntadores a funciones son direcciones del **área de código**.
- El compilador usa **la pila** para direcciones de retorno de funciones, "***stack frame***" y almacenamiento intermedio.
- El compilador también usa **la pila** para variables automáticas y funciones recursivas.
- El **área de datos** contiene variables de programa que almacenan valores durante la vida de su programa.
- La **parte BSS del área de datos** contiene las variables estáticas y globales no inicializadas de un programa C.
- Todas las variables estáticas y globales en el **BSS** tienen valores cero antes de que se ejecute su programa.
- Las **cadenas constantes** suelen vivir en el **área de datos**, pero algunos compiladores pueden colocarlas en el **área de código**.
- El compilador generalmente hace copias separadas de una sola **cadena constante** en el **área de datos** si la define más de una vez. La implementación de **cadenas constantes** depende del compilador y de la máquina.
- El "***heap***" es la memoria que se controla desde las rutinas de la biblioteca C. La memoria del "***heap***" se conserva a través de llamadas a funciones.
- El "***heap***" y la **pila** a veces comparten la misma área de memoria; por lo tanto, una gran cantidad de espacio de **pila** puede disminuir la cantidad de espacio de **pila** disponible y viceversa.
- El **área de código** y el **áreas de datos** tienen un tamaño fijo cuando se ejecuta su programa. La **pila** y el "***heap***" son dinámicos.

CAPÍTULO 3. METODOLOGÍAS DE DISEÑO

3.1. Ingeniería de Software y de Hardware

Conceptos clave:

- Una de las métricas de complejidad comunmente usadas corresponde al número de líneas de código fuente producidas en un proyecto.
- Las categorías de proyectos de software según líneas de código [Frakes et al., 1991] son: Trivial (< 1K lines), pequeño (1K a 3K lines), medio (3K a 50Klines), grande (50K a 100K lines), muy grande (100K a 1M lines), gigante (> 1M lines).
- Las fases de diseño de software [Frakes et al., 1991] son:
  - Exploración del concepto y análisis de factibilidad.
  - Especificación de requerimientos.
  - Diseño.
  - Implementación.
  - Pruebas.
  - Mantenimiento.
- Los atributos de un software de calidad [Frakes et al., 1991] son: que sea correcto, eficiente, fácil de mantener, portable, legible, confiable, re-usable, robusto, fácil de probar, bien documentado.
- Generalmente en hardware, el diseño de detalle corresponde al que se hace a nivel de componentes, mientras que el diseño arquitectónico se desarrolla a nivel de bloques.
- Verificar es comprobar si el diseño cumple los requerimientos funcionales.
- Validar es comprobar si el diseño suple la necesidad de un usuario/cliente.
- Use "*Waterfall*" cuando conoce con certeza las variables del entorno, costos y el producto final. Use "*Agile*" si hay incertidumbre.

3.2. Elaboración y Lectura de Diagramas

3.2.1. Diagramas de bloques
3.2.2. Diagramas de tiempo
3.2.3. Diagramas esquemáticos
3.2.4. Caso de estudio: "La Arquitectura de un Computador"
3.2.5. Aplicación: "Uso de Manuales de Especificaciones"

Conceptos clave para la secciones 3.2.2 y 3.2.3:

Tomados del resumen del capítulo 2, titulado en inglés "*Hardware Fundamentals for the Software Engineer*", de [Simon, 1999], pp. 41-44.

- La mayoría de los componentes de semiconductores, disponibles en chips, se venden en paquetes de plástico o cerámica. Se conectan entre sí mediante soldadura a las placas del circuito impreso.
- Los ingenieros electrónicos dibujan **diagramas esquemáticos** para indicar qué partes se necesitan en cada circuito y cómo deben conectarse entre sí. A menudo se asignan nombres a las señales en los diagramas esquemáticos.
- Las señales digitales siempre están en uno de dos estados: alto y bajo. Se dice que una señal es activa en alto cuando la condición es verdadera. Algunas señales se activan cuando están en alto y otras, cuando están en bajo.
- Cada chip tiene una colección de pines que son entradas y una colección que son salidas. En la mayoría de los casos, cada señal maneja exactamente una salida, aunque se puede conectar a varias entradas.
- Las compuertas de semiconductores estándar realizan funciones booleanas NOT, AND, OR y XOR en sus entradas.
- Además de sus pines de entrada y salida, la mayoría de los chips tienen un pin para conectarse a *VCC* y un pin para conectarse a tierra. Estos pines proporcionan energía para hacer que funcione el chip.
- Los condensadores de desacople evitan caídas de tensión locales en un circuito.
- Una señal que no tiene una salida activa es una señal flotante.
- Los dispositivos de colector abierto pueden reducir sus salidas o dejarlas flotantes, pero no pueden aumentarlas. Puede conectar varias salidas de colector abierto a la misma señal; esa señal será baja si alguna salida se está volviendo baja.
- Los dispositivos de tres estados pueden aumentar o disminuir sus salidas o dejarlas flotar. Puede conectar varias salidas de tres estados a la misma señal, pero debe asegurarse de que solo una de las salidas esté manejando la señal en cualquier momento y que el resto deje que la señal flote.
- Un punto en un **diagrama esquemático** indica que las líneas que se cruzan representan señales que deben conectarse entre sí.

- Una sola salida puede controlar solo un número limitado de entradas. Demasiadas entradas conducen a una señal sobrecargada.
- Los diagramas de tiempo muestran la relación de tiempo entre los eventos de un circuito.
- Los diversos tiempos importantes para la mayoría de los chips son el tiempo de *hold*, el tiempo de *setup* y el tiempo de reloj a Q.
- Los flip-flops D son dispositivos de memoria de 1 bit.
- Los tipos de memoria más comunes son *RAM*, *ROM*, *PROM*, *EPROM*, *EEROM* y flash. Dado que cada uno tiene características únicas, los usará para diferentes cosas.

3.3. Sistemas de Computador Embebido

3.3.1. Definición y usos
3.3.2. Atributos de calidad
3.3.3. Microprocesadores y buses
3.3.4. Metodología de diseño en capas
3.3.5. Casos de estudio: "Análisis de Sistemas Embebidos"

Conceptos clave para la secciones 3.3.1 y 3.3.2:

Tomados del resumen del capítulo 1, titulado en inglés "*A First Look at Embedded Systems*", de [Simon, 1999], pp. 10-11.

- Un sistema embebido es cualquier computador oculto dentro de un producto que no sea un computador.
- Cuando se escribe software para un sistema embebido, se encuentran muchas dificultades adicionales a las que encontrará cuando se escribe software de aplicaciones:
  - *Rendimiento en términos del throughput:* es posible que el sistema necesite manejar muchos datos en un período breve.
  - *Tiempos de respuesta:* es posible que el sistema deba reaccionar rápidamente a los eventos.
  - *Facilidad para comprobar el sistema:* configurar equipos para comprobar y verificar el correcto funcionamiento del software embebido, puede llegar a ser difícil.
  - *Facilidad para depurar el sistema:* sin una pantalla o un teclado, descubrir qué está haciendo mal el software (aparte de no funcionar) es problemático.
  - *Confiabilidad:* los sistemas embebidos deben ser capaces de manejar cualquier situación sin intervención humana.
  - *Espacio de memoria:* la memoria está limitada en los sistemas embebidos y se debe hacer que el software y los datos encajen en cualquier memoria.
  - *Instalación del programa:* se necesitarán herramientas especiales para poder instalar el software en los sistemas embebidos.
  - *Consumo de energía:* los sistemas portátiles deben funcionar con batería y el software de estos sistemas debe ahorrar energía.
  - *Acaparamiento de procesador:* la computación que requiere grandes cantidades de tiempo de la *CPU* puede complicar el problema de tiempos de respuesta.
  - *Costo:* reducir el costo del hardware es una preocupación en muchos proyectos de sistemas embebidos. El software a menudo opera en hardware que apenas es adecuado para el trabajo.
- Los sistemas embebidos tienen un microprocesador y una memoria, y algunos tienen un puerto serial o una conexión a red, y generalmente no tienen teclados, pantallas ni unidades de disco.

Conceptos clave para la sección 3.3.3:

Tomados del resumen del capítulo 3, titulado en inglés "*Advanced Hardware Fundamentals*", de [Simon, 1999], pp. 78-79.

- Un microprocesador típico tiene por lo menos un conjunto de pines de dirección, un grupo de pines de datos, uno o más pines de reloj, un pin de lectura para lectura y un pin de escritura.
- Al conjunto de señales de datos, de direcciones y de control que se conectan entre el microprocesador, la *ROM* y la *RAM* se le denomina el **bus**.
- El ingeniero electrónico debe asegurarse de que se cumplan los requisitos de tiempos de atención de cada una de las partes conectadas al bus. Los estados de **espera** *wait* y **las líneas de espera** *wait lines* son mecanismos para lograr esto.
- Los **circuitos de acceso directo a la memoria** (del inglés *DMA* por *Direct Memory Access*) mueven los datos directamente desde los dispositivos de E/S a la memoria y viceversa sin la intervención del microprocesador.
- Cuando un dispositivo de E/S necesita la atención del microprocesador, fuerza su señal de **interrupción** para informarle al microprocesador.
- Un **receptor/transmisor asíncrono universal** (del inglés *UART* por *Universal Asynchronous Receiver/Transmitter*) convierte los datos entre un formato de ocho bits y el formato de un bit a la vez que se utiliza en los puertos seriales, como los puertos RS-232. El microprocesador controla las *UART* a través de una serie de registros.
- La forma más simple de **dispositivo de lógica programable** (del inglés *PLD* por *Programmable Logic Device*) es la **matriz de lógica programable** (del inglés *PAL* por *Programmable Array Logic*). Una *PAL* contiene una colección de compuertas; puede reorganizar las conexiones entre dichas compuertas mediante un lenguaje de programación particular y un **programador de *PAL***.
- Un **circuito integrado de aplicación específica** (del inglés *ASIC* por *Applicaction Specific Integrated Circuit*) es un componente especialmente construido para un producto determinado.
- Un **temporizador de vigilancia** (del inglés *watchdog timer*) restablece el microprocesador e inicia el software desde el principio si el software no lo reinicia periódicamente.
- Los microprocesadores modernos típicos destinados a sistemas embebidos tienen de manera integrada: temporizadores, *DMA*, pines de E/S de puertos paralelos, decodificación de direcciones y memorias *cache*.
- Además de hacer que los circuitos funcionen de manera apropiada, los ingenieros de hardware deben lidiar con problemas de costo, energía y calor.

3.4. Configuración de Periféricos

3.4.1. Puertos paralelos
3.4.2. Temporizadores
3.4.3. Generadores "*Pulse-Width Modulation*" (PWM)
3.4.4. Conversores "*Analog-to-Digital*" (ADC)
3.4.5. Puertos seriales

Conceptos clave:

- Nota: Los conceptos clave de esta sección dependen de las arquitecturas específicas con las que se trabajará en el curso y están en desarrollo.

3.5. Protocolos de Comunicación

3.5.1. Comunicaciones "*on-board*"
3.5.2. Comunicaciones entre dispositivos
3.5.3. Comunicaciones inalámbricas

Conceptos clave:

- Nota: Los conceptos clave de esta sección están en desarrollo y contienen los siguientes temas:
- Los tipos de comunicación "*on-board*" de la sección 3.5.1 son "I2C" y "SPI" para conectar periféricos conectados sobre la misma PCB.
- Los tipos de comunicación entre sistemas o dispositivos de la sección 3.5.2 debe cubrir "UART", junto con los estándares RS232, RS422 y RS485.
- Los tipos de comunicaciones inalámbricas de la sección 3.5.3 son "Bluetooth", "Zig-bee", entre otros.

CAPÍTULO 4. MÁQUINAS DE ESTADOS FINITOS

4.1. Diagramas de Estado

Conceptos clave:

- Un grafo se puede representar de manera inequívoca mediante una matriz. El calificativo "inequívoco" significa que todo grafo se puede representar mediante una matriz y que a partir de una representación matricial se puede dibujar el correspondiente grafo.
- Un grafo se puede representar matricialmene mediante una matriz de incidencia ó como una matriz de adyacencia.

4.2. Tipos de Máquinas de Estados Finitos

4.2.1. Definiciones formales
4.2.2. Taxonomía de autómatas según existencia de E/S
4.2.3. Taxonomía de máquinas traductoras
4.2.4. Taxonomía de máquinas según memoria
4.2.5. Taxonomía de máquinas según naturaleza determinística

Conceptos clave:

- Cuando se habla de una definición formal se hace referencia a una definición matemática. En este contexto, generalmente se puede definir una máquina de estados finitos mediante la teoría de conjuntos.
- De manera formal, un autómata aceptor se define como una quíntupla $\mathbb{M} = \{\mathbb{Q}, q_0, \Sigma, \delta, \mathbb{F}\}$, donde:
  - $\mathbb{Q}$ es un conjunto finito de estados.
  - $q_0 \in \mathbb{Q}$ es el estado inicial.
  - Σ es el alfabeto de entrada.
  - $\delta: \mathbb{Q} \times \Sigma \rightarrow \mathbb{Q}$ es la función de transición.
  - $\mathbb{F} \subseteq \mathbb{Q}$ es el conjunto de estados finales o de aceptación.
- De manera formal, una máquina de estados finitos de Moore se define como una sextupla $\mathbb{M} = \{\mathbb{Q}, q_0, \Sigma, \Delta, T, G\}$, donde:
  - $\mathbb{Q}$ es un conjunto finito de estados.
  - $q_0 \in \mathbb{Q}$ es el estado inicial.
  - Σ es el alfabeto de entrada.
  - Δ es el alfabeto de salida.
  - $T: \mathbb{Q} \times \Sigma \rightarrow \mathbb{Q}$ es la función de transición.
  - $G: \mathbb{Q} \rightarrow \Delta$ es la función de salida.
- De manera formal, una máquina de estados finitos de Mealy se define como una sextupla $\mathbb{M} = \{\mathbb{Q}, q_0, \Sigma, \Delta, T, G\}$, donde:
  - $\mathbb{Q}$ es un conjunto finito de estados.
  - $q_0 \in \mathbb{Q}$ es el estado inicial.
  - Σ es el alfabeto de entrada.
  - Δ es el alfabeto de salida.
  - $T: \mathbb{Q} \times \Sigma \rightarrow \mathbb{Q}$ es la función de transición.
  - $G: \mathbb{Q} \times \Sigma \rightarrow \Delta$ es la función de salida.
- De manera formal, un autómata "pushdown" se define como una 7-tupla $\mathbb{M} = \{\mathbb{Q}, q_0, \Sigma, r, Z, \delta, \mathbb{F}\}$, donde:
  - $\mathbb{Q}$ es un conjunto finito de estados.
  - $q_0 \in \mathbb{Q}$ es el estado inicial.
  - Σ es el alfabeto de entrada.
  - r es el alfabeto del "stack".
  - $\mathrm{Z} \in r$ es el símbolo top del "stack".

- $\delta: \mathbb{Q} \times \Sigma \rightarrow \mathbb{Q}$ es la función de transición.
- $\mathbb{F} \subseteq \mathbb{Q}$ es el conjunto de estados finales o de aceptación.

- El concepto de "autómata finito" se usa comunmente en la teoría de los lenguajes formales de Chomsky y su respectiva taxonomía para el diseño de compiladores, mientras que el concepto de "máquina de estados finitos" surgió como una forma de diseñar redes secuenciales en hardware. No obstante lo anterior, tienen su fundamento en los mismos principios.
- Una vez diseñada una "máquina de estados finitos" se puede implementar de manera opcional, ya sea en hardware o en software.
- Los autómatas finitos, según la existencia de entrada/salida, se clasifican en:
  - Aceptores: autómatas con entrada y sin salida.
  - Secuenciadores: autómatas con salida y sin entrada.
  - Traductores: autómatas con entrada y salida.
- Los autómatas (ó máquinas) traductoras (con entrada y salida) se clasifican en:
  - Máquinas de Moore: Su salida es función del estado actual.
  - Máquinas de Mealy: Su salida es función de la entrada y el estado actual.
- Generalmente las máquinas de Mealy requieren menos estados que las máquinas de Moore.
- Toda máquina de Moore se puede convertir en una de Mealy y viceversa.
- Según su capacidad de memoria las máquinas se clasifican en:
  - Redes combinacionales: Circuitos lógicos sin memoria.
  - Máquinas de estados finitos: La memoria se usa solo para el estado. La salida es función instantánea del "estado" ó de la "entrada y estado", según la máquina sea de Moore ó Mealy, respectivamente.
  - Máquinas "*push-down*" ó con pila: Además de la memoria para guardar el estado, usan memoria adicional con mecanismo de pila con modo de acceso tipo LIFO (*Last In, Firts Out*).
  - Máquinas de Turing: Máquinas de estado con pila que escriben y borran en una cinta infinita que se mueve hacia adelante y hacia atrás.
- Adicionamente, las máquinas pueden ser:
  - No determinísticas.
  - Determinísticas.
- Toda máquina no-determinística se puede convertir en una máquina determinística.
- Todo sistema digital se compone de unidad de control (o secuenciador) y una unidad de datos (con "*data path*") que transforma el flujo de datos de la entrada a la salida según secuencia en la unidad de control.
- Una CPU es un sistema digital.

4.3. Implementación de Máquinas de Estados Finitos

4.3.1. Implementaciones software vs. hardware
4.3.2. Implementación con "switch-case"
4.3.3. Implementación con "goto-label"
4.3.4. Implementación con declaraciones complejas

Conceptos clave:

- La forma más estructurada de implementar con secuencias de control de flujo una FSM es con un ciclo sin-fin con “while” para iterar sobre los diferentes estados con “switch-case” y en cada estado evaluando la ocurrencia de eventos con un árbol de decisión que usa secuencias de selección priorizada con “if-else” anidados.
- Otra alternativa no-estructurada es mediante el uso de saltos con “goto-label”. No existe beneficio mayor con esta opción.
- Una tercera alternativa “interesante” para implementar FSM es usando declaraciones complejas que implementan tablas con apuntadoras a funciones.

4.4. Máquinas de Estados Finitos Concurrentes

4.4.1. Preservación del contexto en tareas

4.4.2. Contexto privado con tipo de almacenamiento "static"

4.4.3. Contexto en estructuras externas pasadas por referencia

Conceptos clave:

- El contexto de una función corresponde a los parámetros de la función, el conjunto de todas las variables automáticas en la pila, así como la instrucción a ser ejecutada en el PC.
- En la implementación de una FSM por software, la variable que guarda el estado corresponde en un alto nivel de abstracción al "*Program Counter*" de la unidad de control. De aquí la importancia de retener el siguiente estado junto al grupo de variables usadas en la FSM para permitir la ejecución concurrente de múltiples FSM. La retención de las variables de la pila y el siguiente estado junto a la entrada actual determinan la acción o salida de la FSM en software.
- Diferentes FSM concurrentes se pueden implementar con tipos de datos "static".
- Instancias concurrentes de una misma FSM se implementan mejor con estructuras pasadas por referencia a la función que implementa la FSM, en donde la estructura guarda todas las variables de la instancia de la FSM junto al siguiente estado.

CAPÍTULO 5. INTRODUCCIÓN A RTOS

5.1. Interrupciones Hardware

5.1.1. Conceptos básicos sobre interrupciones
5.1.2. Problemas de datos compartidos
5.1.3. Latencia de interrupción

Conceptos clave:

Tomados del resumen del capítulo 4, titulado en inglés "*Interrupts*", de [Simon, 1999], pp. 111-112.

- Algunas características del **lenguaje de ensamble** son las siguientes:
  - Cada instrucción del **lenguaje de ensamble** se traduce en una instrucción del microprocesador. A diferencia del lenguaje C, en la que una sola instrucción corresponde a un grupo de instrucciones del microprocesador.
  - Las instrucciones del **lenguaje de ensamble** mueven datos de la memoria a los **registros** dentro del microprocesador. **Otras instrucciones indican las operaciones** que se realizarán en los registros, mientras que **otras instrucciones mueven los datos de los registros a la memoria**.
  - Los **lenguajes de ensamble** típicos tienen **instrucciones de salto** e **instrucciones de salto condicional**, **instrucciones de llamada y retorno** e **instrucciones para poner y quitar datos de una pila** en la memoria.
- Cuando un dispositivo de E/S le indica al microprocesador que necesita servicio forzando una señal adjunta a uno de los pines de **solicitud de interrupción** del microprocesador, el microprocesador suspende lo que esté haciendo y ejecuta la **rutina de interrupción** correspondiente antes de continuar con el **código de la tarea**.
- Las rutinas de interrupción deben **guardar el contexto** y **restaurar el contexto** nuevamente.
- Los microprocesadores permiten que su software deshabilite las interrupciones (excepto la **interrupción no enmascarable**) cuando el software tiene que realizar un procesamiento crítico.
- Cuando las rutinas de interrupción y el código de la tarea comparten datos, se debe asegurar de que no interfieran entre sí. El primer método consiste en deshabilitar las interrupciones mientras el código de la tarea usa los datos compartidos.
- Se denomina **sección crítica** a un conjunto de instrucciones que no deben interrumpirse si el sistema funciona correctamente. Un conjunto de instrucciones que no será interrumpido (porque, por ejemplo, las interrupciones están deshabilitadas) se dice que es **atómico**.
- No se debe suponer que cualquier sentencia en C es atómica.

- La palabra clave “volatile” advierte al compilador que una rutina de interrupción podría cambiar el valor de una variable para que el compilador no optimice su código de una manera que lo haga fallar.
- La **latencia de interrupción** es la cantidad de tiempo que tarda un sistema en responder a una interrupción. Varios factores contribuyen a esto. Para mantener la latencia de interrupción baja (y en general buenos tiempos de respuesta), debe:
  - Hacer que sus rutinas de interrupción sean cortas.
  - Deshabilitar las interrupciones solo por períodos cortos de tiempo.
- Aunque existen técnicas para evitar deshabilitar las interrupciones, estas técnicas son frágiles y solo deben usarse si es estrictamente necesario.

5.2. Arquitecturas de Software

5.2.1. "*Round-robin*"
5.2.2. "*Round-robin*" con interrupciones
5.2.3. "*Function-Queue-Scheduling*"
5.2.4. Sistema Operativo en Tiempo-Real (*RTOS*)

Conceptos clave:

Tomados del resumen del capítulo 5, titulado en inglés "*Survey of Software Architectures*", de [Simon, 1999], pp. 133-134.

- Los requisitos de tiempos de respuesta suelen motivar la elección de la arquitectura.
- Las características de las cuatro arquitecturas discutidas se muestran en la Tabla 1.
- En general, será mejor eligir una arquitectura más simple.
- Una ventaja de los sistemas operativos en tiempo real es que pueden ser comprados y, por lo tanto, puede resolver algunos de los problemas sin tener que escribir el código.
- Las arquitecturas híbridas pueden tener sentido para algunos sistemas.

Tabla 1. Características de varias arquitecturas de software

	Priorities Available	Worst Response Time for Task Code	Stability of Response When the Code Changes	Simplicity
Round-robin	None	Sum of all task code	Poor	Very simple
Round-robin with interrupts	Interrupt routines in priority order, then all task code at the same priority	Total of execution time for all task code (plus execution time for interrupt routines)	Good for interrupt routines; poor for task code	Must deal with data shared between interrupt routines and task code
Function-queue-scheduling	Interrupt routines in priority order, then task code in priority order	Execution time for the longest function (plus execution time for interrupt routines)	Relatively good	Must deal with shared data and must write function queue code
Real-time operating system	Interrupt routines in priority order, then task code in priority order	Zero (plus execution time for interrupt routines)	Very good	Most complex (although much of the complexity is inside the operating system itself)

5.3. Servicios de una Arquitectura RTOS

5.3.1. Tareas y estados de las tareas
5.3.2. Tareas y datos
5.3.3. Semáforos y datos compartidos

Conceptos clave:

Tomados del resumen del capítulo 6, titulado en inglés "*Introduction to Real-Time Operating Systems*", de [Simon, 1999], pp. 168-169.

- Un **sistema operativo en tiempo real** (**RTOS**) típico es más pequeño y ofrece menos servicios que un sistema operativo estándar, y está más estrechamente relacionado con la aplicación.
- Los RTOS están ampliamente disponibles para la venta y, en general, tiene sentido comprar uno en lugar de escribir uno usted mismo.
- La tarea es el componente principal del software escrito para un entorno RTOS.
- Cada tarea está siempre en uno de tres estados: en ejecución, lista y bloqueada. El planificador en el RTOS ejecuta la tarea preparada de mayor prioridad.
- Cada tarea tiene su pila; sin embargo, todas las tareas comparten otros datos del sistema. Por lo tanto, el problema de los datos compartidos puede reaparecer.
- Una función que funciona correctamente se ejecuta incluso si es llamada por más de una tarea es llamada una **función reentrante**.
- Los semáforos pueden resolver el problema de los datos compartidos. Dado que solo una tarea puede tomar un semáforo a la vez, los semáforos pueden evitar que los datos compartidos causen errores. Los semáforos tienen dos funciones asociadas: tomar y soltar.
- Las tareas pueden usar semáforos para activarse entre sí.
- Se puede introducir cualquier cantidad de errores o *bugs* con los semáforos. La inversión de prioridad del inglés ***priority inversion*** y el ***deadly embrace*** son dos de los más oscuros. Olvidarse de tomar o soltar un semáforo o usar uno incorrecto son de las formas más comunes de fuente de problemas.
- El **mutex**, el **semáforo binario** y el **semáforo de conteo** se encuentran entre las variantes de semáforo más comunes que ofrecen los RTOS.
- Tres métodos para proteger los datos compartidos son deshabilitar las interrupciones, tomar semáforos y deshabilitar los cambios de tareas.

5.4. Otros Servicios del Sistema Operativo

5.4.1. "*Message-Queues*", "*Mailboxes*" y "*Pipes*"
5.4.2. Temporización de funciones
5.4.3. Eventos
5.4.4. Manejo de memoria
5.4.5. Rutinas de interrupción en un RTOS

Conceptos clave:

Tomados del resumen del capítulo 7, titulado en inglés "*More Operating System Services*", de [Simon, 1999], pp. 206-207.

- Las tareas deben poder comunicarse entre sí para coordinar actividades y compartir datos. La mayoría de los RTOS ofrecen alguna combinación de servicios, como colas de mensajes, buzones y conductos, para este propósito. Las características específicas de estos servicios dependen de RTOS; debe leer el manual para saber qué ofrece el RTOS.
- Pasar un apuntador a un búfer de una tarea a otra a través de una cola (o una tubería o un buzón) es una forma común de pasar un bloque de datos.
- La mayoría de los RTOS mantienen un temporizador de latidos que interrumpe periódicamente, que se usa para todos los servicios de temporización de RTOS. El intervalo entre las interrupciones del temporizador de latidos se denomina tic del sistema. Los servicios de temporización RTOS más comunes son estos:
  - Una tarea puede bloquearse a sí misma durante un número específico de pasos del sistema.
  - Una tarea puede limitar cuántos tics del sistema esperará un semáforo, una cola, etc
  - El código puede decirle al RTOS que llame a una función específica después de un número específico de tics del sistema.
- Los eventos son indicadores de un bit con los que las tareas se señalan entre sí. Los eventos se pueden formar en grupos y una tarea puede esperar una combinación de eventos dentro de un grupo.
- Aunque muchos RTOS ofrecen las funciones estándar malloc y free, los ingenieros suelen evitarlas porque son relativamente lentas e impredecibles. Es más común usar la asignación de memoria basada en un conjunto de búferes de tamaño fijo.
- Las rutinas de interrupción en un RTOS deben cumplir dos reglas:
  - No deben llamar funciones RTOS que bloqueen.
  - No deben llamar a ninguna función RTOS a menos que el RTOS sepa que se está ejecutando una rutina de interrupción. Los RTOS utilizan varios mecanismos para saber que se está ejecutando una rutina de interrupción.

5.5. Uso de un Diseño Básico con RTOS

5.5.1. Encapsulamiento de semáforos y colas
5.5.2. Consideraciones de planeación para tiempo real duro
5.5.3. Ahorro de espacio de memoria
5.5.4. Ahorro de consumo de potencia

Conceptos clave:

Tomados del resumen del capítulo 8, titulado en inglés "*Basic Design Using a Real-Time Operating System*", de [Simon, 1999], pp. 259-260.

- El diseño de software de sistemas embebidos es tanto un arte como una ciencia.
- Debe saber qué tan rápido debe operar el sistema y qué tan crítico es cumplir con cada lapso de tiempo límite. Si los tiempos son absolutos, el sistema es de tiempo real duro; de lo contrario, es un sistema suave en tiempo real.
- Debe saber qué hardware tendrá y qué tan rápido es.
- Las preocupaciones generales del software por la estructura, la modularidad, la encapsulación y la mantenibilidad aún se aplican en el mundo del software embebido.
- En gran parte del software embebido, los eventos del mundo real provocan interrupciones, lo que indica que las tareas deben hacer el trabajo. Los sistemas no hacen nada sin interrupciones; las tareas pasan su tiempo bloqueadas a menos que los eventos del mundo real les den algo que hacer.
- Las rutinas de interrupción cortas son mejores ya que las rutinas de interrupción se adelantan a las tareas y son propensas a errores. Mueva el procesamiento a tareas y haga que las rutinas de interrupción señalen las tareas para todos los procesamientos excepto los más urgentes. Sin embargo, no exagere porque la señalización en sí lleva tiempo.
- Es mejor que utilice menos tareas cuando pueda. Más tareas tienden a significar tener más errores, gastar más tiempo del microprocesador en el RTOS y necesitar más espacio de memoria.
- El procesamiento que tiene diferentes prioridades debe ir a diferentes tareas.
- Suele ser una buena idea encapsular el hardware con una tarea.
- La mejor estructura de tareas es la que se bloquea en un solo lugar, esperando un mensaje que le indique qué hacer a continuación. Las tareas a menudo se estructuran como máquinas de estado.
- Por lo general, no es bueno crear y destruir tareas que el sistema está ejecutando. Crea todas las tareas al principio.
- Asegúrese de que necesita dividir el tiempo antes de habilitarlo.
- Restringir la lista de funciones de RTOS le permite reducir el tamaño del sistema; construir un caparazón alrededor del RTOS hace cumplir la restricción y hace que el código sea más portátil.

- Debe encapsular semáforos, colas, etc. en módulos individuales para que la interfaz entre módulos sea una llamada de función.
- Para garantizar que un sistema duro en tiempo real cumpla con sus tiempos, debe asegurarse de que las tareas tengan un tiempo de ejecución predecible en el peor de los casos.
- Una forma de ahorrar espacio de datos en un sistema embebido que utiliza un RTOS es hacer que las pilas de tareas sean tan grandes como sea necesario.
- Puede ahorrar espacio de código en un sistema configurando el RTOS correctamente, usando un número limitado de funciones de biblioteca de C y examinando la salida del compilador de C en busca de construcciones C que requieran mucho espacio de código. Como último recurso, puede escribir el código en lenguaje ensamblador en lugar de hacerlo en C.
- Los sistemas que controlan las baterías ahorran energía apagando parte o todo el sistema. Cada sistema es diferente en lo que puede hacer en este sentido.