UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

METODOLOGIA DE DISEÑO DE SISTEMAS DE PROCESAMIENTO DIGITAL DE SEÑALES

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A N :

RICARDO ARROYO MENDOZA
JULIO ENRIQUE OROZCO TORRENTERA

DIRECTOR DE TESIS: DR. ROGELIO ALCANTARA SILVA

MEXICO, D. F. 1996.

TESIS CON
FALLA DE ORIGEN

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Al Dr. Rogelio Alcántara Silva, por la amistad, la dedicación y la guía que permitió llevar a la conclusión este esfuerzo conjunto.

A Jorge León Martínez, jefe del Centro de Cómputo de la Facultad de Derecho, por el invaluable apoyo brindado y, ante todo, por su sincera amistad.

A Patricia Díaz Hernández y Daniel Torres Patlán, por el afecto de amigos y el apoyo de compañeros.

A la Universidad Nacional Autónoma de México y su Facultad de Ingeniería, porque la formación que nos han dado constituye nuestra más grande riqueza.

A mi madre María Elena Mendoza Camacho por su amor silencioso que día a día vela por mí y a gritos me pide ser hombre. Te amo y no hay más palabras para expresarlo.

A mi hermana Martha Angélica Arroyo por esa mezcla de amor, coraje y valor que me inspiran a ser cada vez mejor.

A la memoria de mi padre Jesús Arroyo Gómez porque con su ejemplo me enseño a distinguir lo bueno y malo de la vida. Algún día estaremos juntos para compartir la gloria.

A Daniel, Jose Ramón e Iván por compartir su amistad a través del tiempo.

A Julio Enrique Orozco por compartir sus conocimientos en la realización de nuestra tesis y por representar en mi vida un ejemplo de honorabilidad y amistad a seguir.

Ricardo Arroyo Mendoza.

A mis padres, Ernesto Orozco y Adelina Torrentera, porque este trabajo no es sino un fruto de las semillas de amor, esfuerzo y apoyo infinitos que han sembrado día con día.

A mi hermano Ernesto, por el ejemplo inagotable de excelencia.

A mis hermanas Gabriela y Pilar, por todo su cariño y aliento.

A toda mi familia y amigos.

A los toliros 19.

A Ricardo, porque la amistad ha sido el testigo y el medio vital de este proyecto.

Julio Enrique Orozco Torrentera,
mayo 1996.

Indice.

página:

Introducción **I-1**

1. Señales, Sistemas y Algoritmos de Procesamiento Digital de Señales **1-1**

INTRODUCCION. 1-1

1.1 GENERALIDADES DE LAS SEÑALES DISCRETAS. 1-2

1.1.1 Concepto de señal. 1-2

1.1.2 Señales en tiempo continuo, en tiempo discreto y señales digitales. 1-2

1.1.3 Digitalización de una señal continua. 1-3

1.1.4 Representación de señales discretas. 1-3

1.1.5 Cambio de la variable independiente (el tiempo). 1-5

1.1.6 Señales discretas básicas. 1-5

1.1.6.1 Impulso unitario (Delta de Kronecker). 1-5

1.1.6.2 Escalón unitario. 1-6

1.1.6.3 Rampa unitaria. 1-7

1.1.6.4 Exponencial compleja discreta. 1-7

1.1.6.5 Propiedades de periodicidad de señales exponenciales complejas discretas. 1-8

1.1.6.6 Exponenciales complejas relacionadas armónicamente. 1-9

1.2 SISTEMAS DISCRETOS. 1-9

1.2.1 Generalidades. 1-9

1.2.2 Representación de sistemas en diagramas de bloques. 1-10

1.2.2.1 Sumador. 1-11

1.2.2.2 Multiplicación por un escalar. 1-11

1.2.2.3 Multiplicación de señales. 1-11

1.2.2.4 Elemento de retraso unitario. 1-12

1.2.2.5 Elemento de adelanto unitario. 1-12

1.2.3 Clasificación de los sistemas discretos según sus propiedades. 1-12

1.2.3.1 Estático y dinámico. 1-12

1.2.3.2 Variante e invariante con el tiempo. 1-13

1.2.3.3 Lineales. 1-13

1.2.3.4 Causales. 1-13
1.2.3.5 Estables e Inestables. 1-13
1.3 ANALISIS DE SISTEMAS DISCRETOS LINEALES E INVARIANTES CON EL TIEMPO. 1-14
1.2.1 Análisis de sistemas lineales en el dominio del tiempo. 1-14
1.2.1.1 Relación entrada-salida o ecuación en diferencias. 1-14
1.2.1.2 Descomposición de la señal de entrada en una suma de señales elementales. 1-15
1.2.1.3 Respuesta de los Sistemas Lineales e Invariantes con el Tiempo a una entrada arbitraria: Sumatoria de Convolución. 1-16
1.2.2 Análisis en el dominio de la frecuencia. 1-16
1.2.2.1 Espectro. 1-16
1.2.2.2 Transformadas. 1-17
1.2.3 Transformada Z. 1-17
1.2.3.1 Definición, Región de Convergencia y Transformada inversa. 1-17
1.2.3.2 Función de Transferencia. 1-18
1.4 ALGORITMOS DE PROCESAMIENTO DIGITAL DE SEÑALES. 1-19
1.4.1 Correlación. 1-19
1.4.3 Transformada Discreta de Fourier. 1-20
1.4.4 Transformada Rápida de Fourier. 1-21
1.4.4.1 Decimación en el tiempo, Rádix-2. 1-22
1.4.4.2 Decimación en la frecuencia. 1-23
1.4.5 Filtros digitales. 1-24
1.4.5.1 Filtros IIR. 1-25
1.4.5.2 Filtros FIR. 1-27
CONCLUSIONES. 1-29
REFERENCIAS. 1-30

2. Metodologías de Desarrollo de Software. **2-1**
INTRODUCCION. 2-1
2.1 METODOLOGIA CLASICA. 2-2
2.2 METODOLOGIA SEMIESTRUCTURADA. 2-3
2.3 METODOLOGIA ESTRUCTURADA. 2-5
2.4 METODOLOGIA DE DISEÑO ORIENTADA A OBJETOS. 2-7
2.5 PROTOTYPING. 2-9
2.6 SISTEMAS EN TIEMPO REAL. 2-11
2.6.1 Manejo de interrupciones. 2-13

2.6.2 Sistema operativo de tiempo real. 2-14
2.6.3 Lenguajes en tiempo real. 2-14
2.6.4 Análisis, modelado y simulación de sistemas en tiempo real. 2-15
CONCLUSIONES. 2-17
REFERENCIAS. 2-18

3. Herramientas de Desarrollo de Sistemas DSP 3-1
INTRODUCCION. 3-1
3.1 CLASIFICACION DE HERRAMIENTAS. 3-1
3.1.1 Herramientas de Diseño de Circuitos Integrados VLSI. 3-1
3.1.2 Herramientas de Simulación de Sistemas DSP. 3-2
3.1.3 Herramientas para Sistemas Heterogéneos y Desarrollo de Prototipos. 3-2
3.2 DESCRIPCION DE HERRAMIENTAS DE DISEÑO DE CIRCUITOS INTEGRADOS VLSI. 3-2
3.2.1 Generalidades de las Herramientas. 3-2
3.2.2 Pyramid. 3-3
3.2.3 VHSIC Silicon Compiler (VSC). 3-4
3.3 DESCRIPCION DE HERRAMIENTAS DE SIMULACION DE SISTEMAS DSP. 3-5
3.3.1 Khoros. 3-5
3.3.2 Mathematica. 3-8
3.4 DESCRIPCION DE HERRAMIENTAS DE SISTEMAS HETEROGENEOS Y DESARROLLO DE PROTOTIPOS. ... 3-9
3.4.1 Ptolemy. 3-9
3.4.2 Matlab. 3-11
CONCLUSIONES. 3-14
REFERENCIAS. 3-15

4. Propuesta de una Metodología de Diseño de Sistemas DSP. 4-1
INTRODUCCION. 4-1
4.1 PROPUESTA DE LA METODOLOGIA. 4-1
4.2 PLANTEAMIENTO DEL PROBLEMA. 4-3
4.2.1 Objetivo. 4-3

4.2.2 Entradas. 4-3
4.2.3 Salidas. 4-4
4.2.4 Actividades. 4-4
4.2.4.1 Detección del problema. 4-4
4.2.4.2 Identificación de requerimientos iniciales. 4-5
4.2.4.3 Redacción de un documento de solicitud de implantación de la solución a un problema. 4-5
4.3 ANALISIS DEL PROBLEMA. 4-5
4.3.1 Objetivo. 4-5
4.3.2 Entradas. 4-5
4.3.3 Salidas. 4-6
4.3.4 Actividades. 4-6
4.3.4.1 Investigación. 4-6
4.3.4.2 Definición de Requerimientos. 4-7
4.3.4.3 Redacción de la especificación. 4-9
4.4 DISEÑO DE LA SOLUCIÓN Y CONSTRUCCION DEL PROTOTIPO. 4-11
4.4.1 Objetivo. 4-11
4.4.2 Entradas. 4-12
4.4.3 Salidas. 4-12
4.4.4 Actividades. 4-12
4.4.4.1 Diseño de la Solución Algorítmica. 4-12
4.4.4.2 Diseño de la Solución Software. 4-14
4.4.4.3 Diseño de la Solución Microcódigo. 4-15
4.4.4.4 Diseño de la Solución Hardware. 4-16
4.4.4.5 Presentación del prototipo de solución. 4-17
4.5 PRUEBA Y REFINAMIENTO DEL PROTOTIPO. 4-17
4.5.1 Objetivo. 4-17
4.5.2 Entradas. 4-18
4.5.3 Salidas. 4-18
4.5.4 Actividades. 4-18
4.5.4.1 Confirmación. 4-18
4.5.4.2 Verificación. 4-18
4.6 OPERACION. 4-19
4.6.1 Objetivo. 4-19
4.6.2 Entradas. 4-19
4.6.3 Salidas. 4-19

4.6.4 Actividades. 4-20

4.6.4.1 Redacción de la documentación de usuario. 4-20

4.6.4.2 Redacción de la documentación del sistema. 4-20

4.6.4.3 Mantenimiento. 4-21

CONCLUSIONES. 4-22

REFERENCIAS. 4-23

5. Principios de Audio Digital. **5-1**

INTRODUCCION. 5-1

5.1 CARACTERISTICAS DE LAS ONDAS DE SONIDO. 5-2

5.1.1 Fenómenos del sonido. 5-3

5.1.2 Medición del sonido. 5-4

5.1.3 Tono Musical. 5-7

5.1.4 Calidad musical (timbre). 5-8

5.1.5 Intensidad del sonido y su efecto en la distancia. 5-8

5.1.6 Formas de onda complejas. 5-9

5.2 MUESTREO DISCRETO EN EL TIEMPO. 5-10

5.3 EL FENOMENO DEL ALIASING. 5-12

5.3.1 Solución al Fenómeno del Aliasing. 5-13

5.4 CUANTIZACION. 5-13

5.4.1 Razón Señal a Error. 5-14

5.4.2 Distorsión en la Cuantización. 5-16

CONCLUSIONES. 5-18

REFERENCIAS. 5-19

6. Diseño de un Sistema de Producción de Efectos Especiales en Señales de Audio. **6-1**

INTRODUCCION. 6-1

6.1 PLANTEAMIENTO DEL PROBLEMA. 6-1

6.2 ANALISIS DE LA SOLUCION. 6-2

6.2.1 Antecedentes Históricos. 6-2

6.2.2 Tipos de Efectos Especiales para Guitarra Eléctrica 6-2

6.2.3 Requerimientos del Sistema 6-3

6.2.3.1 Señales de entrada y salida. 6-3

6.2.3.2 Opciones de procesamiento. 6-4
6.2.3.3 Modos de procesamiento. 6-4
6.2.3.4 Alcance. 6-4
6.2.4 Especificación de la Solución. 6-4
6.2.4.1 Distorsión. 6-4
6.2.4.2 Trémolo. 6-5
6.2.4.3 Efectos de Desplazamiento en el Tiempo (Time Shifting). 6-5
6.3 DISEÑO DE LA SOLUCION ALGORITMICA. 6-8
6.3.1 Distorsión. 6-8
6.3.2 Trémolo. 6-10
6.3.3 Delay. 6-11
6.3.4 Eco. 6-12
6.3.5 Reverb. 6-13
6.3.6 Flanger/Chorus. 6-15
6.4 DISEÑO DE LA SOLUCION SOFTWARE. 6-16
6.5 DISEÑO DE LA SOLUCION MICROCODIGO. 6-19
6.6 PRUEBA Y REFINAMIENTO DEL PROTOTIPO. 6-20
6.6 OPERACION. 6-21
CONCLUSIONES. 6-22
REFERENCIAS. 6-23

Conclusiones y Perspectivas. **Con-1**

APENDICE A: Guía de Utilización de Ptolemy. **A-1**
INTRODUCCION. A-1
A.1 INFORMACIÓN SOBRE ADQUISICIÓN DEL SOFTWARE E INSTALACION. A-1
A.2 CONCEPTOS BASICOS. A-2
A.3 LA INTERFAZ GRÁFICA DE USUARIO Y LOS DEMOS DE PTOLEMY. A-2
A.3.1 Inicio. A-2
A.3.2 Ejecución y cambio de parámetros. A-4
A.4 CREACIÓN DE UNIVERSOS. A-6
A.4.1 Inicio. A-6
A.4.2 Ejemplo de creación de universos. A-7

REFERENCIAS. A-13

APENDICE B: Programas Fuente Generados por Ptolemy. B-1

INTRODUCCION. B-1

B.1 CODIGO GENERADO PARA EL TREMOLO. B-4

B.2 CODIGO GENERADO PARA EL DELAY. B-6

B.3 CODIGO GENERADO PARA EL ECO. B-8

B.4 CODIGO GENERADO PARA EL RETRASO REALIMENTADO. B-10

B.5 CODIGO GENERADO PARA EL FILTRO PASO-TODO. B-12

APENDICE C: Programa de Producción de Efectos Especiales en Señales de Audio Digital. C-1

INTRODUCCION. C-1

C.1 CLASE "EFECTO". C-1

C.2 CLASE "EFECTORETRASO". C-2

C.3 CLASE "EFECTORETRASOPEQ". C-3

C.4 CLASE "DELAY". C-4

C.5 CLASE "ECO". C-5

C.6 CLASE "FLANGER". C-6

C.7 CLASE "DISTORSION". C-7

C.8 CLASE "TREMOLO". C-8

C.9 CLASE "DELAYFB". C-9

C.10 CLASE "ALLPASS". C-9

C.11 CLASE "REVERB". C-10

C.12 EL FORMATO .VOC. C-12

C.13 CODIGO COMPLETO DEL PROGRAMA DE PRODUCCION DE EFECTOS ESPECIALES EN SEÑALES DE AUDIO. C-13

REFERENCIAS. C-26

APENDICE D: Paleta de Efectos de Audio en Ptolemy. D-1

INTRODUCCION. D-1

D.1 ECO. D-2

D.2 DELAY. D-2
D.3 TREMOLO. D-3
D.4 DISTORSION. D-3
D.5 RETRASO REALIMENTADO (DELAYFB). D-4
D.6 FILTRO PASO TODO (ALLPASS). D-5
D.7 REVERB. D-5

APENDICE E: Microcódigo Generado por Ptolemy. E-1
INTRODUCCION. E-1
E.1 DELAY. E-2
E.2 ECO. E-3
E.3 TREMOLO. E-5
E.4 REVERB. E-7

Glosario. G-1

Referencias Bibliográficas y Hemerográficas. R-1

Introducción.

El acelerado incremento en la capacidad y la velocidad de los sistemas de procesamiento de información ha permitido el surgimiento y desarrollo de disciplinas como el Procesamiento Digital de Señales (DSP por sus siglas en inglés: Digital Signal Processing), rama de la ingeniería cuyas aplicaciones incluyen el audio digital, las comunicaciones, el procesamiento de imágenes, el procesamiento de voz, la prospección sísmica, el procesamiento en señales de radar, etc.

La creciente aplicación del Procesamiento Digital de Señales en áreas antes exclusivas de los sistemas analógicos se ha dado en una forma íntimamente ligada al diseño de las arquitecturas de cálculo y ha carecido de un enfoque sistemático de diseño; esto quiere decir, que cada aplicación se ha desarrollado en una forma particular y aislada, sin seguir un proceso que permita la creación de sistemas más eficientes, económicos y robustos.

En la práctica, la creación de un sistema de Procesamiento Digital de Señales implica poner en práctica los conocimientos sobre:

- principios teóricos del análisis de sistemas y señales
- algoritmos de DSP
- diseño e implantación de software de alto nivel
- diseño e implantación de microcódigo
- diseño e implantación de arquitecturas de procesamiento

Estos conocimientos, sin embargo, no se encuentran estructurados en una secuencia de pasos que permita desarrollar un sistema DSP en forma ordenada y sistemática.

Ante tal carencia, el objetivo general de este trabajo consiste en proponer una metodología de diseño de sistemas de Procesamiento Digital de Señales. Para lo cual hemos planteado alcanzar los siguientes objetivos específicos:

- conocer los principios teóricos básicos de la disciplina
- investigar las características generales de las metodologías existentes en el desarrollo de sistemas de software y de arquitecturas de procesamiento.
- investigar y conocer la utilización de herramientas de software que auxilian al diseño de sistemas DSP en sus distintas fases

Las actividades desempeñadas en la consecución de los objetivos anteriores, nos proporcionan las bases necesarias para la propuesta de una metodología, cuya validez probaremos aplicándola al diseño de un sistema de producción de efectos especiales en señales de audio. Esto implica dos objetivos específicos adicionales:

- investigar los principios del audio digital
- diseño e implantación del sistema de producción de efectos especiales en señales de audio

El presente documento consta de 6 capítulos donde se desarrollan los temas correspondientes a los objetivos planteados.

En el capítulo 1 se presenta una revisión de los fundamentos del análisis de señales y sistemas discretos, además de los algoritmos básicos del DSP.

En el capítulo 2 se resumen las características más importantes de algunas de las principales metodologías que se han desarrollado en la ingeniería de software. Las metodologías tratadas son: Clásica, Semiestructurada, Estructurada, Orientada a Objetos y el Prototyping (desarrollo de prototipos). Se aborda también, por su estrecha relación con los sistemas DSP, el diseño de los sistemas en tiempo real como objeto de aplicación de las distintas metodologías.

Las herramientas de desarrollo de sistemas DSP son el tema del capítulo 3. En él, se propone una clasificación general de dichas herramientas de software según su función: a) diseño de circuitos integrados; b) simulación y c) sistemas heterogéneos y desarrollo de prototipos. Dentro de cada clasificación, se detallan las características de algunos productos de software disponibles.

A lo largo del capítulo 4 se desarrolla la propuesta de la metodología, la cual consta de las siguientes fases:

- Planteamiento del problema
- Análisis del problema
- Diseño de la solución y construcción del prototipo
- Prueba y refinamiento del prototipo
- Operación

Para cada fase se detallan el objetivo y las actividades implicadas en cada una, además de desarrollar una narrativa del proceso y de la importancia del uso de las herramientas de software a lo largo del mismo.

Es muy importante mencionar que la metodología propuesta tiene como objeto el diseño y realización de sistemas de procesamiento digital de señales, entendiendo que un sistema de esta naturaleza puede estar constituido por un algoritmo, un programa de alto nivel, un programa en lenguaje ensamblador, una arquitectura de procesamiento o un nuevo circuito integrado, y que puede ser parte de un sistema más amplio y complejo, como un producto terminado de hardware o de software. Este tipo de sistemas incluyen muchas partes cuyo desarrollo escapa al alcance de la metodología. Por ejemplo, en el caso de un teléfono celular que incluya un sistema de procesamiento digital de señales, el diseño ergonómico del gabinete y las teclas, o la

mercadotecnia de su comercialización, que son partes igualmente importantes en el éxito de un producto terminado, serán realizadas por las metodologías y procesos característicos de esas actividades. En el caso de un sistema completo de software, como una consola de producción de audio en un estudio de grabación, la interfaz gráfica y el manejo de bases de datos, por mencionar dos ejemplos, deberán ser desarrolladas siguiendo una metodología específica de ingeniería de software (como la estructurada y la orientada a objetos). En ambos casos, se requiere de un sistema (o subsistema) que se encargue del procesamiento digital de señales, cuyo desarrollo es el objetivo de la metodología aquí propuesta. La utilización de esta metodología, facilitará la integración e interacción de este subsistema en el sistema global. De esta forma, algunas características del sistema general podrán implicar la definición de ciertos requerimientos de la solución DSP, el cual por su parte tendrá un impacto en las especificaciones de otros módulos del sistema global. Asimismo, la aplicación de esta metodología puede extenderse, con las adaptaciones pertinentes, a campos como el control de procesos, las comunicaciones, la robótica, el reconocimiento de patrones, etc.

Con el objetivo de validar, reforzar y ejemplificar la metodología propuesta se realiza el diseño de la parte de DSP de un sistema de producción de efectos especiales en señales de audio. Este sistema aplica conceptos de un área de nuestro interés: el Audio Digital, cuyos principios (cuantización, teorema de muestreo, características de las ondas de sonido y el fenómeno del aliasing, entre otros) se resumen en el capítulo 5.

A partir de dichos principios, en el capítulo 6 se presenta la semblanza de la aplicación de la metodología propuesta en el diseño del sistema que produce diversos efectos especiales como retraso (delay), eco, reverberación (reverb) y trémolo, dándose soluciones a los diferentes niveles establecidos por la metodología.

1

Señales, Sistemas y Algoritmos de Procesamiento Digital de Señales.

INTRODUCCION.

Los seres vivos, incluído el hombre, interactuamos con nuestro medio ambiente a través del intercambio de grandes cúmulos de información. Así, reconocemos, clasificamos, cuantificamos, modificamos, almacenamos, comprimimos, agrupamos, predecimos y transmitimos información para entender y transformar nuestro entorno en alguna forma. Esa información es percibida mediante imágenes, sonidos, texturas, colores, etc., lo que de forma general llamamos *señales*, mientras que a las distintas tareas arriba mencionadas las englobamos en el término *Procesamiento de señales* y a aquél conjunto de elementos que define las relaciones entre señales lo denominamos *sistema,* [ALC88].

El surgimiento y desarrollo de las computadoras electrónicas digitales ha significado que muchas áreas del conocimiento humano hayan cambiado la forma en que tradicionalmente se desempeñaban y hayan surgido otras como la computación, la informática y el Procesamiento Digital de Señales, que es una rama de la ingeniería que plantea, analiza y resuelve problemas del mundo real mediante algoritmos y modelos definidos por *sistemas en tiempo discreto.*

Los elementos del Procesamiento Digital de Señales (DSP por sus siglas en inglés: Digital Signal Processing) son el análisis en el tiempo y en la frecuencia (espectro) de sistemas discretos, el diseño de filtros digitales, la modelización y estimación paramétrica de sistemas, etc. Todos estos elementos intervienen en aquellas aplicaciones que requieran una gran intensidad de cálculo, como los sistemas en tiempo real. Las crecientes capacidades de cálculo y de rendimiento de los circuitos integrados digitales han hecho que las aplicaciones DSP sean cada vez más extensas, más importantes, y estén desplazando a los circuitos analógicos en muchas de las funciones en que éstos habían sido imprescindibles.

Este capítulo consta de cuatro partes en las que presentaremos las generalidades de

las señales y los sistemas en el tiempo discreto, y desarrollaremos los principios fundamentales del Procesamiento Digital de Señales. En la primera parte hablaremos de las señales, de su naturaleza continua y de su discretización y cuantización, definiremos las principales señales discretas y sus propiedades y operaciones. En la segunda parte definiremos a los sistemas discretos, sus características y su clasificación. La tercera parte presenta el análisis de sistemas en los dominios del tiempo y la frecuencia y, finalmente, en la cuarta parte desarrollaremos los principios y algoritmos básicos del Procesamiento Digital de Señales.

1.1 GENERALIDADES DE LAS SEÑALES DISCRETAS.

1.1.1 Concepto de señal.

Según mencionamos en la introducción, una señal implica el transporte de información; es una medida o magnitud física que describe un fenómeno a través de la información que transporta. Matemáticamente, se concibe a las señales como funciones de una variable independiente y se les representa con letras, por ejemplo $x(t)$, donde x es la función de la variable independiente t, [CAD85].

1.1.2 Señales en tiempo continuo, en tiempo discreto y señales digitales.

La información es intrínsecamente cambiante, y la magnitud física que implica el cambio por excelencia es el tiempo, de forma que la variable independiente de gran parte de las señales será el tiempo. Ahora bien, podemos tener señales que estén definidas como funciones para cualquier valor del tiempo, en cuyo caso se denominan señales en tiempo continuo o señales continuas, mientras las señales definidas sólo en valores específicos del tiempo se llaman señales en tiempo discreto o señales discretas. Un ejemplo de señal continua es la temperatura de una habitación, que puede variar en cualquier instante de tiempo. Un ejemplo de señal en tiempo discreto es una cuenta bancaria a plazo fijo, en donde el saldo sólo varía en ciertos instantes definidos de tiempo, que en este caso son los plazos de vencimiento.

La variable dependiente de una señal también puede tomar valores continuos o discretos, en este caso a las señales se les denomina señales cuantizadas. La mayoría de las veces, los valores que pueden tener estas últimas señales son equidistantes y pueden ser expresados como múltiplos de un valor específico. Cuando una señal está definida en el tiempo discreto y además es cuantizada, se dice que es una señal digital. Este será el tipo de señales de mayor interés para este trabajo, puesto que el Procesamiento Digital de Señales se aplica precisamente sobre ellas.

Otro concepto importante es el de señales multicanal y señales multidimensionales.

Las señales multicanal son aquéllas que provienen de un conjunto de fuentes, por ejemplo, un arreglo de sensores colocado en el cráneo para monitorear la actividad cerebral, donde todas las señales son funciones de la misma variable independiente, [PRO88]. Las señales multidimensionales son aquéllas que son funciones de más de una variable independiente, como una imagen de video, donde la luminosidad (para TV en blanco y negro) o el color (para TV a color) de un punto en la pantalla depende de tres variables: la posición horizontal, la posición vertical y el tiempo.

1.1.3 Digitalización de una señal continua.

Muchas aplicaciones se basan en señales que son de naturaleza continua, pero si han de utilizarse las amplias capacidades de procesamiento de las computadoras digitales, estas señales deben convertirse a una forma compatible con los datos que pueden manejarse en las computadoras, es decir, deben transformarse en secuencias de valores numéricos o señales digitales. A esta transformación se le conoce como conversión Analógico-Digital (en inglés: Analog-to-Digital conversion) e implica dos operaciones: discretización y cuantización, [PRO88].

La primera consiste en tomar muestras finitas de la señal continua para convertirla en una señal en el tiempo discreto. Puede conceptualizarse como un interruptor que se abre y se cierra en instantes de tiempo t_n, tomando cada vez que se cierra una muestra $x(t_n)$ de la señal continua $x(t)$. Los instantes de tiempo son equidistantes a un período T que recibe el nombre de período de muestreo, y están definidos por:

$$t_n = nT \quad \text{donde } n = ..., -2, -1, 0, 1, 2, ... \tag{1.1}$$

La cuantización consiste en convertir los valores de las muestras a valores definidos en un conjunto finito. Esto necesariamente implica una aproximación, pues el valor de cada una de las muestras se sustituirá por uno de los valores definidos en el conjunto de valores posibles. Esta sustitución puede ser de diversa índole: redondeo, truncamiento, etc.

Una vez completadas estas dos operaciones, tendremos una señal de naturaleza discreta tanto en el tiempo como en sus valores, es decir, una señal digital.

1.1.4 Representación de señales discretas.

Las formas de representar una señal discreta son [CAD85]:

1.- Mediante una fórmula matemática (expresión cerrada).

2.- Mediante una lista explícita de los elementos de la secuencia (forma tabular):

$$x = \{\ldots, x(-2), x(-1), \underset{\uparrow}{x(0)}, x(1), x(2), \ldots\} \tag{1.2}$$

donde la flecha indica el elemento x(0).

3.- Puede representarse una secuencia gráficamente.

Para ejemplificar las distintas formas de representación tenemos en primer lugar, en expresión cerrada, la secuencia:

$$x(n) = \begin{cases} \left(\frac{1}{2}\right)^n & \text{para } 0 \le n \le 4 \\ 0 & \text{en cualquier otro caso} \end{cases} \tag{1.3}$$

donde n puede tomar sólo valores enteros en el intervalo. Enseguida, tenemos la forma tabular:

$$x(n) = \{\ldots, 0, 0, 0, \underset{\uparrow}{1}, \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{8}, \tfrac{1}{16}, 0, 0, 0, \ldots\} \tag{1.4}$$

En esta representación, se hace evidente la necesidad de la flecha que indica el valor de la secuencia para $n=0$. Finalmente, tenemos la representación gráfica de la secuencia (fig. 1.1).

La forma de representación de una secuencia depende mucho del caso particular, y en ocasiones, los casos no permiten la representación en forma cerrada, pero éstos son los menos.

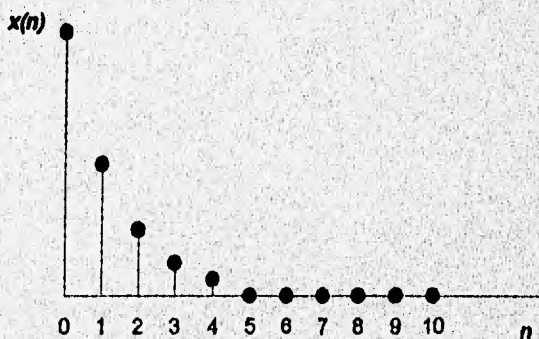


Fig. 1.1

1.1.5 Cambio de la variable independiente (el tiempo).

- Desplazamiento. Se dice que una secuencia *{x(n)}* se desplaza *k* unidades en el tiempo sustituyendo en el argumento de la secuencia a n por *n-k*. Por lo tanto, la secuencia desplazada de *{x(n)}* en *k* unidades es *{x(n-k)}*, y está definida por:

$$x(n-k)=\{\ldots,x(-2-k),x(-1-k),\underset{\uparrow}{x(-k)},x(1-k),x(2-k),\ldots\} \qquad (1.5)$$

donde k es un entero fijo.

- Transposición. La transpuesta de una secuencia se obtiene cambiando la variable *n* por *-n* . Por lo tanto, la transpuesta *x(-n)* es:

$$x(-n)=\{\ldots,x(2),x(1),\underset{\uparrow}{x(0)},x(1),x(2),\ldots\} \qquad (1.6)$$

Las gráficas de las secuencias *x(n)* y *x(-n)* son espejos una de la otra, teniendo como referencia el origen. Cuando *x(-n)=-x(n)* decimos que la señal es impar o antisimétrica, como es el caso de la función seno, mientras que cuando *x(-n)=x(n)*, la señal es par o simétrica, como el coseno.

- Escalamiento. El escalamiento de la variable de tiempo se da sustituyendo a la variable original por un factor escalar de la misma, por ejemplo, si tenemos la señal *x(n)*, podemos tener las señales *x(2n)* y $x(\frac{n}{2})$, [OPP90].

1.1.6 Señales discretas básicas.

Existe una serie de señales discretas básicas muy útiles en el análisis y diseño de sistemas discretos, como el caso de los sistemas de procesamiento digital de señales. Estas señales son:

1.1.6.1 Impulso unitario (Delta de Kronecker).

El impulso unitario (fig. 1.2), a veces también llamado muestra unitaria, está definido por:

$$\delta(n)=\begin{cases}0 & n\neq 0\\ 1 & n=0\end{cases} \qquad (1.7)$$

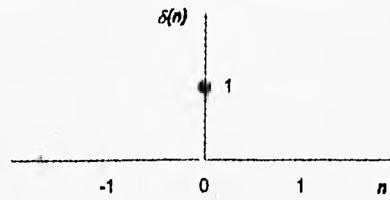


Fig. 1.2

Cabe hacer notar aquí la mayor simplicidad de el impulso discreto comparado con el impulso continuo, que para *t=0* tiene valor infinito y área unitaria, lo que implica una complejidad matemática inherente en su definición, [OPP90].

1.1.6.2 Escalón unitario.

El escalón discreto (fig. 1.3) se define en la siguiente expresión:

$$u(n) = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases} \qquad (1.8)$$

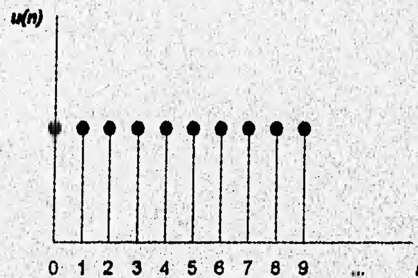


Fig. 1.3

Las relaciones entre estas dos señales discretas son las siguientes:

El impulso unitario es la primera diferencia del escalón unitario. Esta primera diferencia equivaldría a la primera derivada en el dominio continuo, y está definida por:

$$\delta(n)=u(n)-u(n-1) \qquad (1.9)$$

de igual forma, la relación inversa a la primera diferencia es la sumatoria, que **equivaldría a la integral.** Para el caso de nuestras señales, el escalón es la suma del impulso unitario. Esto es [OPP90]:

$$u(n)=\sum_{-\infty}^{\infty}\delta(m)=\sum_{k=0}^{\infty}\delta(n-k) \tag{1.10}$$

1.1.6.3 Rampa unitaria.

Esta secuencia (fig. 1.4) se denota con la expresión:

$$u_r(n)=\begin{cases}0 & n<0\\ n & n\geq 0\end{cases} \tag{1.11}$$

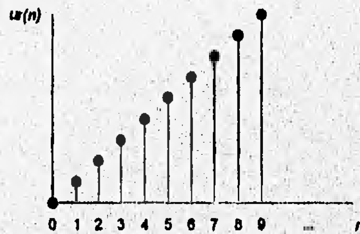


Fig. 1.4

El escalón unitario es la primera diferencia de la rampa, esto es [OPP90]:

$$u(n)=u_r(n)-u_r(n-1) \tag{1.12}$$

1.1.6.4 Exponencial compleja discreta.

La señal exponencial compleja es una secuencia definida por:

$$x(n)=C\alpha^n \tag{1.13}$$

Donde tanto α como C son números complejos y α puede representarse como:

$$\alpha=e^{\beta} \tag{1.14}$$

En el caso en que β sea púramente imaginaria, la secuencia tomaría la forma:

$$C\alpha^n=|C||\alpha|^n\cos(\Omega_0 n+\theta)+j|C||\alpha|^n\text{sen}(\Omega_0 n+\theta) \tag{1.15}$$

Por lo tanto, esta expresión general de la exponencial compleja discreta representa, tanto en su parte real como imaginaria, a una senoidal para α =1, y a una senoidal con envolvente exponencial para los casos en que α<>1. Si α>1, la senoidal es creciente, mientras que si α<1 la senoidal es decreciente, [OPP90]. Esto se aprecia en la fig. 1.5.

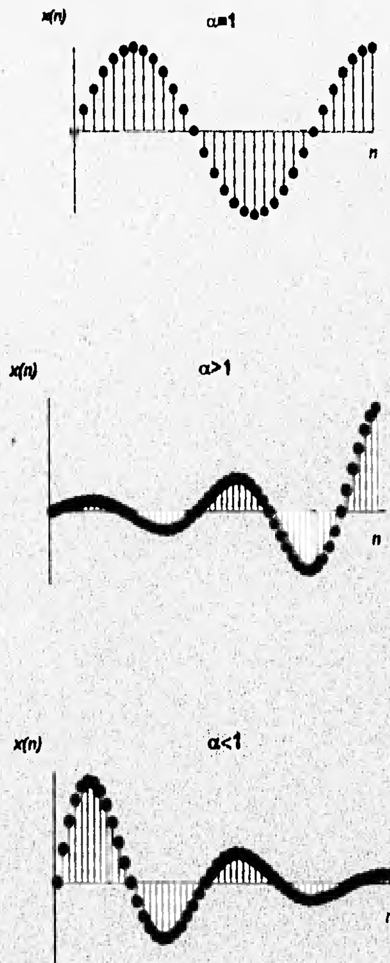


Fig. 1.5

1.1.6.5 Propiedades de periodicidad de señales exponenciales complejas discretas.

Las señales senoidales discretas tienen las siguientes características:

a) Una exponencial compleja discreta es periódica si θ es un número racional.
b) Señales senoidales en el tiempo discreto cuyas frecuencias están separadas

por un múltiplo entero de 2π, son idénticas.

c) Las señales senoidales en tiempo discreto alcanzan las mayores tasas de oscilación cuando $\omega=\pi$ o $\omega=-\pi$, [PRO88].

1.1.6.6 Exponenciales complejas relacionadas armónicamente.

En el análisis de sistemas y señales, resultan útiles los conjuntos de exponenciales complejas relacionadas armónicamente, cuyas frecuencias fundamentales son múltiplos de una frecuencia positiva. Para una señal con frecuencia fundamental $f_0=1/T$, podrán escogerse T exponenciales complejas consecutivas, de $k=k_0$ a $k=k_0+T-1$, para formar un conjunto relacionado armónicamente.

1.2 SISTEMAS DISCRETOS.

1.2.1 Generalidades.

Una manera objetiva de *conceptualizar* un sistema es verlo como cualquier proceso que transforma una señal, y la forma más común de *visualizarlo* es como una "caja negra" con entradas y salidas. Específicamente, un sistema en tiempo discreto es un sistema que transforma una o varias señales de entrada en tiempo discreto a señales de salida también en tiempo discreto según una regla bien definida, [ALC88].

La idea de representar un sistema como una "caja negra" nos permite entender mejor la interconexión entre los sistemas, lo cual es muy importante ya que ésta es una herramienta indispensable en el estudio de los sistemas.

Existen tres formas fundamentales para interconectar los sistemas: ***Interconexión en serie***, aquí la salida de un sistema es la entrada de otro sistema, y cualquier entrada es transformada primero por el sistema 1 y luego por el sistema 2. ***Interconexión en paralelo*** la entrada es la misma para los dos sistemas y la salida es la suma de la salida del sistema 1 con la salida del sistema 2. Con las combinaciones de las dos formas anteriores obtenemos interconexiones más complicadas. En el sistema de ***Feedback o retroalimentación,*** la señal de salida del sistema 1 es la entrada del sistema 2, mientras que la salida del sistema 2 reatroalimenta al sistema y sumándose con la señal de entrada externa produce la señal de entrada actual del sistema 1, [OPP90] (fig. 1.6).

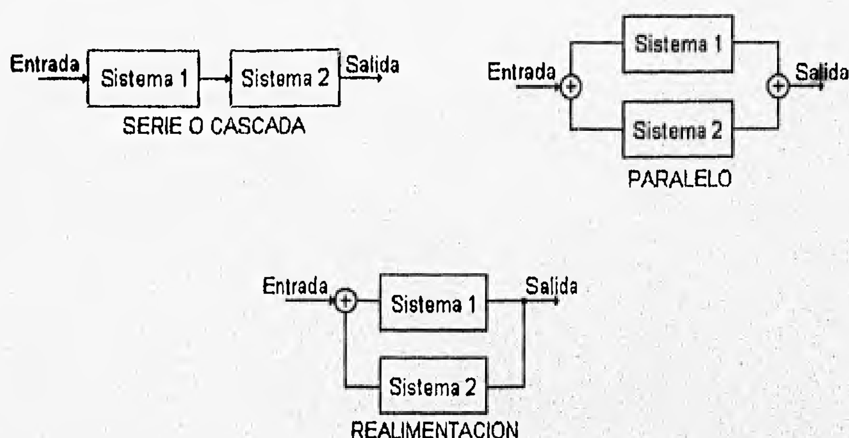


Fig. 1.6

Siempre debe existir una descripción entre la entrada y salida. Por lo general esta descripción es una expresión matemática o regla que define explícitamente la relación entre la entrada y la salida, o sea, el tipo de transformación que va a sufrir la señal de entrada para producir una señal de salida determinada. Esta regla puede expresarse esquemáticamente como :

$$x(n) \xrightarrow{H} y(n)$$

La cual indica que *y(n)* es la respuesta del sistema *H* a la excitación *x(n)*, [PRO88].

Es muy importante mencionar que para muchos sistemas la respuesta del sistema en un instante determinado, no depende únicamente del valor de la entrada en éste instante, sino de los valores de entrada aplicados al sistema antes y después de éste instante. Para ello es necesario considerar las condiciones iniciales del sistema antes de obtener la salida en un instante deseado, [OPP90].

1.2.2 Representación de sistemas en diagramas de bloques.

La representación de un sistema discreto en un diagrama de bloques es una herramienta muy útil en el análisis, diseño y realización de sistemas discretos. Para poder iniciar esta introducción necesitamos definir algunos bloques básicos preconstruidos que se pueden interconectar para armar sistemas más complejos.

1.2.2.1 Sumador.

Este bloque consiste en la adición de dos o más señales para formar una tercera. Esta operación no requiere memoria (fig. 1.7).

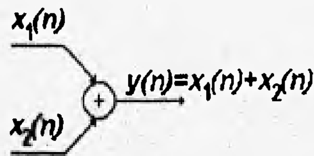


Fig. 1.7

1.2.2.2 Multiplicación por un escalar.

Esta operación simplemente es la aplicación de un factor escalar a la entrada *x(n)*. Matemáticamente, es la multiplicación de un escalar por la señal de entrada (fig. 1.8).

Fig. 1.8

1.2.2.3 Multiplicación de señales.

La multiplicación de dos señales nos genera una tercera, como lo muestra la fig. 1.9

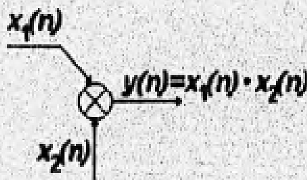


Fig. 1.9

1.2.2.4 Elemento de retraso unitario.

En este bloque, si la señal de entrada es $x(n)$, la salida es $x(n-1)$. Esto indica que el valor de la señal $x(n-1)$ es almacenada en memoria en el tiempo $n-1$ y es llamada desde la memoria en el tiempo n para obtener la respuesta (fig. 1.10).

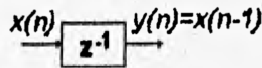


Fig. 1.10

1.2.2.5 Elemento de adelanto unitario.

Esta operación es físicamente imposible de realizar en tiempo real, ya que de la entrada que es $x(n)$ se obtiene una salida $x(n+1)$, no es posible adelantarse al futuro de la señal, es por ésto que utilizamos memoria para almacenar la señal y posteriormente poderla llamar y en una aplicación fuera de línea poder avanzarla en el tiempo (fig. 1.11).

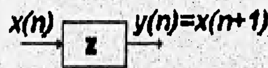


Fig. 1.11

Una vez definidos estos bloques que serán de mucha ayuda, presentaremos una clasificación de los sistemas según sus propiedades básicas.

1.2.3 Clasificación de los sistemas discretos según sus propiedades.

1.2.3.1 Estático y dinámico.

Un sistema se dice estático si no tiene memoria, es decir si la salida del sistema en un instante n depende únicamente del valor de la muestra de entrada en ese mismo instante n. El sistema es dinámico si tiene memoria, es decir si el valor de la salida en un instante n no sólo depende del valor de la muestra de entrada en el instante n sino de valores pasados o futuros de la señal de entrada.

1.2.3.2 Variante e invariante con el tiempo.

Un sistema es llamado invariante con el tiempo si para una variación en el tiempo de la señal de entrada no existe variación en la señal de salida. Si existe la misma variación en la señal de salida, el sistema es variante con el tiempo.

1.2.3.3 Lineales.

Un sistema se dice lineal si posee el principio de superposición, cumpliendo las dos propiedades siguientes:

Aditividad:

1.- La respuesta a la entrada $x_1(n) + x_2(n)$ es $y_1(n) + y_2(n)$

donde la respuesta a $x_1(n)$ es $y_1(n)$ y la respuesta a $x_2(n)$ es $y_2(n)$

Homogeneidad o multiplicación por un escalar:

2.- La respuesta a la entrada $ax_1(n)$ es $ay_1(n)$

1.2.3.4 Causales.

Un sistema se dice causal si la salida en cualquier instante *n* depende únicamente de las entradas actuales o anteriores a ese instante *n*, pero no de las entradas futuras. Si un sistema no cumple con la definición anterior se dice que es un sistema no causal. Un sistema en tiempo real sólo puede realizarse si es causal, pues no pueden adivinarse muestras de entrada futuras. Un sistema no causal sólo puede implantarse fuera de línea.

1.2.3.5 Estables e Inestables.

Un sistema es estable si para entradas acotadas, las salidas lo son también. Si la entrada es acotada o finita, y la salida genera una señal infinita o no acotada, el sistema es inestable. Los sistemas inestables generalmente presentan un comportamiento errático y extremoso y causan una sobrecarga en algunas implantaciones prácticas.

1.3 ANALISIS DE SISTEMAS DISCRETOS LINEALES E INVARIANTES CON EL TIEMPO.

Una vez enumerados los distintos tipos de sistemas en tiempo discreto, se enfocará la atención en una categoría muy importante de los mismos: los sistemas lineales e invariantes con el tiempo, que a partir de este momento llamaremos SLIT (Sistemas Lineales e Invariantes con el Tiempo). A continuación se presentarán las características fundamentales de los principales conceptos del análisis de sistemas. El análisis consiste en determinar el comportamiento de un sistema, lo que genericamente es conocer su respuesta (o salida) dada una cierta excitación (o entrada).

1.2.1 Análisis de sistemas lineales en el dominio del tiempo.

El análisis en el dominio del tiempo consiste en observar como varía alguna de las características de la señal de salida o respuesta del sistema conforme pasa el tiempo. Existen dos métodos básicos para el análisis en el dominio del tiempo del comportamiento o respuesta de un sistema lineal dada una señal de entrada. Estos métodos son:

1.2.1.1 Relación entrada-salida o ecuación en diferencias.

Este método consiste en la solución directa de la ecuación entrada-salida del sistema, la cual tiene la siguiente forma general:

$$y(n)=F[y(n-1), y(n-2), ..., y(n-N), x(n), x(n-1), ..., x(n-M)] \quad (1.16)$$

Donde F[ ] denota alguna función de las cantidades entre paréntesis. Para un sistema LIT, la forma general de la ecuación se denomina ecuación en diferencias y se expresa como:

$$y(n)=- a_i y(n-i)+ b_j x(n-j) \quad (1.17)$$

Donde a_i y b_j son parámetros constantes, [PRO88].

Generalmente, los casos de interés son aquéllos de sistemas causales, es decir, aquellos sistemas cuya salida depende exclusivamente de los valores presente y pasados de la entrada, es decir, aquéllos en los que $x(n)$ y $y(n)$ son nulas para valores de n menores a 0.

Las ecuaciones en diferencias pueden ser resueltas numérica y analíticamente. En el primer caso, es necesario conocer la secuencia $x(n)$ de entrada y un cierto número de

condiciones iniciales de la salida, número que dependerá del orden del sistema. Un sistema es lineal si sus condiciones iniciales son nulas, [ALC88].

La solución analítica de las ecuaciones en diferencias es parecida a la de las ecuaciones diferenciales, es decir, inicia obteniendo la solución homogénea y después continúa proponiendo una solución particular, [PRO88].

Los sistemas discretos pueden ser implantados directamente a través de las ecuaciones en diferencias y los elementos básicos de los sistemas discretos, como son el sumador, el retardo unitario y la multiplicación por un escalar.

1.2.1.2 Descomposición de la señal de entrada en una suma de señales elementales.

Este segundo método consiste en descomponer la señal de entrada en una suma de señales elementales cuya respuesta sea fácilmente determinada. De esta forma, por la propiedad de la linealidad de los sistemas discretos, puede obtenerse la respuesta total del sistema sumando las respuestas de cada una de las entradas elementales [PRO88].

La selección lógica de las señales a ser empleadas en la descomposición es el impulso o muestra unitaria. Puede apreciarse la validez de esto de la siguiente forma:

Para una señal $x(n)$, multiplicamos cada uno de los elementos $x(k)$ por un impulso unitario desplazado hasta el tiempo donde se encuentra el mismo elemento, es decir, por $\delta(n-k)$. El impulso unitario desplazado será nulo para cualquier tiempo distinto de k, donde valdrá uno, lo que quiere decir que como resultado de la multiplicación tendremos a todos los valores de $x(n)$ anulados excepto a $x(k)$, que será multiplicado por el impulso unitario en ese tiempo. De esta forma, es posible probar que toda secuencia puede ser descompuesta en la siguiente suma:

$$x(n)= \Sigma x(k)\,\delta(n-k) \qquad (1.18)$$

Si a cada uno de los sumandos de la sumatoria lo nombramos x_k, tenemos que la respuesta para cada uno está dada por $y_k=F[x_k]$, y la respuesta total es entonces:

$$y(n)=\sum_{k=-\infty}^{\infty} y_k \qquad (1.19)$$

1.2.1.3 Respuesta de los Sistemas Lineales e Invariantes con el Tiempo a una entrada arbitraria: Sumatoria de Convolución.

Habiendo una vez expresado una secuencia de entrada $x(n)$ a un SLIT como la suma de impulsos multiplicados por un escalar, puede obtenerse la respuesta del sistema para cualquier entrada conociendo primero la respuesta del sistema para una entrada impulso unitario. Esta respuesta se expresa de la siguiente forma [PRO88]:

$$y(n)=H[\delta(n)]=h(n) \tag{1.20}$$

Así, si se tiene una entrada arbitraria expresada en forma de suma de impulsos unitarios:

$$x(n)=\sum_{k=-\infty}^{\infty} x(k)\delta(n-k) \tag{1.21}$$

a respuesta estaría dada por:

$$\begin{aligned} y(n)&=H[x(n)]=H[\sum_{k=-\infty}^{\infty} x(k)\delta(n-k)] \\ &=\sum_{k=-\infty}^{\infty} x(k)H[\delta(n-k)] \\ &=\sum_{k=-\infty}^{\infty} x(k)h(n-k) \end{aligned} \tag{1.22}$$

A esta sumatoria que expresa la respuesta de un sistema lineal e invariante con el tiempo dadas una entrada arbitraria y la respuesta a entrada impulso unitario, se le llama sumatoria de convolución y se representa mediante el operador *.

$$y(n)=x(n)*h(n) \tag{1.23}$$

La ejecución de la convolución consta de cuatro pasos: invertir la señal $h(n)$ para obtener $h(-n)$, desplazar esta señal k unidades para tener $h(n-k)$, multiplicar los elementos de esta secuencia por los elementos $x(n)$ y finalmente realizar la suma.

1.2.2 Análisis en el dominio de la frecuencia.

1.2.2.1 Espectro.

La segunda posibilidad de análisis de sistemas discretos es el análisis en el dominio de la frecuencia. Hasta el momento, se han analizado los sistemas en el dominio del tiempo de forma que la gráfica de una señal en este dominio describe la variación de la amplitud o alguna otra característica de la señal conforme varía el tiempo. El

espectro o respuesta en frecuencia de una señal muestra la presencia o ausencia de las distintas frecuencias en una señal. De esta forma, si se tiene una señal con una frecuencia constante f_0, la representación del espectro sería una gráfica con la frecuencia como variable independiente y una sola espiga vertical colocada sobre el punto correspondiente al valor f_0 de la escala horizontal. Si la variable dependiente de interés es la amplitud, la altura de la espiga sería proporcional a la amplitud de la señal.

Cabe mencionar un aspecto muy importante en el análisis en frecuencia de las señales en que la frecuencia varía constantemente, como es el caso de la música: para obtener con precisión el espectro de una señal es necesario realizar el análisis en un período completo de ella, de lo contrario, el análisis puede no resultar significativo. Por ejemplo, el caso en que la información provenga de una sola muestra de la señal; resulta obvio que esta información sería completamente insuficiente para el análisis en la frecuencia de la señal, [DAT91].

1.2.2.2 Transformadas.

Las transformadas son herramientas matemáticas desarrolladas para llevar la representación de una señal del dominio del tiempo al dominio de la frecuencia y viceversa. Existen diversas transformadas, y pueden clasificarse en transformadas continuas, transformadas en series y transformadas discretas. Las primeras se aplican sobre señales continuas; si estas señales tienen sólo un número finito de frecuencias, es decir, frecuencias discretas, la transformada continua se reduce a una transformada en series, por lo que éstas son un caso particular de las transformadas continuas. Por su parte, las transformadas discretas se aplican sobre señales discretas con frecuencias discretas. Las transformadas permiten llevar la representación de un sistema a un dominio donde las operaciones matemáticas necesarias para su análisis y diseño son más fáciles, y una vez hecho lo anterior, puede regresarse al dominio original aplicando la transformación inversa o antitransformación correspondiente, [DAT91].

1.2.3 Transformada Z.

1.2.3.1 Definición, Región de Convergencia y Transformada inversa.

La transformada Z de una señal $x(n)$ en el tiempo discreto se define mediante:

$$Z\{x(n)\}=X(z)=\sum_{n=-\infty}^{\infty} x(n)z^{-n} \tag{1.24}$$

donde $z=e^{j\omega T}$.

La sumatoria que define a la transformada Z tendrá sentido si converge a un límite

definido en el plano complejo Z. Puede notarse que la transformada es una sumatoria infinita, y por lo tanto puede no ser convergente para todas las secuencias. "Para una secuencia dada, el conjunto de valores en el cuál la transformada converge es llamada *Región de Convergencia*", [DIE94].

Dentro de las diversas propiedades que presenta la transformada Z [PSE95], la más importante es la de convolución, que dice:

Si existen las transformadas

$$x_1(n) \leftrightarrow X_1(z) \qquad (1.25)$$

$$x_2(n) \leftrightarrow X_2(z) \qquad (1.26)$$

Y existe su convolución:

$$x(n) = x_1(n) * x_2(n) \qquad (1.27)$$

La transformada de ésta es:

$$x(n) \leftrightarrow X(z) \qquad (1.28)$$

$$X(z) = X_1(z) \bullet X_2(z) \qquad (1.29)$$

Al proceso de obtención de la secuencia original en el dominio del tiempo a partir de la transformada Z se le conoce como inversión de la transformada o antitransformación. Este proceso resulta muy útil para conocer la respuesta de un sistema después de haber realizado las operaciones, matemáticamente más sencillas, en el dominio de la transformada.

Existen tres métodos comunes para la obtención de la transformada Z inversa, que son [CAD85]:

1.- Expansión en una serie de términos de las variables z y z^{-1}

2.- Integración compleja mediante el método de los residuos.

3.- Expansión en fracciones parciales y búsqueda en tablas, [DIE94].

1.2.3.2 Función de Transferencia.

Este es uno de los conceptos más importantes del análisis de sistemas discretos. La función de transferencia es la razón de la transformada Z de la señal de salida de un sistema entre la transformada Z de la señal de entrada al mismo. La función de transferencia se obtiene a partir de la propiedad de convolución de señales. Recordando esta propiedad, se tiene que la respuesta de un sistema es la convolución de la señal de entrada y la respuesta a impulso. Esto es:

$$y(n)=x(n)*h(n) \quad (1.30)$$

donde $y(n)$ es la salida, $x(n)$ es la entrada y $h(n)$ es la respuesta a impulso unitario del sistema. Si se aplica la transformada Z a cada lado de la igualdad, tendremos:

$$Y(z)=Z\{x(k)*h(k)\} \quad (1.31)$$

De la propiedad de convolución, el lado derecho quedaría:

$$Z\{x(k)*h(k)\}=X(z)\bullet H(z) \quad (1.32)$$

Por lo tanto:

$$Y(z)=\frac{X(z)}{H(z)} \quad (1.33)$$

Despejando $H(z)$:

$$H(z)=\frac{Y(z)}{X(z)} \quad (1.34)$$

A $H(z)$ se le conoce como función de transferencia del sistema con entrada $x(n)$, respuesta a impulso $h(n)$ y salida $y(n)$.

"La función de transferencia tiene una trascendencia muy grande en el análisis de sistemas basado en el modelo de entrada-salida. A partir de ella se han desarrollado diversos métodos de interpretación, análisis y diseño de sistemas discretos, incluso los importantes conceptos de estabilidad y respuesta en frecuencia han sido abordados tradicionalmente desde esta base", [DIE94]

1.4 ALGORITMOS DE PROCESAMIENTO DIGITAL DE SEÑALES.

En esta parte introduciremos los principios del Procesamiento Digital de Señales (DSP). En primer lugar desarrollaremos los algoritmos de correlación, que son técnicas que permiten extraer la relación entre dos señales. Posteriormente, presentaremos los conceptos y algoritmos de análisis de sistemas discretos en el dominio de la frecuencia, como la Transformada Discreta de Fourier y la Transformada Rápida de Fourier. Finalmente presentamos la introducción al diseño de filtros digitales.

1.4.1 Correlación.

Un concepto que tiene mucho que ver con la convolución es el de correlación. En este caso, se tienen dos señales y se desea saber qué tanto se parecen una a la otra. El ejemplo tradicionalmente utilizado para explicar este proceso y su importancia es el del radar. Mediante el radar desea conocerse la presencia y, en caso de presentarse ésta,

la distancia a la que se encuentra un objeto Si se tiene una secuencia *x(n)* como salida rastreadora de radar, esta señal regresará con cierto retraso debido al tiempo entre el envío de la señal y el reflejo (o rebote) del objeto (un avión, por ejemplo) sobre la misma y regresará además corrompida por el ruido. En este caso, la señal regresada podría modelarse de la siguiente forma:

$$y(n) = \alpha x(n-D) + \omega(n) \qquad (1.35)$$

donde α es el factor que indica la pérdida de la amplitud de la señal, *D* es el retraso y $\omega(n)$ es el ruido que corrompe a la señal de entrada *x(n)*.

Si a partir de la señal regresada puede obtenerse el retraso *D*, podrá obtenerse también la distancia a la que se encuentra el objeto.

Dadas dos secuencias *x(n)* y *y(n)*, la croscorrelación de la secuencia *x(n)* y *y(n)* es la secuencia $r_{xy}(l)$ definida por:

$$r_{xy}(l) = \sum_{l=-\infty}^{\infty} x(n)y(n-l) \qquad (1.36)$$

Puede obtenerse asimismo la croscorrelación de *y(n)* con respecto de *x(n)* intercambiando las variables en las ecuaciones anteriores.

Otro tipo de correlación que puede efectuarse es la llamada autocorrelación, y que se efectúa sobre la misma señal.

$$r_{xy}(l) = \sum_{l=-\infty}^{\infty} x(n)x(n-l) \qquad (1.37)$$

Mediante la correlación puede reconstruirse una señal destruída por el ruido, pues en la secuencia obtenida de la correlación se identifican los períodos de la señal original, dado que el ruido no es periódico.

1.4.3 Transformada Discreta de Fourier.

La Transformada Discreta de Fourier (DFT por sus siglas en inglés) es la herramienta utilizada para obtener el espectro o respuesta en frecuencia de una onda compleja en términos de un conjunto de senoidales relacionadas armónicamente, cada una con una amplitud y una fase particulares.

La Transformada de Fourier para una señal discreta se define como [SMI85]:

$$X(\omega) = \sum_{-\infty}^{\infty} x(n)e^{-j\omega t} \qquad (1.38)$$

Para una secuencia periódica de N muestras, se define la Transformada Discreta de Fourier (DFT) [PSE95]:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi kn}{N}} \text{ para } k=0,1,2,\ldots,N-1 \tag{1.39}$$

donde

$$e^{-j\frac{2\pi kn}{N}} = \cos(\omega nk) - j\text{sen}(\omega nk) \tag{1.40}$$

$$\omega = \frac{2\pi}{n} \tag{1.41}$$

La transformada inversa se define [PSE95]:

$$X^{-1}(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) W_N^{-kn} \tag{1.42}$$

donde

$$W_N = e^{-j(\frac{2\pi}{N})} \tag{1.43}$$

Los valores de *X(k)* (el espectro) son complejos, con la parte real correspondiendo a los coeficientes a_k (coseno, componente par de la señal) y la parte imaginaria correspondiendo a los coeficientes b_k (seno, parte impar de la señal). Por lo tanto, si expresamos a *X(k)* como número complejo en su forma binomial:

$$X(k)=a+jb \tag{1.44}$$

Su magnitud o módulo es:

$$|X(k)| = \sqrt{a^2 + b^2} \tag{1.45}$$

y su fase es:

$$\Phi(k) = \tan^{-1}(\tfrac{b}{a}) \tag{1.46}$$

La DFT tiene una serie de propiedades importantes que facilitan el manejo de operaciones entre señales y de cambios de la variable independiente. Pseniscka proporciona una lista detallada de las propiedades [PSE95]. Pero entre ellas cabe hacer notar las siguientes:

- El espectro de una señal discreta y periódica es también discreto y periódico.
- El espectro de una señal discreta y no periódica es continuo y periódico.

1.4.4 Transformada Rápida de Fourier.

El cálculo de la DFT de un conjunto de N muestras requiere N^2 multiplicaciones y N^2-N sumas, todas ellas complejas, por lo que este cálculo directo puede considerarse

ineficiente. Se han desarrollado algoritmos de cálculo de la DFT dividiendo la secuencia de entrada en secuencias más pequeñas. Estos algoritmos son llamados "Transformada Rápida de Fourier" FFT (Fast Fourier Tranform). Cuando el número de datos $N=r^v$ (donde r es un número primo) las DFT's serán de tamaño r, el cual es conocido como rádix del algoritmo FFT y es un número entero.

1.4.4.1 Decimación en el tiempo, Rádix-2.

Si el número de datos es N, $N=ML=2^r$, puede seleccionarse $M=\frac{N}{2}$ y $L=2$. La secuencia $x(n)$ se divide en dos secuencias de $\frac{N}{2}$ datos, $f_1(n)=x(2n)$ y $f_2(n)=x(2n+1)$ que corresponden a las muestras pares e impares de $x(n)$, respectivamente. Estas dos secuencias $f_1(n)$ y $f_2(n)$ se obtuvieron de decimar la secuencia original por un factor de 2. De ahí el nombre del algoritmo.

$$x(n)=f_1(n)+f_2(n)=x(2n)+x(2n+1) \qquad n=0,1,2,3,\tfrac{N}{2}-1 \tag{1.47}$$

$$f_1(n)=\{x(0),x(2),x(4),...\} \qquad f_2(n)=\{x(1),x(3),x(5),...\} \tag{1.48}$$

La transformada de Fourier se expresa entonces:

$$X(k)=\sum_{n=0}^{N-1} x(n)W_N^{kn} \tag{1.49}$$

donde $W_N=e^{-j\frac{2\pi}{N}}$ y, por propiedades de periodicidad:

$$W_N^2=W_{\frac{N}{2}} \tag{1.50}$$

sustituyendo:

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1} f_1(x)W_{\frac{N}{2}}^{nk}+W_N^k\sum_{m=0}^{\frac{N}{2}-1} f_2(n)W_{\frac{N}{2}}^{nk} \tag{1.51}$$

$$X(k)=F_1(k)+W_N^kF_2(k) \qquad k=0,1,...,N-1 \tag{1.52}$$

$$X(k+\tfrac{N}{2})=F_1(k)+W_N^{k+\frac{N}{2}}F_2(k) \tag{1.53}$$

Si sustituimos por $W_N^{k+\frac{N}{2}}=-W_N^k$ se tiene:

$$X(k)=F_1(k)+W_N^kF_2(k) \tag{1.54}$$

$$X(k+\tfrac{N}{2})=F_1(k)-W_N^kF_2(k) \tag{1.55}$$

Donde $F_1(k)$ y $F_2(k)$ son las DFT's de $\frac{N}{2}$ puntos de las secuencias $f_1(n)$ y $f_2(n)$

respectivamente. Si a cada una de ellas se le aplica la decimación y se continúa **decimando hasta** que las secuencias resultantes sean de un sólo punto, para $N=2^v$ el **algoritmo reduce** a $\frac{N}{2}\log_2 N$ el número de multiplicaciones y a $N\log_2 N$ el número de **sumas** requeridas para obtener la Transformada Discreta de Fourier, lo cual es menor **que para la** DFT.

1.4.4.2 Decimación en la frecuencia.

Este algoritmo es muy parecido al de decimación en el tiempo, pero en este caso la **decimación, es** decir, la división de la secuencia en dos partes, una par y otra impar, **se realiza ya en el** dominio de la frecuencia. De esta forma:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} X(n)W_N^{nk} + \sum_{n=\frac{N}{2}}^{N-1} x(n)W_N^{nk} \tag{1.56}$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n)W_N^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x(n+\tfrac{N}{2})W_N^{\frac{N}{2}k} \bullet W_N^{nk} \tag{1.57}$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n)W_N^{nk} + W_N^{\frac{N}{2}k}\sum_{n=0}^{\frac{N}{2}-1} x(n+\tfrac{N}{2})W_N^{nk} \tag{1.58}$$

Dado que $W_N^{\frac{N}{2}k} = (e^{-j\pi})^k = (-1)^k$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + (-1)^k x(n+\tfrac{N}{2})\right]W^{2kn} \tag{1.59}$$

Ahora se realiza la decimación en la frecuencia, separando las muestras pares e impares:

$$X(2r) = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + x(n+\tfrac{N}{2})\right]W_N^{2rn} \tag{1.60}$$

$$X(2r+1) = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) - x(n+\tfrac{N}{2})\right]W_N^{(2r+1)n} \tag{1.61}$$

De la propiedad de simetría $W_N^{2rn} = W_{\frac{N}{2}}^{rn}$ **se obtiene:**

$$X(2r) = \sum_{n=0}^{\frac{N}{2}-1} \left[x(n) + x(n+\tfrac{N}{2})\right]W_{\frac{N}{2}}^{rn} \tag{1.62}$$

$$X(2r+1)=\sum_{n=0}^{\frac{N}{2}-1}\left[x(n)+x(n+\tfrac{N}{2})\right]W_N^n \bullet W_{\frac{N}{2}}^{rn} \qquad (1.63)$$

El número de sumas y multiplicaciones también es del orden de $\frac{N}{2}\log_2 N$ y $N\log_2 N$, respectivamente.

1.4.5 Filtros digitales.

Un filtro digital es la relación entrada-salida de señales digitales. Puede representarse mediante una ecuación en diferencias, mediante su respuesta a impulso o mediante su función de transferencia, [DAT91]. Se dice también que un filtro digital es un programa de cómputo que toma una secuencia de números (señal de entrada) y produce una nueva secuencia de números (señal de salida), [SMI85].

La forma general de una ecuación en diferencias es [PSE95]:

$$y(n)=\sum_{i=0}^{M} b_i x(n-i)-\sum_{j=1}^{N} a_j y(n-j) \qquad (1.64)$$

Aplicando transformada Z

$$Y(z)=\sum_{i=0}^{M} b_i z^{-i} X(z)-\sum_{j=1}^{N} a_j z^{-j} Y(z) \qquad (1.65)$$

Factorizando

$$Y(z)\left[1+\sum_{j=1}^{N} a_j z^{-j} Y(z)\right]=X(z)\sum_{i=0}^{M} b_i z^{-1} \qquad (1.66)$$

$$H(z)=\frac{Y(z)}{X(z)}=\frac{\sum_{i=0}^{M} b_i z^{-i}}{1+\sum_{j=1}^{N} a_j z^{-j}} \qquad (1.67)$$

"El diseño de un filtro digital consiste en determinar los coeficientes de la función de transferencia que satisfacen una relación entrada-salida a priori bajo un cierto criterio", [ALC88]

Existen dos tipos de filtros digitales: filtros con respuesta al impulso infinita (IIR) y filtros con respuesta al impulso finita (FIR). A los primeros se les conoce también como filtros recursivos y a los segundos como no recursivos.

1.4.5.1 Filtros IIR.

Un filtro IIR es aquél cuya respuesta a impulso tiene una duración infinita. Su función de transferencia es [PSE95]:

$$H(z)=\frac{Y(z)}{X(z)}=\frac{b_0+b_1z^{-1}+b_2z^{-2}+...+b_kz^{-k}}{1+a_1z^{-1}+a_2z^{-2}+...+a_kz^{-k}}=\frac{\sum_{l=0}^{k} b_l z^{-l}}{1+\sum_{l=1}^{k}} \qquad (1.68)$$

Usando la transformada Z inversa:

$$y(n)=\sum_{l=0}^{k} b_l x(n-l)-\sum_{l=1}^{k} a_l y(n-l) \qquad (1.69)$$

Haciendo uso de las operaciones básicas de sistemas discretas, se tienen los elementos del procesamiento digital de señales.

A) Sumador:

Fig. 1.12

B) Multiplicador por un escalar:

Fig. 1.13

C) Elemento de retardo:

Fig. 1.14

Con estos elementos básicos, puede implantarse cualquier filtro digital.

De esta forma, la representación directa de un filtro IIR se presenta en la figura 1.15.

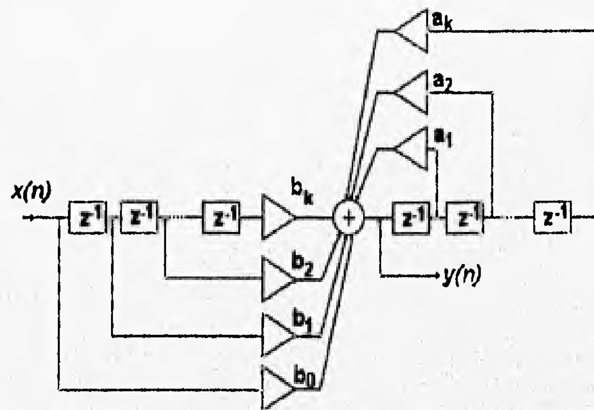


Fig. 1.15

Esta representación no es canónica, pues tiene $2k$ elementos de retardo, mientras el orden del filtro es k. Para convertir de esta forma a la 1a. forma canónica se aplican las siguientes reglas:

1) Se cambian los nodos por sumadores y viceversa.
2) Se cambia la dirección de los amplificadores.
3) Se cambia la entrada la entrada por la salida, [PSE95].

Así, la primera forma canónica del filtro IIR se presenta en la figura 1.16.

La dependencia de los valores de las salidas pasadas (recursividad) provoca que la respuesta del filtro sea de duración infinita, aún cuando las muestras de entrada hayan terminado, [ALC88].

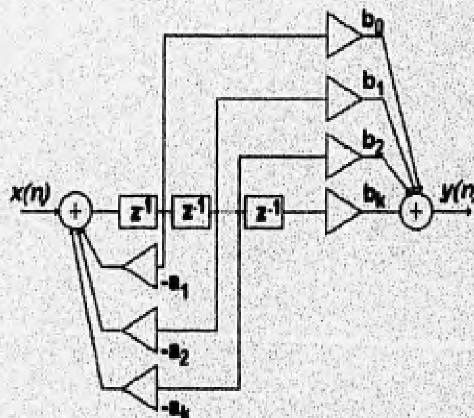


Fig. 1.16

1.4.5.2 Filtros FIR.

Un filtro FIR es aquél que tiene una respuesta a impulso con duración finita, puesto que la salida depende sólo de la entrada presente y las entradas anteriores, no de las salidas anteriores, es decir, es no recursivo.

Su función de transferencia es:

$$H(z) = \frac{Y(z)}{X(z)} = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_k z^{-k} \tag{1.70}$$

$$H(z) = \sum_{i=0}^{k} b_i z^{-i} \tag{1.71}$$

Antitransformando, se obtiene la ecuación en diferencias:

$$y(n) = \sum_{i=0}^{k} b_i x(n-i) \tag{1.72}$$

Su representación directa se presenta en la figura 1.17 y la primera forma canónica se representa en la figura 1.18.

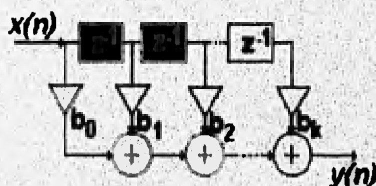


Fig. 1.17

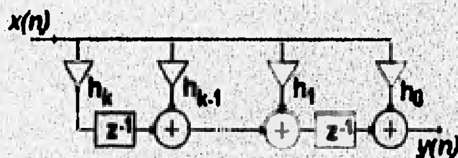


Fig. 1.18

Puede observarse que los coeficientes de la función de transferencia son asimismo las muestras de la respuesta al impulso. Las ventajas y desventajas de estos tipos de filtros digitales son [PSE95]:

Ventajas de los filtros FIR.

1) Los filtros FIR siempre son estables (todos sus polos están en el origen).
2) La fase de la función de transferencia es lineal.
3) La salida no es recursiva, por lo que son menos susceptibles a errores de redondeo que los filtros IIR.

Desventajas de los filtros FIR.

1) Requieren más sumadores, amplificadores y elementos de retraso.
2) Por el uso de un mayor número de elementos, presentan mayor retraso que los filtros IIR.

CONCLUSIONES.

En este capítulo presentamos un breve resumen de los principios de las señales discretas, de los sistemas discretos y su análisis y finalmente de los principales algoritmos de procesamiento digital de señales. Podemos concluir que el procesamiento digital de señales proporciona amplias, variadas y poderosas herramientas para el análisis, el diseño y la realización de sistemas que operen sobre señales digitales, lo que implica mayor flexibilidad y calidad en la transmisión, la modificación, la clasificación y el almacenamiento de la información que pueda llevarse al dominio del tiempo discreto y cuyos valores puedan ser cuantizados. Necesariamente, esto implica una gran capacidad de cálculo matemático, así como la necesidad de arquitecturas y software de control y aplicación muy eficientes, especialmente si se desean realizar sistemas que operen en tiempo real. A partir de esta teoría básica, pueden identificarse los elementos necesarios para una metodología de diseño de aplicaciones DSP y las herramientas que faciliten y automaticen algunos de estos elementos.

REFERENCIAS.

[ALC88] Alcántara, Rogelio. *Apuntes de Procesamiento Digital de Señales.* DEPFI-UNAM, México, 1988.

[CAD85] Cadzow, James A. y Van Landingham, Hugh. *Signals, Systems and Transforms.* Prentice-Hall, Estados Unidos, 1985.

[DAT91] Datta, Jayant. "Digital Signal Processing: Theory" en *Advanced Digital Audio*, Ken C. Pohlman editor. Sams, Estados Unidos, 1991. pp. 293-346.

[DIE94] División de Ingeniería Eléctrica, Facultad de Ingeniería, UNAM. *Apuntes de Control Digital.* FI-UNAM, México, 1994.

[OPP90] Oppenheim, Alan V. *Introduction to Signals and Systems.* Prentice-Hall, Estados Unidos, 1990.

[PSE95] Psenicka, Bohumil. *Apuntes de Procesamiento Digital de Señales.* FI-UNAM, México, 1995.

[PRO88] Proakis, John G. y Manolakis, Dimitris G. *Digital Signal Processing; Principles, Algorithms, and Applications.* MacMillan, Estados Unidos, 1988.

[SMI85] Smith, Julius O. "An Introduction to Digital Filter Theory" en *Digital Audio Signal Processing; an Anthology,* John Strawn editor. William Kaufman, Estados Unidos, 1985. pp. 69-136.

2

Metodologías de Desarrollo de Software.

INTRODUCCION.

Durante las primeras décadas de la informática, el principal desafío era el desarrollo del hardware de las computadoras, de manera que se redujera el costo de procesamiento y almacenamiento de datos. Conforme se dieron los avances en microelectrónica aumentando el poder de cálculo, se comenzó a generar un nuevo problema, mejorar la calidad y el costo del software.

La evolución del desarrollo del software ha tenido constantes cambios desde hace aproximadamente 50 años. Los primeros sistemas desarrollados no poseían una metodología de diseño, es decir, una secuencia de pasos para comenzar y finalizar un sistema que realizara una acción específica. Conforme fue creciendo el número de sistemas de software que facilitaran el trabajo y la complejidad de éstos sistemas fueron surgiendo una serie de metodologías o corrientes para realizar sistemas, todas ellas desde las primeras hasta las últimas tienen pros y contras, sin embargo todas funcionan dependiendo del tipo de sistema, el cual puede ser complejo o muy simple.

Dentro del desarrollo del software existe una división de categorías para los sistemas de software, como son sistemas en linea y sistemas en tiempo real [YOU93].

En este capítulo presentaremos una explicación de algunas metodologías o ciclos de vida existentes en el desarrollo de un sistema de software que nos permiten visualizar en una forma completa el diseño de todo un proyecto de software. Las metodologías que escogimos para explicar son: 1) la metodología clásica o ciclo de vida clásico, 2) la metodología semiestructurada, 3) la estructurada, 4) la metodología orientada a objetos y 5) Prototyping y dado que las aplicaciones DSP son, en muchas ocasiones sistemas en tiempo real, daremos un panorama general sobre el análisis de los sistemas en tiempo real como un objeto de aplicación y estudio de las diferentes metodologías.

2.1 METODOLOGIA CLASICA.

En general casi todos los sistemas tienen una fase de análisis, diseño e implantación. El ciclo de vida clásico de los sistemas o metodología clásica, engloba esta secuencia de pasos como lo muestra la fig. 2.1:

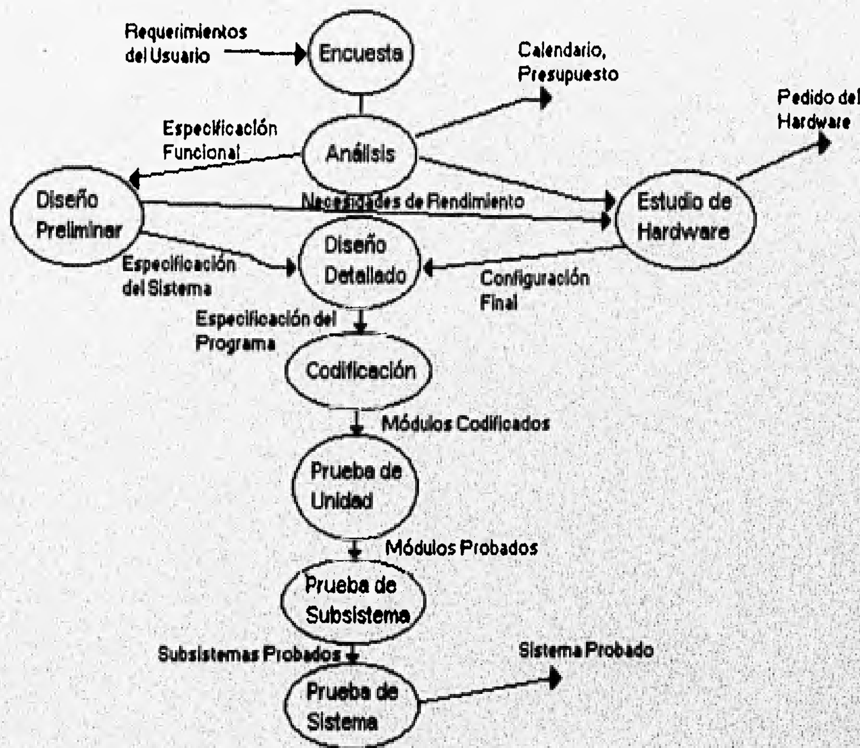


Fig. 2.1

Evidentemente estos pasos pueden cambiar entre desarrolladores o diseñadores, ya que la forma de diseñar depende mucho de la organización del trabajo de las compañías o departamentos que se dedican a desarrollar. Pero la característica principal del ciclo de vida clásico es una fuerte tendencia a la implantación ascendente del sistema y la insistencia en la progresión lineal y secuencial de una fase a otra. La implantación ascendente o ciclo de vida en cascada, fig. 2.2 como se le conoce, resulta ser una de las principales desventajas del ciclo de vida clásico, ya que es necesario que los módulos anteriores estén realizados para poder hacer pruebas al subsistema y luego al sistema completo, y así sucesivamente. La progresión lineal es inevitable, requiere que las fases se sucedan una tras otra, pensando que se ha concluido totalmente una sin volver a tocar, y pretendiendo que el diseño sea perfecto a la primera. Es aceptable que como humanos por lo general no nos salgan las cosas completamente bien a la primera, por lo que es necesario tener una oportunidad de corregir los errores hechos en una fase anterior o cambiar algunos aspectos requeridos

por el usuario que hubieran sufrido cambio durante la fase de diseño (por ejemplo su economía, reglamentos gubernamentales o la competencia). Podemos concluir que el ciclo de vida clásico es muy cerrado en el aspecto de movilidad entre las fases. Una característica adicional es que se apoya en técnicas anticuadas. Es decir tiende a ignorar el uso del análisis estructurado, la programación estructurada o cualquier otra técnica moderna de desarrollo de sistemas como herramientas CASE.

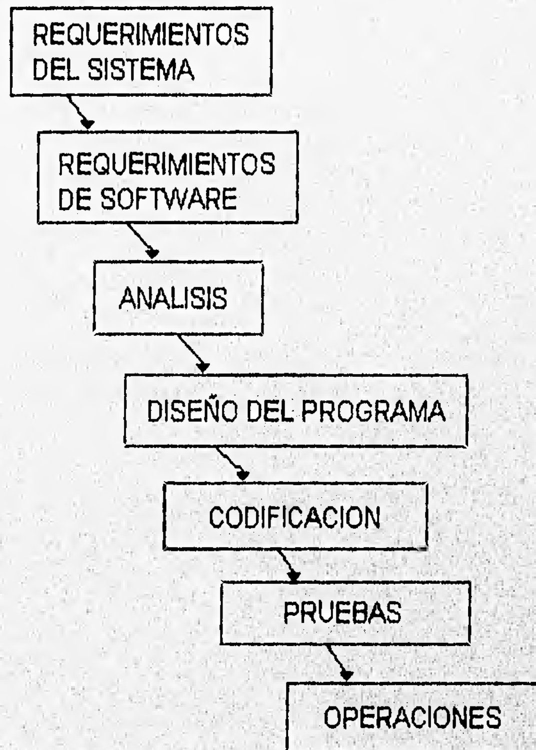
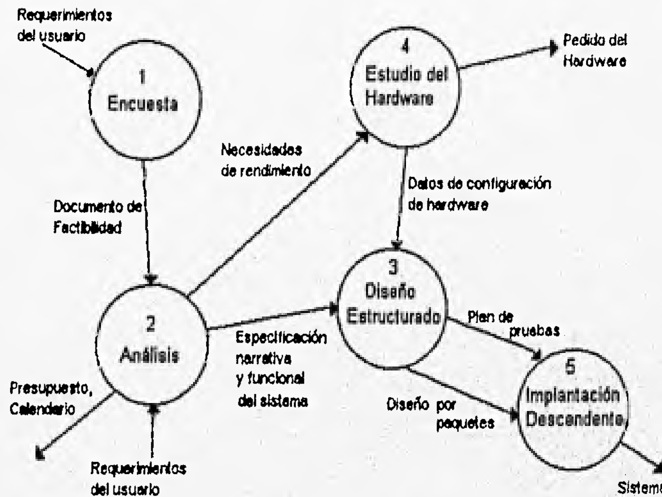


Fig. 2.2

Una ventaja de la metodología clásica es que es tan sencilla que resulta ser la más práctica ya que se adecua perfectamente para el diseño sistemas pequeños o muy simples que no requieren demasiado tiempo y muchas especificaciones [YOU93].

2.2 METODOLOGIA SEMIESTRUCTURADA.

La tendencia hacia un diseño estructurado, una programación estructurada y la implantación descendente dan como resultado la metodología semiestructurada que se muestra en la fig 2.3.



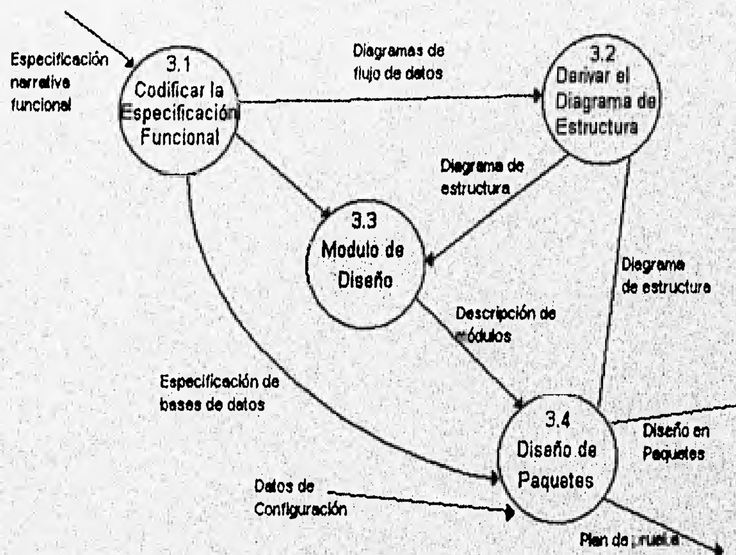


Fig. 2.3

La implantación descendente ofrece retroalimentación entre el proceso de implantación y el de análisis. Analizando en la fig. 2.3 la parte del diseño estructurado resulta ser en realidad un esfuerzo manual para corregir especificaciones erróneas. Además la implantación descendente nos ofrece también la posibilidad de ejecutar paralelamente parte de la codificación y de las pruebas. Podría decirse que se genera una retroalimentación entre la codificación, la prueba y la eliminación de fallas.

La parte más difícil del diseño semiestructurado es la traducción de un documento ambiguo, monolítico y redundante en un modelo útil y no de procedimientos, que sirva de base para crear la jerarquía de los módulos que ejecutaran los requerimientos del

usurario. Este proceso en la metodología semiestructurada se muestra en la actividad 3.1 de la fig. 2.3 dentro de la fase de diseño estructurado.

Como podemos ver, la metodología semiestructurada nos da una ventaja clara sobre la clásica, mantiene un proceso de control de errores y cambios de especificación del sistemas en lo que se refiere al diseño estructurado y permite un mayor contacto con el usuario para seguir retroalimentando las especificaciones de diseño [YOU93].

2.3 METODOLOGIA ESTRUCTURADA.

La metodología estructurada consta de nueve actividades y tres terminadores, como se muestra en la fig. 2.4. Los tres terminadores son usuarios, administradores y el personal de operación, quienes definirán las entradas del sistema y serán sus beneficiarios finales, e interactuarán en las nueve actividades de que consta la metodología.

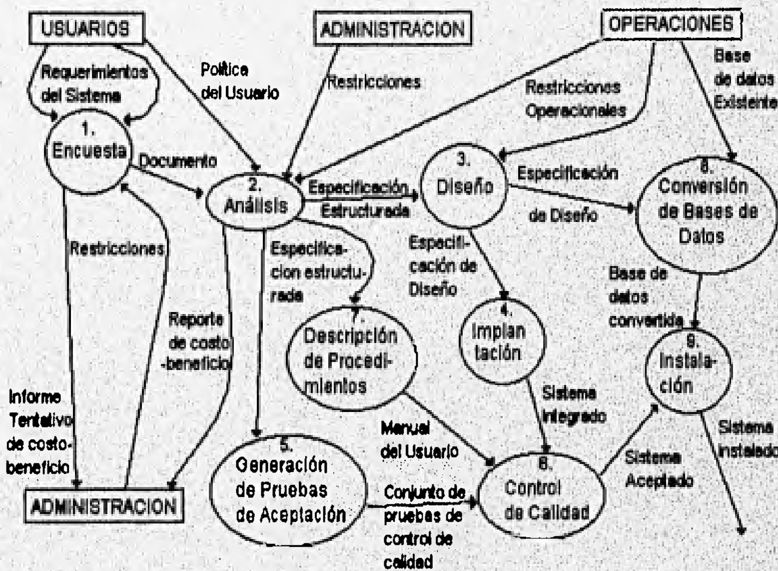


Fig. 2.4

La primera actividad es la *Encuesta*, en la cual se definen quienes serán los usuarios finales o afectados por el sistema y quienes serán los que están a cargo del sistema en colaboración con los diseñadores del sistema. ¿Qué actividades o procedimientos deberán incluirse en el sistema actual? o ¿Qué procedimientos actualmente tienen un desempeño incorrecto y pueden mejorarse en el sistema actual?, es decir, establecer los criterios de desempeño del sistema. Es sumamente importante analizar en ésta etapa si es viable o factible la construcción del sistema y si es posible automatizar el o los procesos que se quiere automatizar. También hacer el análisis costo-beneficio, para evaluar si resulta mucho más costoso hacer el nuevo sistema que mejorar los

procedimientos o sistemas actuales, por que hay veces que con solo corregir algunos procesos resulta mejor que realizar un nuevo sistema.

La segunda actividad es el *Análisis de Sistemas*. El propósito general de esta actividad es transformar sus dos entradas principales, las políticas del usuario y el esquema del proyecto, en una especificación estructurada. Esto implica modelar el ambiente del usuario con diagramas de flujo de datos, diagramas entidad-relación, diagramas de transición de estado, diagrama de estructuras, diccionario de datos, etc.

El análisis de sistema, utiliza todas esta herramientas para formar un modelo ambiental y de comportamiento, los cuales forman el modelo esencial que representa una descripción formal de lo que el nuevo sistema debe hacer, independientemente de la naturaleza de la tecnología que se usa para cubrir los requerimientos. Esta es la parte más importante del desarrollo por el método estructurado [YOU93].

En el *Diseño*, tercera actividad, se asigna una porción del modelo esencial o el resultado del análisis a procesadores adecuados (sean máquinas o humanos) y tareas apropiadas dentro de cada procesador. Dentro de cada tarea la actividad de diseño se dedica a la creación de una jerarquía apropiada de módulos de programas y de interfaces entre ellas para implantar la especificación creada en el análisis, además se transforman los modelos de datos de entidad-relación en un diseño de bases de datos. En ésta etapa se utiliza el modelo de implantación del usuario el cual describe la frontera humano-máquina que separa las partes del modelo esencial que llevará a cabo una persona (como alguna actividad manual), de las partes que se implantarán en una o más computadoras y la interfaz hombre-máquina que es una descripción del formato de las secuencias de entrada que los usuarios proporcionan a la computadora (por ejemplo el diseño de pantallas y el diálogo en línea entre el usuario y la computadora), además del formato y la secuencia de salidas -o productos- que la computadora proporciona al usuario.

La *implantación*, cuarta actividad, incluye la codificación y la integración de módulos en un esqueleto progresivamente más completo del sistema final, apoyándose de la programación estructurada como implantación descendente.

Durante la siguiente actividad que es la *Generación de pruebas de aceptación* se realizan la pruebas necesarias a partir de la especificación estructurada que se hizo del sistema en el análisis con el objetivo de presentar la información que defina al sistema, en su estructura y principalmente en su funcionamiento, para que el usuario final pueda evaluar los resultados antes de realizar la evaluación final.

Posterior a la etapa de generación de pruebas está la actividad llamada *Garantía de Calidad*, donde se realiza una prueba final o prueba de aceptación del sistema, esta actividad requiere como entradas los datos de la prueba de aceptación y el sistema integrado producido en la implantación. El analista pudiera no siempre estar involucrado con la garantía de calidad y tuviera que realizarla un departamento de control de calidad.

Un paso sumamente importante es la *Descripción del Procedimiento* el cual consiste básicamente en estructurar un manual de procedimientos con una descripción formal de las partes del sistema que se harán en forma manual, lo mismo que la descripción de como interactuarán los usuarios con la parte automatizada del nuevo sistema.

En la *Conversión de Bases de Datos* se realiza una especificación de las bases de datos tanto en estructura como en información. Es decir se analiza si se requiere una base de datos para trabajar o si ya existía una, tal vez en algunos casos es necesario reestructurar la base existente o en otros proponer algunas como necesarias para el funcionamiento del sistema y en otros casos adicionar a las ya existentes otras que mejoren el funcionamiento del sistema, todo esto se basa en el diseño hecho en la actividad 3.

La última actividad es la *instalación*, la cual se apoya en el manual de procedimiento, las bases de datos y el sistema aceptado en el paso 6. Se trata de instalar ya sea hardware, software y capacitación del personal o ambas cosas para poner a funcionar el nuevo sistema. Puede ser que sea una tarea muy simple o muy larga y algo compleja dependiendo del tipo de sistema y a quién se le instala.

En resumen el ciclo de vida estructurado como se observo en su diagrama no implica una secuencia de fases, es decir que se realice una y sólo una fase a la vez, por el contrario cada actividad puede realizarse a la par con otra, a diferencia de la metodología clásica donde todo es secuencial y se tiene que terminar una actividad para poder desarrollar otra, y aunque no está implícita una retroalimentación en la metodología estructurada, se sobreentiende o se debe sobreentender que una actividad produce información que puede alimentar a otra actividad futura o pasada para alterarla, e incluso afectar el estudio de factibilidad y viabilidad del sistema y detener bruscamente el proyecto por resultar inoperante. Esta metodología resulta ser bastante eficiente, ya que por ser tan dinámica permite realizar sistemas altamente confiables y bien pensados desde si es necesario un sistema hasta el último detalle de la implantación [YOU93].

2.4 METODOLOGIA DE DISEÑO ORIENTADA A OBJETOS.

Al igual que las demás metodologías, la metodología orientada a objetos delimita el problema del mundo real y lo asocia con una solución en software, sea un programa o sistema; pero a diferencia de las demás metodologías, ésta interconecta o relaciona objetos (elementos definidos por el analista) y operaciones de procesamiento de forma que se modulariza la información y el procesamiento en lugar de dejar aparte el procesamiento.

Un objeto puede ser concebido como un componente del mundo real asociado con un elemento en el campo del software. En el contexto de un sistema basado en computadora es un productor o consumidor o un elemento en sí de información. Este tiene asociado atributos que pueden ser heredados de una clase a la cual pertenece y

de la cual forma una instancia. Al hablar de una clase que lo engloba es hablar de un conjunto de elementos en este caso objetos o instancias que tienen características similares. Además esta clase, subclase u objeto tiene asociada una serie de métodos u operaciones que pueden realizarse sobre ellas, estas operaciones pueden ser heredadas por el objeto de una clase o subclase al igual que los atributos. Las operaciones se identifican examinando todos los verbos de la definición informal o estrategia informal. Aunque existen muchos tipos distintos de operaciones, generalmente se pueden dividir en tres grandes categorías: (1) Operaciones que manipulan los datos de alguna forma (por ejemplo adiciones, eliminaciones selecciones, cambios de formato, etc.); (2) Operaciones que realizan algún cálculo y (3) Operaciones que monitorizan un objeto frente a la ocurrencia de algún suceso de control. Una vez definidos los métodos de los objetos o clases cualquier objeto o método puede invocarlo para lo cual es necesario un mecanismo de invocación del método, a esto se le llama mensaje (mecanismo de comunicación entre objetos). Un objeto envía el mensaje de invocación conteniendo una estructura parecida a la siguiente: (destino, método, parámetros para ejecución del método) y el objeto destino procede a ejecutar el método invocado. Es de suma importancia definir las relaciones entre los diferentes objetos ya definidos para el sistema, pueden ser relaciones uno a uno, uno a muchos, muchos a muchos, etc; además al definir estas relaciones conocemos los caminos o conexiones de los mensajes, es decir, que objetos pueden ejecutar que métodos de qué objetos. Existe otro tipo de objetos llamados objetos de datos los cuales únicamente encapsulan datos. Dentro de un objeto de datos no existen muchas operaciones por lo tanto pueden ser conceptualizadas o representadas como una tabla.

Esta metodología consiste en tres pasos fundamentales; primero que el diseñador defina claramente el problema, posteriormente que defina una estrategia en un lenguaje informal o coloquial para resolver el problema y formalice la estrategia, identificando los objetos de datos que pueden contribuir con la solución, las operaciones que se podrán aplicar a estos objetos, especificando las interfaces entre los objetos y proporcionando detalles de la implementación y operaciones o procedimientos. Estos pasos se realizan en dos etapas fundamentales que son el Análisis Orientado a objetos (OOA) y el Diseño Orientado a Objetos (OOD). El papel del OOD es tomar las clases y los objetos básicos definidos en el OOA y refinarlos con los detalles adicionales de diseño e implantación.

Los pasos que sigue esta metodología son básicamente los siguientes:

1.- Definir el problema.
2.- Desarrollar una estrategia informal (narrativa de procesamiento) para realizar el software, haciendo abstracción del problema dentro del mundo real.
3.- Formalizar la estrategia de solución mediante los siguientes subpasos:
  a. Identificar los objetos y sus atributos, sean heredados de las clases o subclases o propios del objeto. Es de suma importancia leer la narrativa informal para poder identificar los atributos del objeto que en gran manera lo definen, por ejemplo si hablaremos de un sistema de estadísticas de football,

los atributos de un objeto que podría ser **Jugador** serían el número de partidos que ha jugado, durante que tiempo ha metido más goles, que rendimiento en número de pases tiene durante el primer tiempo y cual durante el segundo tiempo, etc.

b. Identificar las operaciones o métodos que se les puede aplicar a los objetos. Una operación o método cambia un objeto de alguna manera, más concretamente cambia o modifica el valor de uno o más atributos del objeto, de tal forma que las operaciones permiten modificar las estructuras de datos definidas para el objeto.

c. Establecer interfaces que muestren las relaciones entre los objetos y las operaciones.

d. Decidir aspectos del diseño detallado que proporcionen una descripción de la implementación de los objetos.

4.- Volver a aplicar los pasos 2, 3 y 4 recursivamente.

5.- Refinar el trabajo realizado en el Análisis orientado a objetos (actividad de clasificación donde se analiza un problema con el fin de determinar clases objetos que sean aplicables en el desarrollo de la solución, buscando las subclases, las características de los mensajes y otros detalles de elaboración).

6.- Representar la (s) estructura (s) de datos asociada (s) con los atributos del objeto.

7.- Representar los detalles procedimentales asociados con cada operación [PRE93].

En resumen el diseño orientado a objetos crea un modelo del mundo real que puede ser realizado en software. Los objetos proporcionan un mecanismo para representar el ámbito de información, mientras que las operaciones describen el procesamiento asociado con el ámbito de información. Los mensajes (un mecanismo de interfaz) proporcionan el medio por el que se invocan las operaciones. La característica distintiva de esta metodología es que los objetos saben qué operaciones se pueden aplicar sobre ellos. Este conocimiento se consigue combinando abstracciones de datos y de procedimientos en un solo componente de programa llamado objeto o paquete.

2.5 PROTOTYPING.

Con la invención del enfoque descendente antes discutido se fue desarrollando una variación de éste llamado enfoque de prototipos, con el cual se obtiene un resultado en forma rápida y fácil del modelo del sistema a construir, sea cual sea la metología utilizada, y poder así regular el desarrollo del sistema. Según fue popularizada por Bernard Boar: "la definición de un sistema se realiza mediante el descubrimiento evolutivo y gradual y no a través de la previsión omnisciente... Este tipo de enfoque se llama de prototipos. También se le conoce como modelado del sistema o desarrollo heurístico. Este desarrollo ofrece una alternativa atractiva y practicable a los métodos de especificación para tratar mejor la incertidumbre, la ambigüedad y la volubilidad de los proyectos reales" [YOU93].

La palabra Prototype en inglés es prototipo, y la acción es Prototyping, debemos hacer la distinción ya que no debe confundirse, en general el prototipo puede ser el modelo o

primer modelo de desarrollo del comportamiento de un sistema, mientras que el prototyping es la acción de crear el prototipo[GUZ89].

La fig. 2.5 muestra el ciclo de desarrollo de un prototipo presentando cada una de las fases. Es de suma importancia hacer notar que el uso de herramientas automatizadas de 4ª generación que permita la generación de prototipos en forma rápida y eficiente es fundamental para el buen desarrollo de un sistema, permitiendo ver al usuario como funciona su sistema y facilitar al programador la modificación o adición de especificaciones o requerimientos del sistemas sobre el ptototipo. Estas herramientas generan automáticamente código fuente basándose en la especificación anterior, además con el uso de herramientas CASE podemos generar y modificar en forma rápida las diferentes herramientas de análisis y diseño que se conocen como diagramas de flujo, de entidad-relación, etc.

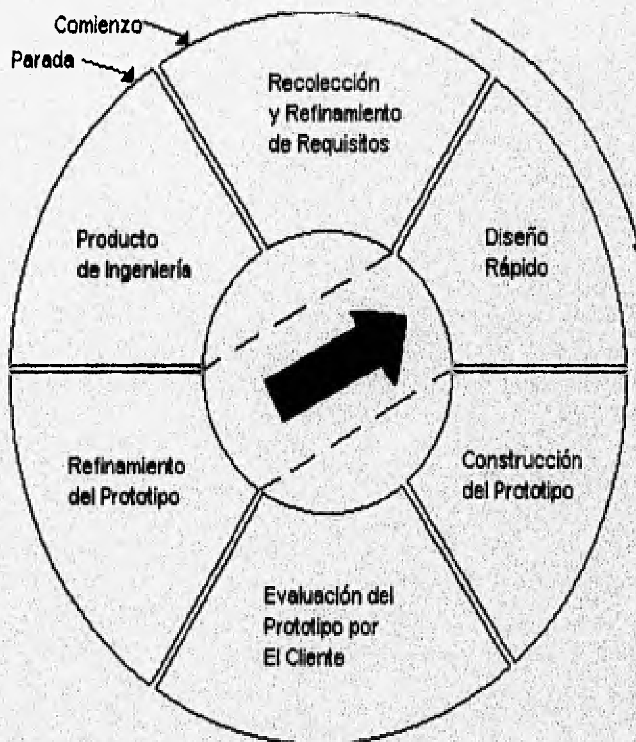


Fig. 2.5

En general el uso de prototipos se ha desarrollado como una herramienta de apoyo fenomenal en los últimos años, permitiendo crear sistemas en mucho menor tiempo, y generando todo un mercado de compra y venta. Además con el uso de la Metodología Orientada a Objetos que veremos a continuación se ha incrementado el poder de diseño y análisis.

2.6 SISTEMAS EN TIEMPO REAL.

Para definir un sistema en tiempo real tomaremos la siguiente definición:

"Un sistema computacional de tiempo real puede definirse como aquél que controla un ambiente recibiendo datos, procesándolos y devolviéndolos con la suficiente rapidez como para influir en dicho ambiente en ese momento. Además de la velocidad existe otra característica de los sistemas en tiempo real, éstos pueden tener un ambiente amigable con el usuario o ser sumamente autónomos y a veces hostiles, lo importante es que respondan en un tiempo aveces menor o igual a los milisegundos o microsegundos" [YOU93].

En un sistema de tiempo real es de suma importancia, cuando se esta haciendo el diseño, considerar el comportamiento dependiente que tenga el sistema.
Desde un punto de vista de su puesta en práctica, los sistemas de tiempo real se caracterizan por lo siguiente:

- Simultáneamente se lleva a cabo el proceso de muchas actividades.
- Se asignan prioridades diferentes a diferentes procesos: algunos requieren servicio inmediato mientras que otros pueden aplazarse por periodos razonables.
- Se interrumpe una tarea antes de concluirla, para comenzar una de mayor prioridad.
- Existe una gran intercomunicación entre tareas, ya que todas conforman y son parte fundamental de un proceso general.
- Se accede simultáneamente a datos comunes, tanto en memoria como en el almacenamiento secundario (bases de datos en tiempo real), por lo que se requiere de un elaborado proceso de sincronización y semáforos de colas que gestionan el tráfico o acceso a los recursos del sistema para asegurar que los datos comunes no se corrompan, también controlan buffers, que son como buzones donde se almacena la información que genera un proceso y los cual puede ser utilizados por otro proceso. Finalmente otro recurso para controlar la sincronización es un sistema de mensajes entre procesos, uno envía a otro un mensaje y éste último se activa automáticamente por el sistema de soporte de tiempo de ejecución o sistema operativo para que se procese el mensaje.
- Existe un uso y asignación dinámicos de memoria RAM, dado que no es posible por el aspecto económico, disponer de memoria fija para solucionar situaciones pico de alto volumen.
- Generalmente incluyen un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y un componente de monitorización que coordina todos los demás componentes, de forma que pueda mantenerse la respuesta en tiempo real (típicamente en el rango de 1 microsegundo a 1 minuto) [YOU93].

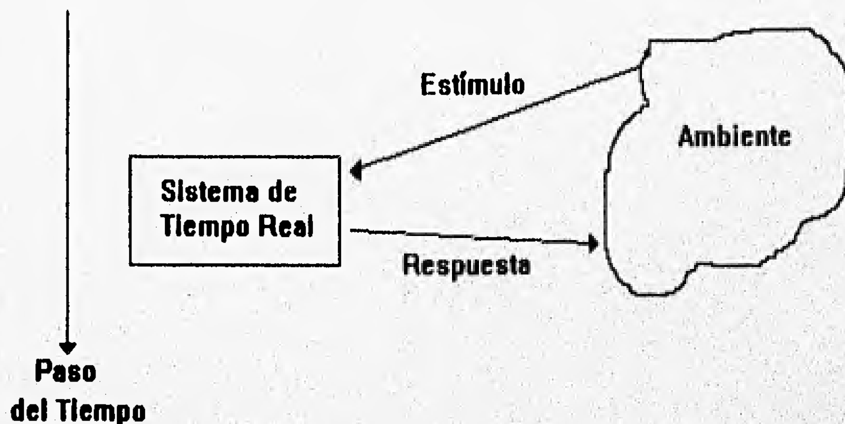


Fig. 2.6

Muchas de las características mencionadas antes son el trabajo de el sistema operativo de tiempo real, las bases de datos en tiempo real y la sincronización y conmutación de tareas que se utilizan en un sistema de tiempo real. Debido a que los sistemas en tiempo real operan bajo restricciones de rendimiento muy riguroso, el diseño de los sistemas computacionales de tiempo real se ve profundamente influenciado por la arquitectura del hardware, la del software, las características del sistema operativo, los requisitos de la aplicación, así como un tanto por los lenguajes de programación y los aspectos de diseño [YOU93].

El rendimiento de un sistema en tiempo real se determina principalmente por su tiempo de respuesta, es decir, el tiempo que tarda el sistema desde que detecta la excitación de un suceso externo o interno hasta generar la respuesta a esa excitación, y la razón de transferencia de datos. El proceso de los datos dentro del sistema para determinar o calcular la respuesta del sistema a la excitación, es lo que generalmente puede producir algoritmos complejos que consuman mucho tiempo, ya que la recepción y envío de la información hacia y del sistema suele ser relativamente rápida. Entre los parámetros que afectan al tiempo de respuesta están el cambio de contexto y la latencia de la interrupción. El cambio de contexto no es más que el tiempo y overhead necesitados para conmutar entre tareas, es decir el tiempo que el sistema operativo se toma para almacenar el estado de la computadora y los contenidos de los registros, de forma que pueda volver a la tarea de procesamiento después de servir a una interrupción, la cuál se tratará más adelante, y la latencia de la interrupción es el tiempo máximo que transcurre antes de que el sistema pueda conmutar a una tarea.

Otros parámetros que afectan el tiempo de respuesta son el acceso a memoria masiva y la velocidad de cáculo. La razón de transferencia de datos es la rapidez de la entrada y salida de datos del sistema tanto analógicos como digitales. El rendimiento del dispositivo de E/S, la latencia del bus, el tamaño del buffer, el rendimiento del disco y otros factores, aunque importantes, son sólo parte de la historia del diseño de un sistema en tiempo real [YOU93].

A diferencia de los sistemas interactivos, los cuales también deben de tener tiempos de respuesta óptimos, la fiabilidad, reinicialización y corrección de fallos, no es tan demandante como en los sistemas de tiempo real, ya que éstos están monitoreando constantemente el mundo real y se requieren sistemas muy completos de recuperación de fallas, reinicialización y restauración para asegurar fiabilidad.

2.6.1 Manejo de interrupciones.

Otro aspecto muy importante dentro de los sistemas de tiempo real es el manejo apropiado y jerarquizado de las interrupciones, las cuales son estímulos externos en un tiempo dictado por el mundo externo. Este manejo es jerarquizado porque las interrupciones tienen prioridades utilizadas para determinar si se da servicio a la solicitud de una interrupción y parar el proceso en acción. Es importante tener un control sobre las interrupciones para evitar deadlock y loops sin fin. Un enfoque global de una interrupción se muestra en la fig. 2.7.

En muchas situaciones el servicio a una interrupción puede ser interrumpido a su vez por otro servicio a una interrupción de mayor jerarquía, así que es importante llevar un control, para poder ir regresando las tareas hastá continuar con el flujo normal de procesamiento, si por alguna razón se diera paso a una interrupción de menor jerarquía sobre una de mayor jerarquía se perdería la secuencia de restauración del flujo normal de procesamiento y se caería en un loop sin fin.

Otra característica importante en el diseño de sistemas en tiempo real es el uso de bases de datos en tiempo real, esto es, bases de datos distribuidas las cuáles son ventajosas por la forma de acceder a los datos, ya que se evitan colas de espera para la obtención de éstos y es más fácil acceder la información desde diferentes partes. La caída de un base de datos puede no afectar al proceso en tiempo real si se cuenta con procesos eficientes de redundancia, pero se genera otro problema, el control de concurrencia de los datos y su integridad los cuales deben controlarse para que no exista incongruencia al escribir o leer información. Actualmente se cuenta con procesos novedosos para salvar éste problema como son los protocolos de escritor exclusivo [PRE93].

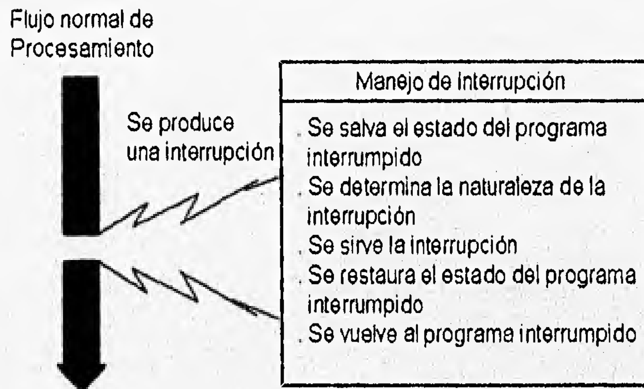


Fig. 2.7

2.6.2 Sistema operativo de tiempo real.

Los sistemas operativos para tiempo real (SOTR) tienen ventajas y desventajas con respecto a los sistemas operativos de propósito general, de ahí que sea difícil elegir uno. Podemos catalogar a los SOTR de dos formas: (1) sistemas operativos específicos para tiempo real y (2) sistemas operativos de propósito general que se han reforzado para suministrar capacidades de tiempo real. Tanto el diseño exclusivo como el refuerzo de capacidades deben contemplar un mecanismo de planificación de prioridades sobre las interrupciones y un mecanismo de bloqueo de memoria para mantener un mínimo de programas en memoria y así evitar el overhead. Como las interrupciones ocurren en respuesta a sucesos asíncronos no concurrentes, éstas deben atenderse sin consumir tiempo de carga de un programa. Una manera de evaluar un sistema operativo de tiempo real es evaluar su cambio de contexto y su latencia de la interrupción [PRE93].

2.6.3 Lenguajes en tiempo real.

Varias características hacen a un lenguaje de tiempo real diferente de un lenguaje de propósito general, estas incluyen la capacidades multitarea, construcciones para implementación directa de funciones de tiempo real y características modernas de programación que ayuden a asegurar la corrección del programa. Ejemplos de lenguajes específicos de tiempo real son: ADA, Jovial, HAL/S, Chill y otros que se utilizan frecuentemente en aplicaciones especiales de comunicaciones y militares [PRE93].

2.6.4 Análisis, Modelado y Simulación de Sistemas en Tiempo Real.

Esta es una tarea sumamente importante ya que de ella depende el buen funcionamiento del sistema. Existen herramientas matemáticas de análisis como el método de Thomas McCabe que proporciona herramientas de modelización de elementos de software y hardware que representan el control de una forma probabilística aplicando el análisis de redes, teoría de colas y grafos y un modelo matemático marcoviano para derivar el tiempo del sistema y el tamaño del recurso [MCC85].

Existen también herramientas de modelización y simulación como el método Statemate el cual permite construir un modelo completo del sistema que sea lo suficientemente preciso como para fiarse de el y lo suficientemente claro para que sea útil. El modelo que genera el método Statemate trata los conceptos habituales del flujo de información de los procesos y la función que realiza cada proceso, pero también contempla la dinámica y aspectos de comportamiento del sistema. Este modelo se puede probar mediante las herramientas de recuperación y análisis de Statemate, porque proporcionan varios mecanismos para inspeccionar y depurar la especificación y la recuperación de la información a partir de ella. Mediante la prueba del modelo se puede ver como se comportará el sistema especificado cuando se implemente [HAR90]. El método utiliza tres planteamientos diferentes del sistema: *el diagrama de actividades* en el cual se plantea el concepto de funcionalidad del sistema, es decir qué actividades se van a desarrollar y cuál es su función u objetivo, *el diagrama de estado* en el cual se plantea el comportamiento secuencial o concurrente del sistema, presentando los diferentes estados por los que puede pasar el sistema, el orden en el cual los va a pasar y las transiciones de uno a otro, por tanto son diagramas de control de las actividades a realizar, y *el diagrama de módulos* que es la representación física del sistema, es decir la descomposición por módulos, entiéndase módulos como componentes ya sea de hardware o de software o una mezcla de ambos y su relación con el modelo conceptual. Estos módulos se conectan con líneas que representan flujos de información entre éstos.

Una vez modelados los diferentes diagramas con los que cuenta Statemate, se procede a una revisión sintáctica y semántica de los modelos y sus relaciones, análogamente como la comprobación realizada por un compilador antes de la verdadera compilación de un lenguaje de programación. Posteriormente se puede utilizar el lenguaje llamado Simulation Control Language (SCL) del Statemate, el cual fue creado para realizar la simulación extensa de un modelo de tiempo real, éste permite retener el control general de como se desarrollan las ejecuciones, pero al mismo tiempo explotar la potencia de la herramienta para encargarse de muchos detalles. Alternativamente se puede programar con SCL para establecer puntos de ruptura y controlar ciertas variables, estados o condiciones para una revisión o debugger del programa mas detalladamente.

Realizada la simulación, se puede traducir automáticamente a código ejecutable mediante una función de creación de prototipos. Los diagramas de actividades y sus diagramas de estado, que ejercen el control, se pueden traducir a un lenguaje de programación de alto nivel como ADA o C. EL código producido por tales herramientas CASE como la función de creación de prototipos de Statemate, permite probar el funcionamiento del sistema en circunstancias cercanas a las del mundo real, pero no siempre reflejar el rendimiento preciso del sistema en tiempo real, ya que debe considerarse solo como un prototipo llevado a código, pero no código final o de producción.

El modelo de Statemate es solo una herramienta CASE que nos facilita la tarea de análisis y modelado, claro que constantemente se estan desarrollando nuevas herramientas de software para poder facilitar éstas tareas. Lo que es sumamente importante es recordar que si ésta actividad se desarrolla con el debido cuidado y tratando de pensar en todos los posibles problemas que se pueden presentar, se logra generalmente un funcionamiento sumamente óptimo del sistema.

CONCLUSIONES.

Hemos analizado en forma muy breve algunas de las diferentes metodologías que existen para desarrollar software observando que todas ellas realizan las tareas de análisis y diseño. También debemos hacer énfasis que el prototyping es una herramienta de análisis, diseño y creación rápida que nos ahorra tiempo muy valioso. El mejor sistema de software es aquél que logra englobar todos los casos particulares que se pudieran presentar antes de automatizar.

Podemos rescatar también que la metodología orientada a objetos requiere de mayor abstracción en los procesos de analisis y diseño, pero una vez realizado un buen trabajo en la definición de objetos y procedimientos, la programación resulta ser muchísimo más fácil que las otras metodologías. Durante el desarrollo de un sistema de software, se utiliza la metodología que más se acople al sistema que se está realizando. Si es un sistema muy pequeño y muy sencillo la metodología clásica resulta ser la más práctica y conveniente, pero si el sistema es verdaderamente grande y complejo, disponemos de las demás metodologías para atacarlo, como la estructurada o la orientada a objetos.

CONCLUSIONES.

Hemos analizado en forma muy breve algunas de las diferentes metodologías que existen para desarrollar software observando que todas ellas realizan las tareas de análisis y diseño. También debemos hacer énfasis que el prototyping es una herramienta de análisis, diseño y creación rápida que nos ahorra tiempo muy valioso. El mejor sistema de software es aquél que logra englobar todos los casos particulares que se pudieran presentar antes de automatizar.

Podemos rescatar también que la metodología orientada a objetos requiere de mayor abstracción en los procesos de analisis y diseño, pero una vez realizado un buen trabajo en la definición de objetos y procedimientos, la programación resulta ser muchísimo más fácil que las otras metodologías. Durante el desarrollo de un sistema de software, se utiliza la metodología que más se acople al sistema que se está realizando. Si es un sistema muy pequeño y muy sencillo la metodología clásica resulta ser la más práctica y conveniente, pero si el sistema es verdaderamente grande y complejo, disponemos de las demás metodologías para atacarlo, como la estructurada o la orientada a objetos.

REFERENCIAS.

[GUZ89] Guzman Robles, Luz Maria y Villagran Araiza, Manuel Agustin. Tesis de Licenciatura: "El uso del prototyping como metodología en el desarrollo de sistemas y su aplicación sobre lenguajes de cuarta generación". UNAM, F.I., México 1989.

[HAR90] Harel, D. "STATEMATE: A Working Enviroment for the Development of Complex Reactive Systems", IEEE Transactions Software Engineering, vol. 16, no. 3, April 1990.

[MCC85] McCabe, T. J. et.al. "Structured Real-Time Analysis and Design", COMPSAC-85, IEEE, Octubre de 1985.

[PRE93] Pressman, Roger S. *Ingeniería del Software: un enfoque Práctico*. 3ª ed. McGraw-Hill, México, 1993.

[YOU93] Yourdon, Edward. *Análisis Estructurado Moderno*. Prentice-Hall Hispanoamérica, México, 1993.

3

Herramientas de Desarrollo de Sistemas DSP.

INTRODUCCION.

En este capítulo presentaremos una revisión de las diferentes herramientas de software existentes para el diseño de sistemas DSP. En primer lugar hablaremos de los distintos tipos de herramientas. Posteriormente daremos una breve descripción de algunas de las más importantes, mencionando sus características, sus capacidades, sus alcances, la plataforma sobre la que trabajan y sus requerimientos de hardware y software. Al final del capítulo contaremos con una visión clara de estas herramientas y la aplicación que pueden tener en los distintos pasos de una metodologia general de diseño de sistemas DSP.

3.1 CLASIFICACION DE HERRAMIENTAS.

3.1.1 Herramientas de Diseño de Circuitos Integrados VLSI.

Uno de los distintos enfoques de diseño de sistemas DSP tiene que ver con la creación de nuevos circuitos integrados a muy grande escala (Very Large Scale of Integration), tanto de procesadores de señales generales como circuitos integrados de aplicación específica o ASIC (Application Specific Integrated Circuit). Se han desarrollado herramientas de software, que en el mundo de los chips se llaman herramientas CAD (Computer Aided Design: Diseño Asistido por Computadora) para automatizar los distintos pasos del diseño de este tipo de hardware. Por ejemplo, existen lenguajes que normalizan la representación de algoritmos DSP, programas que sintetizan circuitos integrados a partir de diagramas de bloques de una aplicación DSP, sistemas de simulación y parametrización de circuitos integrados, programas que optimizan el espacio y la potencia de una arquitectura, etc. Las herramientas más avanzadas, que son sobre las que abundaremos en este trabajo, son aquéllas que integran varias de estas aplicaciones.

3.1.2 Herramientas de Simulación de Sistemas DSP.

Existen también herramientas para la simulación de sistemas DSP que tienen distintos alcances. Los ejemplos más simples son aquellos simuladores de los procesadores de señales de amplio uso, como los de la familia TMS320 de Texas Instruments o el DSP5600 de Motorola. Existen, sin embargo, ambientes gráficos de más alto nivel que permiten analizar y diseñar sistemas DSP a partir de su representación en diagramas de bloques o de flujo se señales. Estos ambientes generalmente son programables, de forma que el usuario pueda generar sus propios algoritmos, y además permiten el estudio de sistemas muy complejos. Ejemplos de estas herramientas son Khoros [ZEP90] y Mathematica [EVA93].

3.1.3 Herramientas para Sistemas Heterogéneos y Desarrollo de Prototipos.

Son las herramientas más avanzadas hasta el momento. Permiten analizar y diseñar sistemas heterogéneos a partir de una programación gráfica orientada a objetos. La parte gráfica está basada en los lenguajes de diagramas de bloques y de flujo de señales que permiten trabajar con sistemas DSP, de comunicaciones y de otros tipos en una forma más clara y más relacionada con el análisis y diseño tradicionales. Aparte de la simulación, estas herramientas pueden sintetizar código para un procesador específico a partir de la representación gráfica del sistema, y pueden además sintetizar una arquitectura de hardware óptima que desempeñe la aplicación representada gráficamente. Asimismo, pueden llevar a cabo el control de la arquitectura en una aplicación en tiempo real. A este concepto se le conoce como *Prototyping*. Ejemplo de este tipo de herramientas son Ptolemy [BUC94], desarrollado por el grupo de investigación en DSP del departamento de ingeniería eléctrica y ciencias de la computación de la Universidad de California en Berkeley, y Matlab, de la empresa Mathworks [MAT96].

3.2 DESCRIPCION DE HERRAMIENTAS DE DISEÑO DE CIRCUITOS INTEGRADOS VLSI.

3.2.1 Generalidades de las Herramientas.

Como mencionamos anteriormente, uno de los enfoques del DSP implica el desarrollo de nuevos circuitos integrados a muy gran escala. Estos circuitos pueden ser procesadores dedicados que desempeñen sólo una aplicación o bien procesadores generales que puedan microprogramarse para realizar distintos algoritmos DSP en una arquitectura general (y, por lo tanto, más compleja). Un ejemplo de los primeros sería un chip para comprimir imágenes en JPEG a instalarse en una tarjeta de video. Por su parte, los procesadores generales están destinados a tarjetas de aplicación general

(llamadas DSP cores, que además del chip DSP tienen convertidores A/D y D/A, memoria, buses, etc.). El diseño de un circuito integrado es un proceso muy complejo que integra varias etapas, como son:

- Diseño de la arquitectura básica
- Simulación de la arquitectura diseñada
- Estimación de los parámetros de desempeño (velocidad, tasas de transferencia de datos, etc.)
- Estimación de los parámetros de construcción (tamaño del circuito, accesibilidad de conexiones, consumos de energía, disipación de calor, etc.)
- Diseño de la configuración (layout) de la impresión de los componentes y dispositivos en el silicio.

Todos estos pasos han significado que el proceso tras el cuál se tenga un chip funcionando tome demasiado tiempo, consuma demasiados recursos y sea susceptible a imprecisiones en la transición de un paso a otro. Con el fin de incrementar la calidad y la productividad en el desarrollo de circuitos integrados, se han creado diversas herramientas para automatizar e integrar las actividades involucradas en el proceso. A continuación presentamos las características de dos de ellas.

3.2.2 Pyramid.

PYRAMID [HUI88] es un ambiente interactivo de diseño automatizado de circuitos integrados para aplicaciones DSP. Consta de tres partes (o subambientes, como los nombran en la referencia original): el Ambiente de Síntesis, el Ambiente de Configuración y el Ambiente de Generación de Módulos.

- *Ambiente de Síntesis*: Permite sintetizar una arquitectura de procesamiento a través de la traducción de la descripción del comportamiento de un sistema en alto nivel. Esta descripción se realiza en el lenguaje Silage [HIL85], lenguaje que de forma estructurada, con una sintaxis y una semántica definidas, sirve para representar algoritmos DSP. El Ambiente, llamado *Cathedral II*, consta de dos programas: *Jack* y *Atomics*. El primero traduce la descripción en Silage en operaciones primitivas de transferencia de registros y genera una representación gráfica de las conexiones de datos, de forma que pueden identificarse cuellos de botellas y modificarse la arquitectura en forma interactiva. Una vez definidas las primitivas, *Atomics* se encarga de la planeación (scheduling) temporal de la arquitectura, asignando transiciones de estados y los ciclos de reloj necesarios para desempeñar el algoritmo. Este ambiente trabaja en paralelo con el Ambiente de Generación de Módulos, cuyas funciones veremos más adelante.

- *Ambiente de Configuración*: Este ambiente toma la arquitectura generada para crear representaciones tipo caja negra de los módulos de la misma, de forma que se

obtenga un diagrama más general y de más alto nivel del hardware, diagrama sobre el que se diseñará la localización de los módulos y el direccionamiento global y detallado de las conexiones entre los mismos y de las conexiones de los módulos con las salidas y las fuentes de voltaje y tierra.

- *Ambiente de Generación de Módulos*: Genera vistas del área que ocuparía el circuito impreso, de la configuración de los componentes (layout), de los aspectos de temporización del circuito, así como pruebas de funcionalidad.

Esta herramienta permite disminuir considerablemente el tiempo de diseño de circuitos integrados para DSP integrando estos tres ambientes. Cabe destacar que su enfoque es lo que podríamos llamar automatización interactiva, es decir, el usuario debe estar al tanto de los módulos y los diseños generados por los distintos ambientes para definir cambios en los mismos cuando identifique una falla o una posibilidad de mejora. Ulteriores desarrollos deberán estar orientados a la optimización de la síntesis de arquitecturas, haciendo innecesaria la presencia del usuario.

3.2.3 VHSIC Silicon Compiler (VSC).

VSC es un conjunto de herramientas de análisis y simulación de arquitecturas dedicadas (ASIC: Application Specific Integrated Circuit) [FRA88]. Fue desarrollado por un consorcio contratado por la fuerza aérea de los Estados Unidos e integrado por Intermetrics, IBM y Texas Instruments [MIL94]. Permite al diseñador de circuitos integrados evaluar el desempeño de un ASIC propuesto en el comportamiento global de un sistema, desarrollando una especificación ejecutable del ASIC, utilizando la especificación en el análisis y la simulación del sistema y, finalmente, proporcionando información de estos procesos para el diseño detallado del ASIC, lo que también soporta VSC.

Los objetivos de VSC, según explica la fuente original, son:

- *la detección temprana de errores de diseño*
- *la validación del diseño*
- *la documentación del diseño*
- *la reutilización de los diseños*
- *el almacenamiento de los diseños para futura referencia*

VSC consta de tres herramientas básicas:

- *El Sistema de Diseño y Evaluación de la Arquitectura (ADAS: Architecture Design and Assesment System)*: Proporciona el modelado funcional y de desempeño del software y del hardware. Tiene una interfaz gráfica para construir los algoritmos y la descripción de la arquitectura utilizando elementos genéricos de una biblioteca. Incluye además un simulador de desempeño, donde los algoritmos (software) se mapean y se simulan en la arquitectura (hardware) del sistema. El software se representa mediante

diagramas de flujo, mientras el hardware se representa con modelos jerárquicos estructurados. ADAS permite también evaluar parámetros clave en el desempeño del hardware, tales como tamaño del chip, número de interconexiones y consumo de potencia.

- *El Lenguaje VHSIC de Descripción del Hardware (VHDL: VHSIC Hardware Description Language)*: Proporciona los modelos de comportamiento del hardware dentro del sistema, modelos que serán utilizados para evaluar su desempeño y funcionalidad. El hardware puede ser simulado y validado utilizando el conjunto de herramientas del VHDL, que son un analizador de lenguaje, un analizador inverso, un simulador y un administrador de la biblioteca de diseño. Esta biblioteca es el archivo primario de VSC, puesto que todos los diseños de circuitos integrados y las especificaciones de ADAS pueden ser convertidos a formato VHDL y almacenados en la librería para ser utilizados posteriormente.

- *El Sistema del Compilador de Silicio GENESIL*: Es un sistema de diseño de circuitos integrados que soporta las especificaciones militares del ejército de Estados Unidos, que son las más estrictas y demandantes. Incluye un conjunto de funciones, distintos bloques funcionales personalizables, RAM, ROM, PLA, colas FIFO y lógica aleatoria. El diseñador especifica las funciones requeridas, sus parámetros y sus interconexiones. A partir de ello, GENESIL sintetiza un modelo del chip, modelo que puede ser evaluado por el diseñador, después de lo cual GENESIL genera una especificación detallada del chip para comenzar su construcción.

La parte más importante de esta herramienta es el lenguaje VHDL, que evolucionó hasta un estándar IEEE en 1988, y que puede ser una herramienta en sí misma, pero además puede ser la base para la creación de herramientas automatizadas más complejas, gracias al amplio uso y documentación con que de el lenguaje se cuenta actualmente, [MIL94].

3.3 DESCRIPCION DE HERRAMIENTAS DE SIMULACION DE SISTEMAS DSP.

3.3.1 Khoros.

Khoros es un ambiente de análisis y simulación de sistemas desarrollado por la Universidad de Nuevo México. Aunque ahora se ha constituido una empresa, inicialmente fue de dominio público, corre en plataforma UNIX y está compuesto por varias capas o subsistemas que interactúan para conformar la totalidad del sistema. Tiene dos interfaces o ambientes básicos: el de línea de comando y el lenguaje de programación visual (Cantata). Incluye 260 rutinas preprogramadas, las cuáles están almacenadas en las siguientes bibliotecas [ZEP95]:

- Procesamiento de Imágenes
- Procesamiento de Señales
- Reconocimiento de Patrones
- Percepción Remota
- Visión Artificial
- Sistemas de Información Geográfica

Interfaz de línea de comando.- En el ambiente o interfaz de línea de comando, cada rutina puede ser invocada como un programa de UNIX, estando Khoros activo. De esta forma, para ejecutar la rutina que comprime una imagen por un factor de 2 debe teclearse:

```
% vshrink -i KHOROS_HOME/data/images/ball.xv -o archivo_salida -s 2
```

donde los parámetros -i y -o definen los archivos de entrada y salida de la rutina, respectivamente, mientras que -s define el factor de compresión del algoritmo.

Interfaz de lenguaje de programación visual.- Se llama Cantata y es el ambiente más usado de Khoros, pues es una interfaz de usuario gráfica (GUI) donde pueden construírse análisis y simulaciones a partir de la representación gráfica de los algoritmos o rutinas de las bibliotecas de Khoros y de rutinas creadas por el usuario.

Los sistemas se representan generalmente en forma de diagramas de bloques o gráficas de flujo de datos; la utilización de Cantata es muy parecida a estos métodos, haciéndolo un ambiente muy útil y poderoso.

El proceso de creación de un programa visual es el siguiente:

- Se comienza abriendo una forma o espacio de trabajo en blanco.

- Se seleccionan de las bibliotecas adecuadas las tareas de interés. Cuando se selecciona una tarea, aparece una caja de diálogo que permite definir sus parámetros.

- Una vez definidos los parámetros, Cantata crea un elemento gráfico llamado glyph que representa la tarea seleccionada y que es equivalente a un programa individual de Khoros. Los glyphs de la biblioteca de Procesamiento de Señales incluyen funciones como generadores de señales, sumadores, elementos de retardo, correlaciones, autocorrelaciones, transformadas de Fourier, graficadores, etc. Cada glyph tiene los siguientes elementos: por lo menos un puerto o flecha de entrada; un puerto o flecha de salida; un botón de cancelación, que elimina al glyph del espacio de trabajo; un botón de configuración, que permite redefinir los parámetros de la tarea; y, finalmente, un botón de ejecución (switch on/off) que ejecuta la tarea del glyph.

- Una vez creados y configurados los glyphs, se conectan sus flechas de entrada y de salida conforme lo requiera el problema a analizar o simular. Por ejemplo, podemos simular el efecto del ruido sobre una señal senoidal. Para esta simulación requeriríamos un glyph para generar la señal senoidal, un glyph para generar la señal

de ruido, un glyph que sumara las dos señales anteriores (el cual tiene más de una flecha de entrada) y finalmente un glyph para graficar la salida del sumador. Este conjunto de glyphs interconectados es un programa visual de Khoros, que puede ser almacenado en un archivo ASCII para su inclusión en una biblioteca y posterior reutilización.

- Para ejecutar el programa visual, puede hacerse click sobre el botón "RUN" de la forma, o bien hacerse click secuencialmente sobre el botón de ejecución de cada glyph hasta completar todo el programa.

Programación de Khoros.- El usuario puede crear sus propias rutinas de Khoros haciendo uso de las herramientas que el ambiente proporciona para tal efecto. Existen dos tipos de rutinas: vroutines y xvroutines. Las primeras son las rutinas estándares de Khoros, que pueden ser ejecutadas desde la línea de comando de UNIX y que tienen una interfaz gráfica cuando son ejecutadas desde Cantata, mientras que las xvroutines son programas que al ejecutarse desde la línea de comando presentan una interfaz gráfica, sin necesidad de operar bajo Cantata. El procedimiento de creación de una vroutine se resume a continuación [ZEP95]:

- Creación de un archivo de Especificación de la Interfaz de Usuario (User Interface Specification), donde se incluirán todas las especificaciones del programa para las Interfaces de Línea de Comando y Gráfica, asegurando la consistencia de los parámetros de entrada y salida.

- Generación de código a través de las rutinas ghost: estas rutinas permiten generar código automáticamente a través de una plantilla de la Especificación del Programa (PS), de forma que la nueva rutina sea adecuada al estándar Khoros y tenga la estructura y la documentación adecuadas.

- Escritura de la rutina: A partir de la plantilla se corrige el archivo PS insertando el código fuente y la documentación creadas en el paso anterior.

- Instalación del programa: el programa kinstall permite integrar el nuevo programa a Cantata.

Como podemos observar a partir de esta breve descripción, Khoros es un ambiente muy completo de análisis y simulación de distintos tipos de sistemas y es particularmente poderoso en Procesamiento de Imágenes y de Señales. Tiene una Interfaz de Programación Visual que permite analizar y simular sistemas complejos que incluyan rutinas estándar de una biblioteca, y rutinas creadas por el usuario de una forma amigable, fácil de usar y altamente modular y automatizada.

3.3.2 Mathematica.

Mathematica [WOL88] es uno de los llamados "Sistemas de Algebra por Computadora", que permiten resolver problemas de ciencias básicas e ingeniería en forma numérica y simbólica. Estos sistemas son particularmente útiles en la manipulación de fórmulas con valores complejos, la factorización de polinomios y la expansión en fracciones parciales. Sus capacidades algebraicas les permiten realizar análisis de nodos y de mallas en redes resistivas, mientras el manejo de números complejos les permite calcular fasores en circuitos de corriente alterna y los métodos de solución de sistemas de ecuaciones no lineales facilitan el análisis y diseño de circuitos activos, [EVA93].

Mathematica consiste de dos programas principales: el kernel y la interfaz de usuario. El kernel proporciona las funciones de cálculo, las reglas condicionales, una aritmética de precisión arbitraria e incluso un lenguaje de programación. Las funciones, operadores y constantes del sistema se utilizan mediante nombres que comienzan con una letra mayúscula. Por ejemplo, $\int \cos(x)dx$ es para Mathematica: `Integrate[Cos[x],x]`, que devuelve `Sin[x]`. La interfaz de usuario se llama Notebook, es gráfica y para la plataforma PC está basada en Windows.

A pesar de la potencia inherente de Mathematica, la simulación y análisis de sistemas lineales requiere de más y mayores capacidades. Son necesarios, por ejemplo, nuevos operadores como el retraso y la convolución; nuevas señales como el impulso y el escalón; nuevas operaciones simbólicas como las transformadas, y nuevas opciones de graficación como diagramas de polos y ceros y de respuesta en frecuencia [EVA93]. Para solventar estas necesidades, el Tecnológico de Georgia desarrolló los Paquetes de Procesamiento de Señales (SPP: Signal Processing Packages) [EVA90] como extensión de Mathematica.

Los SPP definen señales como el pulso discreto y continuo (`Pulse` y `CPulse`), el impulso (`Impulse`) y el escalón discreto y continuo (`Step` y `CStep`), entre otras. Se han implantado, además, herramientas simbólicas como la convolución, las transformadas Z y de Fourier que son básicas en el análisis de sistemas y señales, y para las que se han incluido funciones de evaluación de polos y ceros y de la región de convergencia. Asimismo, se implantaron nuevos formatos de graficación para evaluar el lugar geométrico de las raíces, para obtener diagramas de polos y ceros y distintas formas de respuesta en frecuencia, como los diagramas de Bode y de Nichols. A partir de estas habilidades de graficación y de las transformadas simbólicas, se implantaron dos funciones (ASPAnalyze y DSPAnalyze) que llevan a cabo el análisis general de señales de una y dos dimensiones. Este análisis incluye información textual y gráfica, como la transformada de Fourier, la transformada Z (o de Laplace, según sea el caso), los criterios de estabilidad, la gráfica de la respuesta en el dominio del tiempo, en el dominio de la frecuencia y el diagrama de polos y ceros, [EVA93].

Finalmente, haciendo uso de la interfaz gráfica de usuario, el mismo personal del Tecnológico de Georgia ha desarrollado un conjunto de tutoriales que permiten al

usuario conocer, a partir de animaciones interactivas, ejemplos de diseño de filtros y de obtención de las transformadas Z y de Fourier, entre otros, [EVA93].

3.4 DESCRIPCION DE HERRAMIENTAS DE SISTEMAS HETEROGENEOS Y DESARROLLO DE PROTOTIPOS.

3.4.1 Ptolemy.

Ptolemy es una herramienta de simulación y desarrollo de prototipos (prototyping) de sistemas heterogéneos. Fue desarrollado por el Departamento de Ingeniería Eléctrica y Ciencias de la Computación de la Universidad de California en Berkeley a partir de su antecesor Gabriel y como resultado de un extenso programa de desarrollo de lenguajes, herramientas y metodologías automatizadas de diseño. Está basado en la programación orientada a objetos (C++), utilizando las características de ésta para modelar cada subsistema de una forma natural y eficiente, así como lograr su integración en un sistema global. Incluye prácticamente todos los aspectos del diseño de sistemas de procesamiento de señales y de sistemas de comunicaciones, aspectos que van desde los algoritmos y estrategias de comunicaciones, simulación, diseño y síntesis automática de hardware y software, computación paralela y generación de prototipos en tiempo real [BUC94].

Para poder soportar todos los modos anteriores la herramienta debe ser muy general, lo que se logra definiendo objetos llamados dominios, que son modelos amplios con características muy generales que permitan su interacción, pero de los cuáles, haciendo uso de la OOP (Object Oriented Programming), puedan derivarse objetos más específicos para realizar las distintas tareas y actividades de los diferentes subsistemas. Existen así dominios de flujo de datos estático y dinámico, de eventos discretos y otros específicos para software de control y microcontroladores interconstruídos. Ptolemy es ideal para aquellas aplicaciones heterogéneas, como [BUC94]:

- diseño de redes multimedia
- software interconstruído de tiempo real
- diseño paralelo de software y hardware
- control y procesamiento de llamadas en redes de telecomunicaciones
- desarrollo rápido de prototipos de servicios de telecomunicaciones
- simulación de hardware de modos múltiples (mixed-mode)
- sistemas de procesamiento de señales y control

Anteriormente se han desarrollado lenguajes y herramientas para sistemas heterogéneos, como el sistema STATEMATE de i-Logix que es un ambiente de simulación de circuitos VLSI complejos, Granular Lucid y Linda, que son lenguajes de coordinación. Pero todos ellos asumen una serie de reglas semánticas predefinidas, lo

que limita los modelos que puedan manejar y hacen casi imposible la incorporación de nuevos modelos.

Cuando hablamos de modelos nos referimos a diagramas de flujo, diagramas de flujo de datos, diagramas de estados, etc., que son los modelos con los que se diseñan y analizan distintos tipos de sistemas de procesamiento de señales, sistemas digitales, sistemas de control y software.

Se ha mencionado [BUC94] que la generalidad es una alternativa a la heterogeneidad, siendo VHDL un ejemplo del cual hemos hablado anteriormente en este capítulo. VHDL es un lenguaje muy amplio que puede acomodar distintos tipos de sistemas. Sin embargo, la generalidad implica la dificultad de contar con compiladores eficientes y del análisis de los sistemas definidos.

Ptolemy, en cambio, permite trabajar con sistemas heterogéneos sin limitar el diseño de los distintos subsistemas. Para ello cuenta con un kernel no dogmático que no presupone un modelo de flujo de datos o un modelo funcional. Por ello, Ptolemy, más que un lenguaje, es una herramienta de coordinación.

Los objetivos que Ptolemy pretende alcanzar mediante la Programación Orientada a Objetos son [BUC94]:

- *Agilidad*: soporte de distintos modelos

- *Heterogeneidad*: coexistencia de los distintos modelos

- *Extensibilidad*: permitir la integración de un nuevo modelo

- *Interfaz amigable*: utilizar una interfaz gráfica moderna basada en la representación de los sistemas como diagramas de bloques jerárquicos.

Estructura de Ptolemy.- El objeto básico de Ptolemy es el Bloque (Block), cuyo método go() se invoca cuando desea ejecutarse la función contenida en el bloque. Los distintos bloques se comunican a través de los Puertos (Portholes). De esta forma, cuando se ejecuta el método go() de un Bloque, se examinan los datos presentes en sus Puertos de entrada y se generan datos en sus Puertos de salida. Cuando varios bloques coexisten en una aplicación, el orden (o semántica) de su ejecución es coordinado por un Organizador (Scheduler).

Jerárquicamente, de los Bloques se derivan los objetos Estrella (Star), Galaxia (Galaxy) y Universo (Universe). El Universo es una aplicación completa, que contiene Galaxias, las cuáles a su vez pueden contener más Galaxias o Estrellas, que son los objetos atómicos de Ptolemy. De esta forma, un Universo es una Galaxia de tipo ejecutable (Runnable). El objeto Objetivo (Target) controla la ejecución de una aplicación, generalmente invocando a un Organizador (Scheduler), el cuál maneja el orden en que se ejecutan los métodos contenidos en las Estrellas.

Heterogeneidad: el Dominio.- Un Dominio es un conjunto de Bloques, Objetivos y Organizadores que conforman un modelo computacional, que es la semántica operacional que determina la forma en que los bloques interactúan entre sí [BUC94]. Los dominios existentes en Ptolemy son:

- Flujo de Datos Dinámico (DDF: Dynamic Data Flow)
- Flujo de Datos Síncrono (SDF: Synchronous Data Flow)
- Flujo de Datos Booleano (BDF: Boolean Data Flow)
- Evento Discreto (DE: Discrete Event)

Para lograr la heterogeneidad, Ptolemy soporta los siguientes conceptos: el Orificio de Gusano (Wormhole) y el Horizonte de Eventos (Eventhorizon). El primero es una abstracción donde dentro de un dominio se puede incluir otro dominio como un bloque normal, una estrella, por ejemplo, de forma que el modelo distinto permanezca encapsulado en el Orificio con sus propios Objetivos y Organizadores.

Para lograr la interoperabilidad de la estructura interna de un Orificio de Gusano con el dominio exterior en el que se ha insertado, se necesita de una interfaz. Esta interfaz es el Horizonte de Eventos. La interfaz es general para todos los dominios existentes en Ptolemy, y se encarga fundamentalmente de la conversión de los paquetes de datos, que son llamados Partículas (Particles).

Plataforma.- Ptolemy es de dominio público, fue desarrollado en C++ y estaciones de trabajo Sparc de Sun Microsystems, pero ha sido ya transportado a las estaciones DECstation de Digital y está por transportarse a plataforma HP, [BUC94].

La Interfaz Gráfica.- La interfaz gráfica de usuario de Ptolemy se llama pigi (Ptolemy Interactive Graphical Interface) y permite desarrollar aplicaciones de simulación, análisis y desarrollo de prototipos interconectando iconos de forma muy similar a como se realiza en Khoros. Pigi consta de dos procesos de UNIX: VEM, que es el editor gráfico y pigiRPC, que contiene el kernel de Ptolemy y la interfaz gráfica de este kernel. Para la comunicación de estos dos procesos, se utiliza el protocolo UNIX RPC. El usuario puede también crear e instalar sus propios bloques gráficos.

Se menciona, sin embargo, que la interfaz gráfica no puede soportar todos los distintos modelos de Ptolemy, y por ello se incluyen dos interfaces de modo texto para poder construir aplicaciones de simulación o prototyping más complejas: una es un lenguaje parecido al Lisp y la otra es un intérprete interconstruido, llamado Tcl.

3.4.2 Matlab.

Matlab es un producto comercial de la empresa Mathworks. Se trata una herramienta de cálculo y análisis numérico con un poderoso soporte de graficación y un lenguaje de programación de alto nivel. Es la base de una extensa plataforma de productos

destinados al análisis de algoritmos, desarrollo de prototipos y desarrollo de aplicaciones. Las características básicas de Matlas son:

- Análisis y manipulación de matrices.
- Análisis estadístico
- Análisis de Fourier, correlación y covarianza
- Funciones trigonométricas
- Ecuaciones diferenciales y lineales

Las capacidades de graficación incluyen gráficas de diversos tipos en dos y tres dimensiones, iluminación y sombreado de superficies, así como animación y sonido. El lenguaje de programación incluye estructuras de control (IF, FOR, WHILE) y manipulación de cadenas.

Matlab funciona a partir de la creación de archivos, llamados M-files, que invocan las distintas funciones incluidas en el software y las relacionan mediante el lenguaje de programación. Matlab está orientado al manejo de matrices, lo que implica que generalmente los datos y los parámetros de las funciones y de los procedimientos están en forma de vectores y matrices.

Cuenta además con una interfaz gráfica de usuario (GUI) que permite editar y rastrear y corregir los errores (debug) de los archivos M, así como manipular las gráficas generadas.

Existen una gran diversidad de productos que extienden Matlab, entre los cuales se cuentan:

- Un compilador que transforma los archivos M en código en C, aumentando la rapidez de ejecución de los programas.

- Cajas de herramientas (toolboxes) que incluyen funciones específicas para distintos campos, como procesamiento de señales, control, ingeniería financiera, lógica difusa, procesamiento de imágenes, redes neuronales, técnicas de optimación, ecuaciones diferenciales parciales, control robusto, matemáticas simbólicas, estadística, etc.

- Simulink, que es el ambiente gráfico y visual de modelado, simulación y desarrollo de prototipos. Permite la creación interactiva de programas visuales tomando elementos gráficos de bibliotecas de objetos. El esquema de utilización sigue los mismos lineamientos de las interfaces gráficas de Khoros y Ptolemy, mencionados anteriormente en este capítulo. Al igual que Ptolemy, permite el modelado de sistemas heterogéneos que incluyan distintos modos de cálculo u operación, como sistemas lineales y no lineales y sistemas en tiempo continuo y discreto. Simulink permite, además, la creación de simulaciones "en vivo", que son simulaciones donde el despliegue de resultados o salidas se realiza conforme se cambian los parámetros de los elementos, sin necesidad de recompilar y reejecutar el programa visual.

• Existen extensiones de Simulink, llamadas conjuntos de bloques (blocksets), que permiten incluir bloques y elementos gráficos propios de campos como el procesamiento digital de señales, las comunicaciones, etc.

• Para la aplicación de esta plataforma al procesamiento de señales, el toolbox y el blockset correspondientes incluyen funciones y elementos específicos del DSP, como son: convolución, correlación, retrasos, FFT, decimación, ventanas (windowing), manejo de números complejos, operaciones de matrices, diseño de filtros IIR y FIR, etc.

• Finalmente, el producto más avanzado de la plataforma lo constituye el Taller en tiempo real (Real-Time Workshop), que permite generar código automáticamente a partir de los programas visuales de Simulink. El código generado es optimizado y portátil, y en el caso de que la plataforma de desarrollo cuente con el hardware necesario, puede ser compilado y cargado en la arquitectura para el desarrollo rápido de prototipos y la simulación en tiempo real.

CONCLUSIONES.

Existen diversas herramientas automatizadas que hacen más rápido, fácil, versátil y poderoso el proceso de diseño de sistemas de procesamiento digital de señales. Dentro de los distintos alcances de este diseño, tenemos herramientas para el diseño de circuitos VLSI, para la simulación de algoritmos y sistemas mediante programación gráfica y, finalmente, tenemos las herramientas más avanzadas que permiten analizar, simular y construir prototipos de sistemas heterogéneos, sintetizando código y arquitecturas de hardware o circuitos VLSI a partir de la representación gráfica de un sistema.

Podemos concluir que estas herramientas conforman ambientes de desarrollo para realizar automáticamente una gran parte de las tareas que en el diseño tradicional consumían mucho tiempo, recursos y que generaban problemas en la integración con otros pasos del proceso de diseño. El papel y lápiz, las plantillas de dibujo, la calculadora, al análisis numérico y simbólico de los sistemas están siendo superados completamente gracias al desarrollo de estas herramientas.

REFERENCIAS.

[BUC94] Buck, Joseph et. al. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". Department of Electrical Engineering and Computer Science, University of California, Berkeley, Estados Unidos, 1994.

[EVA90] Evans, Brian, et. al. "Symbolic Z Transforms Using DSP Knowledge Bases", en *Proceedings ICASSP-90*. IEEE, Estados Unidos, 1990. pp. 1175-1179.

[EVA93] Evans, Brian, et. al. "Learning Signals and Systems with Mathematica". IEEE Transactions on Education. Vol. 36. No. 1. IEEE, Estados Unidos, 1993.

[FRA88] Frank, Geoffrey A. y McLin, David M. "Systems to Silicon Design and Assessment", en *VLSI Signal Processing III*, IEEE Press, Estados Unidos, 1988.

[HUI88] Huisken, J.A. et. al. "Design of DSP Systems on Silicon Using the Pyramid Library and Design Tools", en *VLSI Signal Processing III*, IEEE Press, Estados Unidos, 1988.

[MAT96] *Matlab*; Product Catalog. Mathworks, Estados Unidos, 1996.

[MIL94] Milne, George. *Formal Specification and Verification of Digital Systems*. McGraw-Hill, Reino Unido, 1994.

[ZEP95] Zepeda, Claudia. Tesis de licenciatura: *Sistema de Clasificación de Señales EEG Utilizando el Ambiente de Programación Visual Khoros*. Facultad de Ciencias Físico Matemáticas, BUAP, México, 1995.

[WOL88] Wolfram, S. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley, Estados Unidos, 1988.

4

Propuesta de una Metodología de Diseño de Sistemas DSP.

INTRODUCCION.

En este capítulo proponemos una metodología básica de diseño de sistemas de procesamiento digital de señales. Es importante mencionar que esta metodología puede también aplicarse en áreas como el control de procesos, las comunicaciones, la robótica, reconocimiento de patrones, etc. La metodología ha sido integrada a partir de metodologías existentes, sobre todo aquéllas utilizadas en el diseño de sistemas de información y en el diseño de arquitecturas de hardware, así como en cierta experiencia en el diseño de sistemas DSP. Haremos especial énfasis en la utilización de las herramientas de software como parte fundamental de varios de los pasos de la metodología. En primer lugar, se presentará el proceso general de la metodología propuesta, y posteriormente se detallarán el objetivo, las actividades, las entradas y salidas de cada fase de la misma.

4.1 PROPUESTA DE LA METODOLOGIA.

La metodología está definida en el diagrama de flujo de la figura 4.1. Consta de 5 fases:

1.- *PLANTEAMIENTO DEL PROBLEMA.*
2.- *ANALISIS DEL PROBLEMA.*
3.- *DISEÑO DE LA SOLUCION Y CONSTRUCCION DEL PROTOTIPO.*
4.- *PRUEBA Y REFINAMIENTO DEL PROTOTIPO.*
5.- *OPERACION.*

La primera fase consiste en plantear el problema a ser solucionado. Posteriormente, en la segunda fase o análisis se estudiará el problema y su entorno, de forma que al final de la etapa se tenga una especificación completa de los requerimientos de la solución, así como de sus modelos formal y funcional. A partir de esta especificación, en la fase de diseño se desarrollará la solución del problema y se creará a partir de ella un

prototipo que será evaluado (confirmado y verificado) y refinado (fase 4) hasta tener la solución final, que entrará en la fase 5: operación.

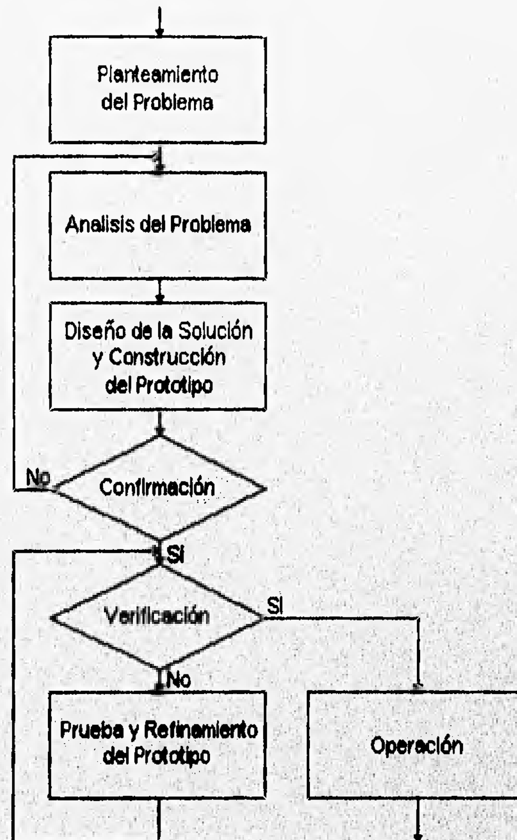


Fig. 4.1

Puede observarse en la figura 4.1 que se tienen dos diamantes de decisión, nombrados confirmación y verificación. Estos diamantes conforman el proceso de evaluación del prototipo, pero se han simplificado y dibujado como diamantes de decisión para dar la idea que del resultado que tengan dependerá qué fase de la metodología habrá que seguir. La confirmación consiste en realizar las actividades para responder a la pregunta "*¿Se está construyendo la solución correcta?*", mientras la verificación consiste en responder a la pregunta "*¿Se está construyendo correctamente la solución?*". Estas preguntas pueden parecer iguales entre sí, pero la diferencia existe y es muy importante. Si el diseño parte de una especificación correcta, se estará construyendo la solución correcta, si bien esa solución puede no ser óptima y requerir un mejor diseño, es decir, probablemente no se está construyendo correctamente esa solución; en este caso, la verificación arroja un resultado negativo y habrá que regresar a la fase de diseño. Pero una solución puede estar muy bien diseñada y no ser la solución correcta, es decir, puede no ser la solución adecuada al

problema; en este caso, habrá que regresar a la fase de análisis para especificar correctamente la solución.

4.2 PLANTEAMIENTO DEL PROBLEMA.

Es el paso inicial de la metodología, en el que se define el problema y la necesidad de su solución. De él parte la estructura de un buen diseño y una correcta implantación. En la figura 4.2 se observa el diagrama de esta fase.

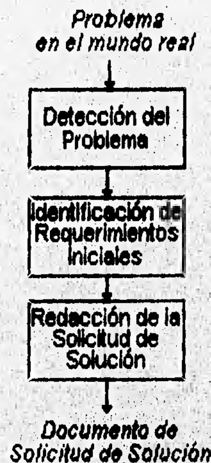


Fig. 4.2

4.2.1 Objetivo.

El objetivo de este paso de la metodología es plantear un problema y la necesidad de una solución para el mismo.

4.2.2 Entradas.

La entrada es un problema existente en el mundo real que será detectado, reconocido y definido en esta fase.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

4.2.3 Salidas.

La salida es un *Documento de Solicitud de Solución al Problema*, en donde deberán definirse el problema detectado, la necesidad de una solución y las características iniciales o primarias del problema y de la solución.

4.2.4 Actividades.

Las actividades implicadas en este paso son:

4.2.4.1 Detección del problema.

Para iniciar el paso y la metodología, es necesario que exista un problema y que éste sea debidamente detectado, es decir, debe percibirse la existencia del mismo, debe reconocerse y debe definirse claramente, lo que quiere decir que no se involucre con otros problemas distintos aunque interrelacionados, o que se identifique como un todo algo que no es sino una parte de un problema más amplio. Esta idea es muy importante, pues en muchas ocasiones un sistema DSP será parte, quizá la más importante, de un sistema más grande, como un producto terminado de hardware o software, productos que requerirán metodologías adicionales y especializadas para el diseño y construcción de sus distintas partes o subsistemas, y que pueden ir desde el análisis de mercado hasta la comercialización. Por ejemplo, un teléfono celular puede incluir un sistema DSP, pero el diseño ergonómico del gabinete o el diseño del tamaño y forma de las teclas, por mencionar algunos ejemplos, escapan al alcance de la aplicación de esta metodología, aunque *pueden definir algunos de los requerimientos de la solución DSP*. Para enfoques más generales que involucran todos los niveles de desarrollo de un producto, existen metodologías como la propuesta en [GON92].

El problema debe tener características tales que sea susceptible de resolverse mediante un sistema de Procesamiento Digital de Señales (comprendiendo tal término desde un algoritmo matemático, una arquitectura, un programa de alto nivel hasta un sistema integral que incluya tanto hardware como software). Constatar que el problema posea estas características evitará desarrollar los siguientes pasos en caso de no cumplirlas. En caso de que estas características no fueran fácilmente identificables, existe la fase de análisis donde claramente se deberá definir el enfoque de la solución del problema. Pero siempre habrá que buscar la economía de tiempo y de recursos. Cuando un problema no puede ser resuelto mediante un sistema DSP, habrá que situarlo en un proceso más amplio de ingeniería.

4.2.4.2 Identificación de requerimientos iniciales.

Habrá muchas ocasiones que en este paso se identificarán y definirán algunas características del problema o de la solución que son inherentes a éstos. Reforzar esta actividad puede hacer más eficiente el segundo paso de la metodología, que es el análisis del problema. Sin embargo, habrá que tomarlos más como directrices útiles de orientación o guía que como restricciones absolutas, pues el paso de análisis puede revelar enfoques más amplios y provechosos. Por ejemplo, si se identifica que la solución del problema es una arquitectura dedicada, esto definirá mejor las siguientes fases de la metodología.

4.2.4.3 Redacción de un documento de solicitud de implantación de la solución a un problema.

En esta actividad deberá redactarse el documento donde se planteará el problema y la necesidad de la implantación de una solución. Se sugiere que el documento sea breve, que tenga un título que resuma el problema o su solución y, como contenido, únicamente el planteamiento del problema, definiendo claramente, en caso de haberlos, los requerimientos iniciales de la solución.

4.3 ANALISIS DEL PROBLEMA.

En esta fase (fig. 4.3) se tomará el documento de solicitud de solución, se realizarán las actividades pertinentes, como las entrevistas con los usuarios y la definición de requerimientos para tener como resultado una especificación completa de la solución o sistema a diseñar. Esta especificación consta de tres partes: una lista de requerimientos, un modelo estructural y un modelo funcional.

4.3.1 Objetivo.

Generar un documento de especificación del sistema, en donde deberá modelarse la solución estructural y funcionalmente. El modelo estructural representará a los elementos que constituyen el sistema y su organización. El modelo funcional representará la forma de actuar e interactuar de los elementos del sistema.

4.3.2 Entradas.

La entrada fundamental a esta fase es el documento de solicitud de solución al problema planteado. A partir de este documento deberá comenzarse el análisis, la detección de requerimientos y la definición de especificaciones.

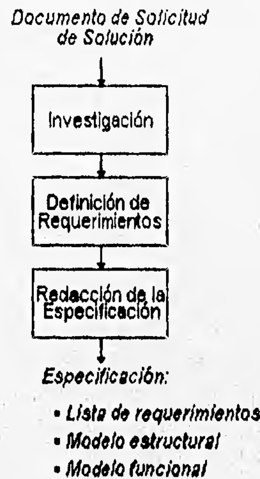


Fig. 4.3

4.3.3 Salidas.

La salida de este paso es el documento de especificaciones del sistema. El documento, como se mencionó anteriormente, incluirá los requerimientos, el modelo estructural y el modelo funcional del sistema. Para estos modelos pueden utilizarse distintas técnicas, siendo la más simple una especificación narrativa en lenguaje natural, pasando por los diagramas de flujo, diagramas de bloques, diagramas de transición de estados, etc.. Cualquier técnica deberá quedarse siempre a alto nivel, siendo el diseño detallado la actividad fundamental de la siguiente fase.

4.3.4 Actividades.

4.3.4.1 Investigación.

En esta actividad el analista recopilará ordenadamente la mayor cantidad de información sobre la naturaleza del problema. Un primer paso sería ubicar al problema dentro de una o varias de las áreas de aplicación para los sistemas DSP, como por ejemplo:

- *Comunicaciones*
- *Reconocimiento de patrones*
- *Bioingeniería*
- *Audio digital*
- *Prospección sísmica*
- *Procesamiento digital de voz*

- *Procesamiento digital de imágenes*
- *Procesamiento digital en señales de radar*

Muchas aplicaciones entran en alguna de estas grandes ramas, cada una de las cuales tiene un conjunto de algoritmos, de arquitecturas, de entornos de operación y de bibliografía que será muy útil tomar en cuenta para el proceso de diseño.

Otra herramienta importante para esta actividad es la realización de entrevistas con las personas involucradas en el problema a ser solucionado, como los usuarios potenciales, el personal de operación, etc. Para que la información sea útil y se recabe más eficiente y ordenadamente, habrá que tomar en cuenta la lista de requerimientos que se describirá en el siguiente inciso.

4.3.4.2 Definición de Requerimientos.

Esta actividad consiste en detectar y especificar los requerimientos del sistema DSP. Para llevar a cabo esta definición, habrá que emplear la información obtenida en la actividad anterior y organizarla adecuadamente. Los principales requerimientos de un sistema DSP pueden dividirse en dos: requerimientos de procesamiento de señales y requerimientos adicionales (o de no procesamiento de señales). Dentro de los primeros tenemos:

Características de las Señales de Entrada y de Salida: El objetivo general del sistema o solución a ser desarrollado mediante esta metodología es el procesamiento de señales. Es por ello que se requiere realizar una caracterización adecuada de las señales de entrada y de salida del sistema. Dentro de las características de las señales a ser definidas tenemos: número de canales, si son pasivas o activas, si son analógicas o digitales, la información deseada, si son determinísticas o aleatorias, los anchos de banda, las formas de onda, etc. [DEF88].

Opciones de procesamiento: Esto se refiere al hecho de que un sistema puede involucrar distintas opciones de procesamiento, que son los grandes campos de aplicación mencionados anteriormente, como prospección sísmica, procesamiento de imágenes, etc. [DEF88]. Por ejemplo, una red multimedia puede involucrar sistemas DSP para comunicaciones, procesamiento de imágenes, procesamiento de audio, etc.

Modos de procesamiento: Los modos de procesamiento son niveles básicos dentro de una opción de procesamiento de señales [DEF88]. Por ejemplo, dentro de la opción de procesamiento de imágenes, pueden tenerse modos de compactación, de segmentación y de vectorización. Deben definirse además las posibles concurrencias entre modos y los parámetros a controlar dentro de cada uno de ellos.

Rendimiento: En este punto deben especificarse los parámetros de capacidad de cálculo, de velocidad de transferencia de datos y de restricciones de memoria que

habrá de cumplir la solución diseñada para el problema planteado. En este punto se definirá si el sistema deberá funcionar en tiempo real.

Interfaz: En este punto se definirá la forma en que el sistema DSP habrá de comunicarse con su entorno. Esto es muy importante, pues un sistema DSP puede ser una arquitectura interconstruída (embedded) dentro de un sistema más general, donde sus entradas y sus parámetros estarán proporcionados por un sistema sensor externo, o bien un sistema DSP puede estar destinado a una interacción extensiva con uno o más usuarios humanos, como es el caso de una consola de efectos de audio y video, donde la necesidad de una interfaz más sofisticada y compleja es evidente.

Alcance:

En este punto se define el alcance que deberá tener la solución [ALC92], y que puede ser:

Solución algorítmica: Es la solución del problema expresada en funciones primitivas de Procesamiento Digital de Señales, tales como correlaciones, transformadas discretas de Fourier, filtros IIR o FIR, etc. Este es el nivel más importante de la solución, pues los niveles subsiguientes dependen directamente de éste. Cabe hacer notar que la mayor parte de la fase de diseño se concentrará en desarrollar una correcta solución algorítmica, pues la tendencia actual y futura en cuanto a desarrollo de sistemas DSP es el uso extensivo de herramientas de software que a partir de la representación en diagrama de bloques sintetizan el código en C, el código en ensamblador e incluso el hardware necesarios para ejecutar la solución algorítmica, además de poder realizar el control en tiempo real de la arquitectura (prototyping).

Solución software: Es la codificación en lenguaje de alto nivel (típicamente C) de los algoritmos desarrollados en la solución del nivel anterior. Esta solución se realiza generalmente con fines de simulación, aunque puede ser la solución final.

Solución microcódigo: Es la solución que se desarrolla cuando se tiene una arquitectura existente, basada en un procesador especializado (un chip DSP como el TMS320C50 o el DSP56000) o de propósito general (el 68HC11 o el Intel 80486) para los que habrá que escribir el código en lenguaje ensamblador.

Solución hardware: Esta solución consiste en diseñar una nueva arquitectura, que podrá diseñarse alrededor de un procesador específico. El nivel más bajo lo representa el diseño de un nuevo circuito integrado (ASIC: Application Specific Integrated Circuit).

Requerimientos de adicionales: En estos requerimientos se incluyen aquéllos relativos al entorno de operación del sistema, tales como:

- *Humedad*
- *Temperatura*
- *Tamaño*
- *Resistencia*
- *Forma*
- *Peso*
- *Costo*
- *Consumo de potencia*
- *Tolerancia a fallas*

Cabe hacer notar que estos requerimientos se especifican generalmenta para aquellos sistemas que llegarán a los niveles más bajos de solución (hardware) y se convertirán en productos terminados.

4.3.4.3 Redacción de la especificación.

En esta actividad deberá producirse el documento de especificación de los requerimientos y de los modelos estructural y funcional del sistema.

Un modelo representa objetos o sistemas. Existen distintos tipos de modelos: matemáticos, esquemáticos, de simulación, maquetas, diagramas, etc. Las distintas funciones que pueden tener los modelos son [KRI89]: predicción, control, comunicación, conceptualización y adiestramiento.

Para esta actividad, el analista del sistema podrá utilizar un conjunto de técnicas de modelado como la narrativa en lenguaje natural, los diagramas de flujo de datos [YOU93], los diagramas de bloques, los diagramas de flujo, etc. Generalmente, las especificaciones mezclan diversas técnicas para tener un modelo más completo y versátil.

Es necesario entender la diferencia entre modelo estructural y modelo funcional. El primero se refiere a la estructura o forma, mientras el segundo se refiere al comportamiento, en el que la secuencia de eventos es lo más importante. Por ejemplo, para un sistema de conversión de texto en voz, el modelo estructural podría estar definido por un diagrama como el de la figura 4.4.

El modelo funcional, usando la técnica de narrativa en lenguaje natural detallaría la serie de eventos que conforman el funcionamiento del sistema.

```
1.- El usuario introduce un texto mediante el teclado.
2.- El programa procesa el texto.
3.- El programa genera un archivo de salida de datos
4.- La tarjeta de sonido convierte los datos a una señal de au-
    dio.
5.- La señal de audio es enviada a los altavoces de salida.
```

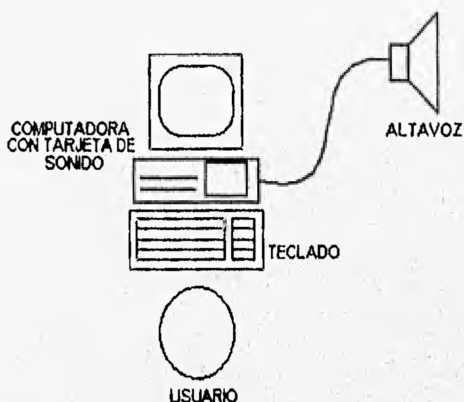


Fig. 4.4

Este es un ejemplo sencillo, pero claro, de las diferencias entre el modelo formal y el modelo funcional de la solución. La especificación deberá concentrarse en el *qué*, es decir, qué tiene que hacer el sistema, y a partir de estos modelos, el diseño se encargará de definir *cómo* realizar ese *qué*.

Se sugiere que el documento de especificación tenga una estructura cómo la siguiente:

1.- Introducción.- En esta parte se deberá presentar el problema planteado, su entorno, sus características generales y la estructura del resto del documento.

2.- Modelo del sistema.

a. Formal.
b. Funcional.

En esta parte del documento deberán detallarse los modelos de sistema haciendo uso de las técnicas que se mencionaron anteriormente. Esta es la especificación a alto nivel de la solución propuesta para el problema planteado.

3.- Requerimientos de procesamiento.- Debe enumerar y describir los requerimientos de procesamiento explicados anteriormente. Esta descripción debe ser completa pero concisa, de forma que se evite la redundancia en la información.

4.- Requerimientos adicionales.- Aquí se deben detallar los requerimientos adicionales (o de no procesamiento de señales) que deberá cumplir el sistema. Generalmente, una lista o una tabla serán de mucha utilidad para la fase de diseño.

5.- Apéndices.- En esta parte se incluirán apéndices con información adicional para que el personal encargado de las fases posteriores, principalmenta la de operación, tengan una visión más amplia y clara del sistema. Entre estos apéndices pueden incluirse resúmenes técnicos del área de aplicación del sistema, glosarios de términos, síntesis de las técnicas de modelado, etc.

6.- Indice.- Debe presentar la estructura del documento, y puede incluírse también un índice alfabético para localizar más rápidamente la información deseada. Esto es particularmente útil en aquellos sistemas muy complejos cuya especificación incluya distintas técnicas de modelado, modelos de sistemas y subsistemas, etc.

4.4 DISEÑO DE LA SOLUCIÓN Y CONSTRUCCION DEL PROTOTIPO.

Durante esta fase se diseñará una solución al problema planteado a partir del modelo estructural y funcional definidos a alto nivel en la fase de análisis; la solución puede ser una solución algoritmica, solución software, solución microcódigo o solución hardware, que deberá ser probada y validada mediante la generación de un prototipo.

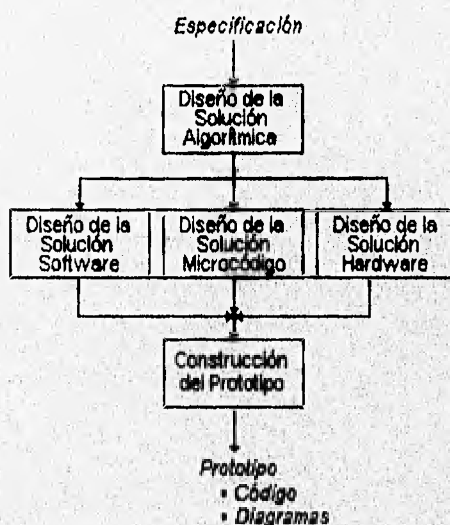


Fig. 4.5

4.4.1 Objetivo.

Generar un prototipo de la solución, el cual físicamente puede ser desde la simulación y prueba de los algoritmos diseñados, hasta un producto terminado de hardware o software. El uso de herramientas de software apoyará en la generación rápida del prototipo.

4.4.2 Entradas.

Como entrada tenemos un modelo estructural y funcional de la solución a alto nivel del problema, los cuales definen estructuralmente los elementos que componen la solución y funcionalmente qué hacen esos elementos y cómo interactúan entre ellos. Con éstos elementos comenzaremos el diseño de las diferentes soluciones.

4.4.3 Salidas.

Como salida del diseño se tiene el prototipo de la solución, entendiendo el desarrollo de un prototipo como la generación e implantación de manera rápida de la solución diseñada, haciendo uso de herramientas de software que permiten construir y simular la solución diseñada y observar los resultados obtenidos de ésta (ej. Ptolemy, Khoros, Mathematica, Matlab, etc). Es importante mencionar que éste prototipo no siempre es la solución final, sino que requiere someterse a una fase de prueba e iteración para llegar a la misma.

4.4.4 Actividades.

4.4.4.1 Diseño de la Solución Algorítmica.

Una vez recibida la especificación de la solución a alto nivel, debemos seguir explotando el modelo hasta llegar a la solución algorítmica, que consiste en plantear las funciones del sistema en términos de los algoritmos básicos de DSP, esto es, realizar una traducción epistemológica del concepto o la idea que nos proporciona el modelo funcional y estructural a un algoritmo que utilice elementos propios de los sistemas discretos y el DSP como son Transformadas Discretas de Fourier, filtros IIR, filtros FIR, etc, con lo cual podremos aplicar los conocimientos de DSP al diseño mediante diagramas de bloques (a sus diferentes niveles) de una solución. Por ejemplo: supongamos que se requiere aplicar sobre una señal de audio digital el efecto denominado eco. En primer lugar, durante la fase de análisis fue necesario investigar y documentarse sobre qué es y cómo funciona el eco para modelarlo a alto nivel.

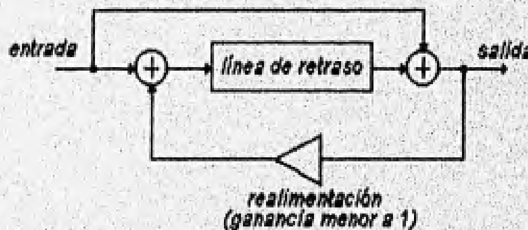
Posteriormente, habrá que traducir ese funcionamiento en términos de algoritmos DSP. Como se apreciará con más detalle en el capítulo 6 de este documento, se sabe que el eco consiste en la adición de la señal original con una versión retrasada la misma, siendo necesaria la presencia de realimentación con ganancia menor a la unidad. De esta forma se logra que el sonido vaya disminuyendo en amplitud paulatinamente hasta perderse. En este ejemplo, el paso de la especificación (alto nivel) a la solución algorítmica es casi transparente, pues el problema no es complejo y la solución ha sido especificada claramente. Los elementos de los sistemas discretos necesarios para

diseñar la solución algorítmica son una línea de retraso, dos sumadores y un amplificador de ganancia menor a la unidad.

En la actualidad existen diversas herramientas de software (referenciadas en esta tesis), algunas de las cuales han sido desarrolladas bajo la tecnología orientada a objetos, cuentan con interfaces gráficas de usuario sumamente amigables y están destinadas a diferentes labores como son diseño de circuitos integrados, simulación de sistemas DSP, integración de sistemas heterogéneos y desarrollo de prototipos. Con este tipo de herramientas la prueba y simulación de los algoritmos diseñados se convierte en una tarea sencilla. Cuentan generalmente con bibliotecas de elementos de uso general previamente programados y permiten la creación de nuevas rutinas y elementos por parte del usuario.

Muchas de los algoritmos y tareas del DSP se expresan mediante diagramas de bloques, y la tendencia de las diversas interfaces gráficas con que cuentan las herramientas de desarrollo es la creación de programas visuales en *formas* (también llamadas plantillas o templetes) en las que se "dibujan" e interconectan objetos gráficos o iconos que representan los elementos DSP, y que se extraen de bibliotecas predefinidas. Los parámetros y características de estos elementos son definidos generalmente en cuadros de diálogo. A partir de la forma se ejecuta el programa visual, cuyo objetivo será la simulación del algoritmo o bien la generación automática de código, la síntesis de arquitecturas y de circuitos integrados.

Retomando el ejemplo de la generación del eco, esquemáticamente podríamos representar su funcionamiento con el siguiente diagrama de bloques:



ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

Fig. 4.6

Para simular esta solución algorítmica en una herramienta de software como Ptolemy, debe construirse el programa visual colocando, interconectando y configurando los elementos correspondientes a la adición de señales, a una línea de retraso y a un amplificador lineal. Para fines de simulación puede utilizarse un impulso como señal de entrada y un graficador como último bloque del programa, que al ser ejecutado presentará la gráfica de una serie de impulsos que decaen exponencialmente. Este resultado indicará al diseñador que el algoritmo constituye la solución al problema planteado: la generacion de eco en una señal de audio. Sin embargo, habrá que llevar

esta solución algorítmica más cerca del entorno original del problema, que es la acústica. Será necesario entonces tener una señal de audio como entrada y una señal de audio como salida[1].

Para el ejemplo mencionado se remarcó el hecho de que el uso de una herramienta facilita y hace más rápida y eficiente la conversión de un concepto a un algoritmo DSP, por lo que la metodología planteda en nuestra tesis hace énfasis en el uso de estas herramientas como un apoyo trascendental en el diseño. Asimismo, deseamos destacar el hecho de que la solución algorítmica es la parte medular de esta fase, pues esas mismas herramientas tienden a automatizar la creación de los niveles subsiguientes (solución software, solución microcódigo y solución hardware) a partir de la solución algorítmica.

4.4.4.2 Diseño de la Solución Software.

La solución software consiste en codificar en un lenguaje de alto nivel (C++, Pascal, Smalltalk, etc) la solución algorítmica diseñada. Para realizar ésto la mayoría de los diseñadores DSP siguen codificando de una manera muy personal, es decir, sin aplicar metodologías de programación de software como la orientada a objetos, la cual proporciona grandes beneficios (software y código reutilizable, prototipos rápidos, fácil rediseño y un buen control sobre las estructuras de datos) o la metodología estructurada que orienta a una prueba constante del diseño realizado hasta obtener la solución final. Por lo tanto, consideramos importante recomendar que estas técnicas se conviertan en un estandar de programación de algoritmos DSP, en lenguajes poderosos como C++ orientado a objetos.

En la mayoría de las aplicaciones DSP se requiere que la solución presente un rendimiento en tiempo real. La característica principal de los sistemas de tiempo real es que pueden controlar un ambiente recibiendo datos, procesándolos y devolviéndolos con la suficiente rapidez como para influir en dicho ambiente en ese mismo instante. Por lo tanto, el rendimiento de la codificación a alto nivel para una arquitectura de procesamiento de propósito general puede ser insuficiente, lo que es particularmente claro en algoritmos de alta intensidad de cálculo (como las transformadas discretas de Fourier). La aplicación de la solución software será, por lo tanto, la simulación fuera de línea de los algoritmos diseñados.

También se recurre a la solución software porque muchas veces las tareas de DSP forman parte de un proyecto más grande (software o hardware) en donde el código programado en la solución software pudiera integrarse, cumpliendo una tarea específica. Por ejemplo, un sistema que controle operaciones de edición, grabación y reproducción de audio en una consola, requerirá una interfaz amigable con el usuario y

---

[1] Ptolemy incluye, dentro de sus fuentes de señales (bloques para definir señales de entrada) archivos de audio digital en formato au (formato de audio PCM ley-μ usado en las estaciones de trabajo SUN) y dentro de sus bloques de salida un elemento que convierta una secuencia de valores lineales al formato au y los reproduce en el altavoz de la estación de trabajo. De igual forma, se tienen opciones para imágenes, video, datos de distintos tipos, etc. Herramientas similares cuentan con elementos correspondientes según sea su plataforma de trabajo. De no existir los elementos, será labor del diseñador la creación de los mismos.

toda una metodología de diseño de software para su creación, pero muy probablemente el código generado en la solución software de los algoritmos DSP se integraría al sistema completo cumpliendo una tarea específica dentro del funcionamiento del mismo (ej. generación de eco, reverberación, amplificación de la señal, etc.).

Por otra parte, herramientas de software como Ptolemy sintetizan el código de alto nivel necesario para desempeñar la solución representada visualmente en el ambiente gráfico, y debido a que estas herramientas han sido desarrolladas bajo la tecnología de orientación a objetos, el código que generan es altamente eficiente y tiene una fuerte congruencia con los modelos algorítmicos generales utilizados para el procesamiento digital de señales. Estas posibilidades son reales y la tendencia presente y futura, sobre todo en los sistemas de información, es que la codificación y construcción de un sistema consuman cada vez menos tiempo y recursos, adquiriendo más importancia las fases de análisis y diseño conceptual. Finalmente, es importante mencionar que las labores de codificación estarán orientadas a la creación de nuevos elementos y nuevas bibliotecas que extiendan las amplias capacidades de las herramientas de desarrollo.

4.4.4.3 Diseño de la Solución Microcódigo.

Otra de las posibilidades existentes es que dentro de la solución al proyecto de DSP se requiera programar una arquitectura basada en un procesador de propósito general (como el Intel 486) o un arquitecura basada en un procesador especializado DSP (como el DSP56000 de Motorola). A la codificación de los algorimos en el lenguaje ensamblador del procesador a utilizar se le conoce como solución microcódigo.

Al igual que la programación en alto nivel de los algoritmos DSP, la codificación en lenguaje ensamblador de los mismos se sigue realizando de forma primitiva. Generalmente resulta difícil modificar un programa en lenguaje ensamblador, pues la programación no sigue una metodología. Actualmente la tendencia es crear un estándar de programación apoyándonos en alguna metodología existente. La metodología más conveniente para ésto es la orientada a objetos, por las ventajas que posee de proporcionar codigo reutilizable y buen control de las estructuras de datos, siendo un estándar de facto el lenguaje C++. Matti Karjalainen [KAR92] propone la aplicación de herramientas orientadas a objetos para la programación de chips DSP en tiempo real, en particular el TMS320C30 de Texas Instruments, y da algunos ejemplos de la programación de algoritmos DSP típicos como filtros FIR en un ambiente orientado a objetos llamado QuickC30. Este ambiente está basado en la definición de clases de tipos de datos y métodos como un tipo específico de funciones y procedimientos.

Sin embargo, independientemente de la manera en que se programe el microcódigo, algunas de las herramientas de desarrollo (como Ptolemy) permiten sintetizar el microcódigo para un procesador específico a partir de la representación gráfica. Entre las ventajas de esta generación automática de código están el ahorro de tiempo en la

codificación y la certeza de que el código sintetizado funciona adecuadamente, pues la etapa de simulación del sistema valida el funcionamiento del algoritmo.

4.4.4.4 Diseño de la Solución Hardware.

La solución hardware se presenta cuando el proyecto requiere el diseño y construcción de una arquitectura enfocada a resolver una o muchas tareas de DSP en forma general o específica.

Así como existen metodologías de desarrollo de software, para el desarrollo del hardware existen metodologías como la planteada en [DEF88] que en sus fases incluye eventos que van desde la definición de los requerimientos de procesamiento y no-procesamiento, hasta el mapeo e implantación en hardware de la solución algorítmica planteada. En resumen, la solución hardware puede tener dos caminos en su diseño:

1) Construir una arquitectura alrededor del funcionamiento de un procesador general o específico.

2) Diseñar y construir un nuevo procesador y la arquitectura necesaria para su funcionamiento.

El nivel más bajo de diseño está representado por la construcción de un nuevo chip (ASIC), el cual en forma general debe ser dividido en operaciones primitivas directamente implantables en su diseño. Un ejemplo de una operación primitiva es una FFT para cuatro muestras.

En el caso de que se desee implantar la solución en un procesador específico, se debe buscar que las operaciones primitivas directamente implantables elegidas estén preprogramadas en el procesador y cumplan con los requerimientos de desempeño y de utilización de recursos; de no ser así deberá iterarse hasta seleccionar otras primitivas que logren el rendimiento adecuado. Cuando hablamos de un procesador general, deberán analizarse procesadores existentes hasta encontrar uno con la mejor respuesta para las operaciones primitivas propuestas. Por otro lado si lo que se requiere es un procesador nuevo, el enfoque a desarrollar es más flexible: se definen un conjunto de operaciones primitivas y se utilizan para definir la arquitectura de procesamiento necesaria para efectuarlas, considerando requerimientos de programabilidad, flexibilidad, capacidades aritméticas, velocidad de cálculo y requerimientos de no procesamiento.

Para realizar la parte de la solución hardware de una manera más fácil están las herramientas de desarrollo de prototipos, como Ptolemy, que hace el trabajo mucho más sencillo sintetizando una arquitectura de hardware óptima que desempeñe la aplicación representada gráficamente, y nos auxilian en las tareas de simulación, además de llevar a cabo el control de la arquitectura en una aplicación en tiempo real.

4.4.4.5 Presentación del prototipo de solución.

Conceptualmente el prototipo es la construcción de la solución diseñada para el problema planteado; físicamente, el prototipo está determinado por el alcance definido en el análisis.

Cuando se requiere la solución algorítmica, el prototipo generalmente consiste en el diseño algorítmico de la solución, representado generalmente por un diagrama de bloques o de flujo de datos, identificando los elementos DSP que intervienen, y cómo se deben interconectar para lograr la solución. La verificación de este tipo de solución se lleva a cabo mediante una herramienta de simulación.

Para una solución software, el prototipo será la codificación y prueba de los algoritmos en algún lenguaje de alto nivel como C++ . Físicamente, debe presentarse el programa corriendo y una documentación completa del funcionamiento del mismo, los parámetros que requiere, la especificación de la entrada o entradas del programa y cuál será la salida.

Para la solución microcódigo, el prototipo que se presentará será la simulación de los algoritmos DSP en alguna herramienta de software y el código en ensamblador para un procesador definido, ya sea general o dedicado, la simulación del funcionamiento del microprocesador en alguna herramienta e inclusive la arquitectura con el microprocesador programado.

El prototipo presentado para la solución hardware es la simulación de la operación y control de la arquitectura en tiempo real mediante un software de simulación, presentando el funcionamiento de cada elemento de la arquitectura diseñada y la especificación de todos los parámetros. Además, deberán hacerse las asignaciones de hardware y software iniciales, es decir, programar los chips necesarios y desarrollar un software de control y de aplicación.

En caso de haberse desarrollado un nuevo procesador, se deberán desarrollar dos documentos: el de principios de operación (POO), que define la arquitectura y el de especificación de desempeño del programa de cómputo (PPS), que define los requerimientos del sistema operativo [DEF88].

4.5 PRUEBA Y REFINAMIENTO DEL PROTOTIPO.

4.5.1 Objetivo.

Probar el prototipo de solución, mediante un proceso de validación (confirmación y verificación) y rediseño tratando de apegarse los más posible a la solución real requerida por el usuario, modificando en cada iteración condiciones o agregando nuevos requerimientos.

4.5.2 Entradas.

EL primer prototipo de solución generado directamente del diseño mediante herramientas de software de simulación.

4.5.3 Salidas.

EL prototipo final de solución después de varias iteraciones y pruebas haciendo lo que se quiere que haga y haciendolo bien.

4.5.4 Actividades.

La prueba y refinamiento del prototipo consta de dos grandes decisiones, una referente a lo que se está haciendo y la otra si se está haciendo bien, es decir, como se ilustra en la fig. 1, la confirmación responde a la pregunta ¿se está haciendo lo pedido en el análisis? y la verificación responde a: lo que se está haciendo, ¿se hace en forma correcta o de la mejor manera?.

4.5.4.1 Confirmación.

La confirmación requiere tener muy presentes las especificaciones y requerimientos del análisis para poder checar que se cumplan fielmente de acuerdo. Si la confirmación falla debemos revisar el análisis de la solución, buscando redefinir uno o varios requerimientos y/o especificaciones y reestructurando el modelo funcional, el modelo estuctural, o ambos. Esta revisión debe hacerse con el usuario final del proyecto, quien validará que el prototipo sea la solución correcta al problema planteado. Inmediatamente se procede a elaborar un nuevo prototipo (aquí la importancia de generar un prototipo en forma rápida) que represente los cambios hechos en el análisis para volver a someterse a confirmación; si éste es rechazado, comienza un proceso de iteración hasta que sea confirmado el prototipo, una vez hecho lo cuál, se pasará al proceso de verificación.

4.5.4.2 Verificación.

La verificación consiste primero en identificar si existe algún error o alguna mejora al prototipo de solución y después identificar la parte de la solución en la que hay que hacer las correcciones o modificar. Al identificar el error, es necesario volver al diseño, para modificar o mejorar la solución algorítmica, con lo que habrá que reediseñar las soluciones derivadas de ella: solución software, solución microcódigo y solución hardware (si así se requiere); posteriormente, habrá que reflejar los cambios en el

prototipo y comenzar el proceso iterativo hasta que el prototipo representa a la mejor solución posible.

No obstante que exista una fase de prueba y refinamiento del prototipo en la fase final de la metodología, es conveniente ir realizando la prueba de cada parte de la solución por separado, con lo que se asegura que la prueba final será mucho mas completa y con menos posibilidades de error. Por ejemplo, si se está desarrollando como solución al problema un procesador o un chip específico, algunos de los parámetros a definir serán: almacenamiento (local y masivo), anchos de banda de las señales definidas, precisión y respuesta, entre otros, por lo que durante la fase de diseño de estos elementos, habrá que probar su correcto funcionamiento, preferentemente en una herramienta de simulación, independientemente de que al final del diseño y contruccion del prototipo tenga que hacerse una prueba general.

4.6 OPERACION.

Esta fase es la última de la metodología, y en ella el sistema se pondrá en funcionamiento para dar solución al problema para el que fue diseñado. Esta fase se divide claramente en dos secciones (fig. 4.6): documentación y mantenimiento. Las salidas parciales de la primera drcción serán la documentacion, tanto del sistema como del usuario, que permita iniciar la operación y realizar el mantenimiento del sistema, donde se observará la necesidad de reiniciar la metodología para adaptarlo, corregirlo o mejorarlo.

4.6.1 Objetivo.

Proporcionar los instrumentos de documentación necesarios para llevar a cabo la operación y el mantenimiento correctivo, adaptivo y de perfeccionamiento del sistema de una forma eficiente y ordenada.

4.6.2 Entradas.

Las entradas a esta fase serán los documentos de planteamiento del problema, el documento de especificación y el código y los diagramas desarrollados durante la fase de diseño.

4.6.3 Salidas.

Las salidas de esta fase serán el manual de usuario y el manual del sistema desarrollado. Cuando en esta fase se detecte la necesidad de modificación del sistema, se recomenzará la metodología de diseño.

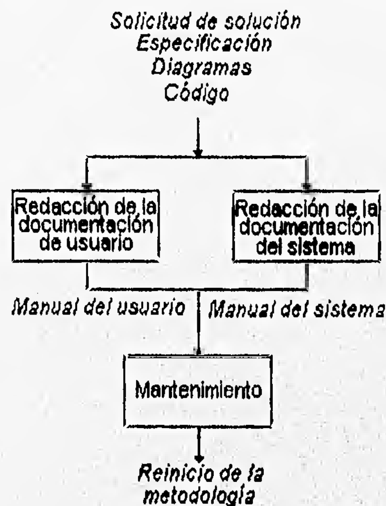


Fig. 4.7

4.6.4 Actividades.

4.6.4.1 Redacción de la documentación de usuario.

En esta actividad se producirá un documento que contenga:

- Guía de instalación del sistema
- Guía de introducción, donde se explicará como iniciar la operación del sistema
- Manual del referencia, donde se describirá cómo operar la totalidad de las funciones del sistema.

4.6.4.2 Redacción de la documentación del sistema.

Aquí se redactará la documentación del sistema, que incluye desde la especificación hasta una guiá de prueba del sistema. Para esta actividad será de gran utilidad contar con los documentos de planteamiento del problema, con la especificación y con los diagramas y el código fuente que se hayan generado en la fase de diseño. En esta documentación se deben incluir:

- El documento de especificación, incluyendo la lista de requerimientos y los modelos formal y funcional del sistema.

- La descripción de los módulos de software y componentes de hardware que constituyan la solución con sus especificaciones técnicas, así como la descripción de su operación.
- Una guía de pruebas para cada módulo o componente y una guía de prueba general o de integración [SOM88] para la totalidad del sistema.

4.6.4.3 Mantenimiento.

En esta actividad se iniciará el proceso de transformación de la solución que ya se encuentra en operación. Existen tres modalidades de mantenimiento [SOM88]:

- Mantenimiento correctivo: Se presenta cuando se han localizado fallas en la operación del sistema.
- Mantenimiento adaptivo: Se presenta cuando han cambiado las condiciones del entorno de operación del sistema.
- Mantenimiento de perfeccionamiento: Se presenta cuando el usuario desea un cambio en la forma en que funciona el sistema para hacerlo más eficiente, más poderoso o para cambiar alguna de sus características.

Hemos dicho que la actividad inicia el proceso de transformación de la solución, puesto que cuando se detecta una necesidad de mantenimiento equivale a volver a iniciar la metodología. La profundidad con que se desarrollará la metodología dependerá de la magnitud del cambio requerido. Debe resultar evidente que para cambios menores no será necesario generar nuevos modelos formales o funcionales, pero sí habrá que tomar en cuenta la utilidad de aplicar la metodología para un desarrollo eficiente.

CONCLUSIONES.

Hemos detallado la propuesta de una metodología básica de diseño de sistemas DSP integrando elementos de las diferentes metodologías existentes para el desarrollo de sistemas de información y de arquitecturas de procesamiento, así como las herramientas de software que simplifican varios pasos de la metodología. El desarrollo de la tecnología , de la que estas herramientas son producto, esta impulsando el que la labor fundamental de los diseñadores de sistemas DSP se centre en la creación de una solución algorítmica óptima, puesto que las actividades restantes del diseño son y serán en el futuro realizadas de una manera simple y rápida e incluso automática por las herramientas de software en cuyo desarrollo se está trabajando ampliamente.

Otro aspecto importante que deseamos recalcar es que esta metodología esta enfocada al diseño de sistemas DSP, tomando en cuenta que un sistema DSP puede ser a su vez un componente o elemento de un sistema más complejo , como puede ser un producto terminado de software o hardware, para cuyo diseño deberá aplicarse una metodología apropiada. Es entonces necesaria una constante interacción entre los distintos equipos de desarrollo, pues del sistema general surgirán restricciones y especificaciones que influirán en la parte DSP del mismo, que a su vez definirá ciertas condiciones que deberán tomarse en cuenta en el desarrollo del sistema general.

REFERENCIAS.

[ALC92] Alcántara, Rogelio. "Metodología de diseño de sistemas para el procesamiento digital de señales". DEPFI-UNAM, México, 1992.

[DEF88] DeFatta, David J. et.al. *Digital Signal Processing. A system design approach*. Wiley, Estados Unidos, 1988.

[GON92] González Villela, Víctor Javier. Tesis de maestría: *Metodología de diseño electrónico en un proyecto de desarrollo tecnológico*. Facultad de Ingeniería-UNAM, México, 1995.

[KAR92] Karjalainen, Matti. "Object -Oriented Programming of DSP Processors: A Case Study of Quick C30", *Proceedings ICASSP-92*, vol. 5. IEEE, Estados Unidos, 1992.

[KRI89] Krick, Edward V. *Introducción a la ingeniería y al diseño en la ingeniería*. 2a. ed. Limusa, México, 1989.

[SOM88] Sommerville, Ian. *Ingeniería de software*. 2a. ed. Addison-Wesley, México, 1988.

5

Principios de Audio Digital.

INTRODUCCION.

El Audio Digital se ha desarrollado intensamente en los últimos años, debido a que algunas funciones especiales y particulares usadas en la tecnología de audio son más baratas y más simples, se disminuyen los niveles de degradación sónica que existen en los procesos analógicos, los datos con los que se representan las señales se operan sobre altas velocidades con el uso de una computadora (Procesamiento Digital de Señales, DSP). Además es inherentemente más inmune a la interferencia que el audio analógico y la calidad del sonido es independiente del medio por el cual se transmita o almacene la señal de audio.

En general con el DSP es posible detectar y corregir errores en la señal que se recibe, los cuales pueden haber sido causados por problemas en la transmisión, como interferencias o distorsiones debidas a la presencia de algún campo magnético.

Como el audio es analógico por naturaleza, es necesario aplicar procesos que discreticen las señales de audio. Es por esto que necesitamos muestrear y cuantificar la señal analógica, teniendo en cuenta la existencia de errores como el aliasing que es un tipo de distorsión que se produce por no apegarse al teorema de muestreo o el error que se genera de una mala cuantización de la señal analógica.

En este capítulo presentaremos algunos conceptos generales sobre audio digital, así como las unidades de medición usadas en audio y conceptos importantes sobre la discretización de señales analógicas de audio.

5.1 CARACTERISTICAS DE LAS ONDAS DE SONIDO.

El audio digital puede ser concebido como la digitalización del sonido. El sonido es originado por un movimiento vibratorio que se propaga a través de medios materiales elásticos. Uno de dichos medios de propagación es el aire. El aire atmosférico en ausencia de sonido presenta una determinada presión que se considera constante (presión estática), siendo la presión normal: $P_o = 1023 \times 10^5 \frac{N}{m^2}$. Al generarse un sonido se producen una serie de compresiones y refracciones de las partículas afectadas por la perturbación, lo cual hace que la presión varíe por encima y por debajo de la presión estática. En la propagación de la onda sonora físicamente lo que ocurre es una transferencia de energía de una molécula a la próxima; el movimiento de una molécula origina el movimiento de las demás. Por tanto, la propagación de un sonido no implica traslación de materia, lo que se traslada es energía. Vista en forma matemática es una onda longitudinal cuya energía sería la amplitud de la señal, la cual se puede digitalizar mediante el muestreo y la cuantización que se explicarán más adelante [HER81].

La longitud de onda constituye el espacio recorrido en un periodo y representada gráficamente es la distancia entre puntos sucesivos correspondientes en la misma fase y es simbolizada por la letra griega lambda "λ", Fig 5.1. El rango entre la frecuencia más baja y la más alta se puede definir como el ancho de banda de un sistema. La amplitud es la altura de la señal por abajo y por arriba del nivel de equilibrio o presión atmosférica estática [TAL94]. El periodo es el tiempo que se tarda en producir una vibración completa. Este es el inverso de la frecuencia: $T = \frac{1}{F}$. Como la unidad de medida es el segundo, se puede inferir que la frecuencia es el número de periodos que se producen en un segundo; generalizando un simple despeje de la fórmula del periodo, la frecuencia es el inverso de éste: $F = \frac{1}{T}$

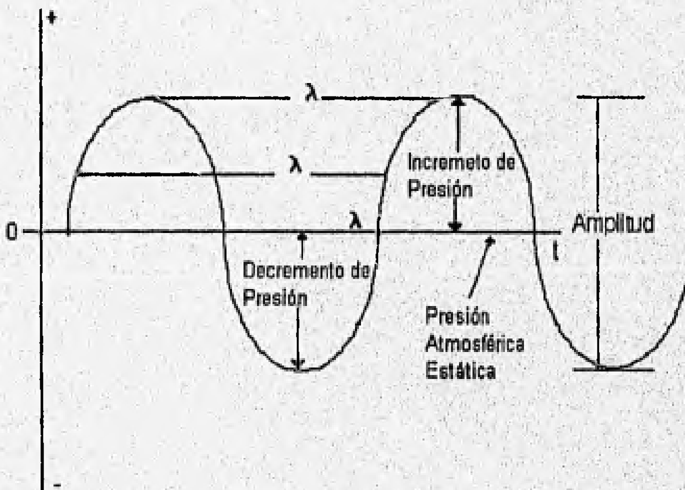


Fig. 5.1

Normalmente el oído humano puede percibir frecuencias desde los 16 Hz hasta aproximadamente 16 KHz y la velocidad de propagación de las ondas del sonido depende del medio por el que se propagan. En el aire viene dada por :

$$C = C_0 \sqrt{\frac{T}{273}} \frac{m}{s} \qquad (5.1)$$

Siendo C_0 la velocidad de propagación a 0°C (331.6 m/s). La temperatura T se expresa en grados Kelvin; a la temperatura centígrada normal se le sumará 273. Por ejemplo a una temperatura de 20°C la velocidad será $C = 331.6\frac{m}{s}\sqrt{\frac{20+273}{273}} \cong 343\frac{m}{s}$.

Existe una relación muy útil entre la longitud de onda, la velocidad de propagación del sonido y la frecuencia de la onda, resumida en una sencilla fórmula: [TAL94].

Frecuencia x Longitud de onda = velocidad (f x l = V) (5.2)

reescribiendo la ecuación 5.2 tenemos:

$$\text{Longitud de Onda} = \frac{\text{Velocidad}}{\text{Frecuencia}} \text{ o } \lambda = \frac{V}{f} \qquad (5.3)$$

Es importante mencionar que la velocidad de las ondas no se afecta por el hecho de que la longitud de onda cambie, por lo tanto la velocidad es constante, así que si la longitud de onda es mayor, la frecuencia es menor y viceversa. En otras palabras, cuando la fuente que genera las ondas de sonido se acerca al receptor, se produce el efecto Doppler, el cual dice que siempre que exista un movimiento relativo entre la fuente y el receptor, se produce un cambio en la frecuencia de la onda.

5.1.1 Fenómenos del sonido.

El sonido puede ser reflejado o absorbido de muchas maneras. Por ejemplo el paso del sonido por el aire atenúa la energía de éste, atenuando mucho más las frecuencias altas. En un lugar cerrado, el sonido puede reflejarse hacia atrás o chocar con las paredes para generar un efecto como el eco o una reverberación de éste.

El fenómeno de reflexión o reflejo del sonido (eco) ocurre cuando la longitud de onda es mucho más pequeña que las dimensiones del obstáculo con el cual choca, produciéndose el efecto del eco, que es cuando la señal choca con el obstáculo y rebota.

El fenómeno de difracción ocurre cuando la longitud de la onda es apreciablemente más grande que las dimensiones del objeto con el cual choca, en este caso el sonido no se refleja, sino que rodea al objeto y continua desplazándose en dirección del desplazamiento original de la onda.

En realidad los dos fenómenos siempre existen juntos, es decir son complementarios, pero uno se produce más que el otro dependiendo de la longitud de la onda y el tamaño del obstáculo contra el cual choque ésta [TAL94].

La refractación del sonido es el cambio de dirección con el cambio de la temperatura. Es por esto que cuando la temperatura es baja la velocidad disminuye y el sonido tenderá a cambiar de dirección desde una zona caliente hacia una zona fría [POH89].

5.1.2 Medición del sonido.

Los conceptos sobre las unidades de medición más comunes del sonido se definen a continuación:

Potencia. Es la cantidad de trabajo o energía que se produce o consume en un proceso. Sus unidades son los watts. Como las señales de audio son una forma de energía, podemos decir que en potencia sus unidades serían las mismas, pero el oído humano es muy sensitivo, por lo que requerimos unidades mas pequeñas de potencia, como son los microwatts (μW) [TAL94].

Intensidad. Debe entenderse como la caída de potencia o energía de la señal de audio cuando se encuentra sobre o pasa por un área específica, sus unidades obviamente serían $\frac{W}{m^2}$ o $\frac{\mu W}{m^2}$ [TAL94].

Decibeles. El sistema auditivo responde de una forma aproximadamente logarítmica. Las sensaciones de sonoridad que percibimos son proporcionales a los logaritmos de las intesidades sonoras que originan los estímulos. Hay una relación definida entre las intensidades sonoras $\frac{W}{cm^2}$ y las sensaciones que éstas nos producen, las que pueden expresarse tal como lo muestra la sig. tabla 5.1:

Intensidad	Sensaciones
I_0	0
$I_1=10^1\ I^0$	1
$I_2=10^2\ I^0$	2
$I_3=10^3\ I^0$	3
$I_4=10^4\ I^0$	4
.	.
.	.
$I_n=10^n\ I^0$	n

Tabla 5.1

I_0 es la intensidad sonora mínima audible para $F = 1000Hz.\left(I_0 = 10^{-16}\frac{W}{cm^2}\right)$, a la cual se le asigna la sensación 0.

Según puede observarse, una sensación doble a la que nos produce el nivel I_1, no la percibimos con 2 veces el nivel I_1 ($2\,I_1$) sino que es preciso aumentar 10 veces el nivel I_1 ($10\,I_1$), con lo cual obtenemos el nivel I_2. Para percibir una sensación doble a la que nos produce el nivel I_2 será preciso aumentar 100 veces el nivel I_2, con lo cual obtenemos el nivel I_4, que será 10.000 veces mayor que I_0. Fácilmente se deduce que mientras las intensidades sonoras aumentan en progresión geométrica, las sensaciones lo hacen en progresión aritmética. A cada aumento en una unidad de la sensación sonora le corresponde un aumento de 10 veces la intensidad, lo cual es una relación logarítmica.

Queda claro, pues, que el sistema auditivo responde de una forma logarítmica; las sensaciones son proporcionales a los logaritmos de las intensidades que originan los estímulos. Puesto que el sistema auditivo responde de forma logarítmica, las intensidades sonoras no tienen un significado muy real ya que dichas magnitudes no corresponden a las sensaciones sonoras que producen. Por ello, las atenuaciones y aumentos de potencia, presiones, intensidades, etc., se miden con una unidad logarítmica: el decibel.

Según lo explicado anteriormente, podemos decir que las sensaciones vienen dadas según la expresión:

$$\textit{nivel de intensidad}\,(IL) = K_1 \log(I) + K_2 \qquad (5.4)$$

donde I es el nivel de intensidad sonora a medir y K_1 y K_2 son constantes que pueden variar según se construya la escala de valores. Asignando la sensación 0 al nivel de intensidad 10 y haciendo $K_1 = 10$, se obtiene:

$$0 = 10\log(I_0) + K_2 \;\; \text{y} \;\; K_2 = -10\log(I_0) \qquad (5.5)$$

de la ecuación 5.4 y 5.5 se establece la fórmula:

$$IL = 10\log\frac{I}{I_0}dB \qquad (5.6)$$

donde I_0 es el nivel de referencia e I el nivel de intensidad a medir. Esta expresión nos permite expresar sensaciones de sonoridad, relativas a un nivel de referencia, en un sistema de unidades logarítmico cuya unidad se denomina decibel (dB).

$$IL = 10\log\frac{I}{I_0} = 10\log\frac{10^{-12}}{10^{-16}} = 40dB \qquad (5.7)$$

Una de las unidades más empleada en acústica es la denominada Sound Pressure Level, nivel de presión sonora (SPL).

$$SPL = 10\log\left(\frac{P}{P_0}\right)^2 = 20\log\frac{P}{P_0}dB \qquad (5.8)$$

Siendo P la presión sonora a medir y P_0 la presión sonora de referencia, que vale, en el aire: $P_0 = 2 \times 10^{-4}$ μbar.

Dado que las atenuaciones y aumentos de potencia se miden en unidades logarítmicas y tomando en cuenta la definición del nivel de presión sonora podemos definir al decibel como una relación logarítmica entre dos potencias.

$$\text{Nivel de Potencia} = 10\log\left(\frac{P_1}{P_2}\right)dB \qquad (5.9)$$

donde P_1 es una potencia determinada y P_2 un nivel de potencia de referencia. Si el número de dB resulta negativo (por ejemplo -3 dB), se trata de una atenuación, siendo entonces P_1 menor que P_2.

Por ejemplo, si P2 = .001 watt es la potencia de referencia y la potencia de salida de un micrófono es de 0.0000001 watt, el nivel de potencia es :

$$\begin{aligned} \text{nivel de potencia} &= 10\ \text{Log}(P1/P2)\ \text{dB} \\ &= 10\ \text{Log}\ 10^{-7}/10^{-3} \\ &= 10\ \text{Log}\ 10^{-4} \\ &= 10\ (-4)\ \text{dB} \\ &= -40\ \text{dB} \end{aligned}$$

El decibel, además de en acústica, se emplea en electrónica para medir atenuaciones o ganancias de tensión, potencia, etc. Por ejemplo, en un amplificador se usa para medir el realce (o atenuación) de graves y agudos mediante los controles de tono, la relación señal-ruido, etc.

Cuando se usan señales de voltaje, corriente o presión de sonido, cuyos cuadrados son proporcionales a la potencia, la ecuación 5.9 se reescribe :

$$\text{Nivel de Potencia} = 10\log_{10}\left(\frac{P_1}{P_2}\right)^2 dB = 2x10\log_{10}\left(\frac{P_1}{P_2}\right)dB = 20\log_{10}\left(\frac{P_1}{P_2}\right)dB \qquad (5.10)$$

El SPL más bajo que el umbral auditivo humano permite es de 0 dB SPL y el umbral más alto antes de sentir malestar o molestia es de 120 dB SPL.

En lo relativo a los instrumentos musicales en general se emplean una serie de términos que por una parte se relacionan con la acústica y por otra con la música, uno de éstos términos es la octava, la cual se define más adelante.

5.1.3 Tono Musical.

Por Tono podemos entender la posición de una nota en una escala musical, la cual a su vez se asocia con una frecuencia [TAL94].

- El Tono estándar se considera un sonido a una frecuencia de 440 Hz. Este es conocido como LA, esta nota afina a una orquesta antes de tocar.
- Un Tono musical o nota musical, puede relacionarse con una frecuencia determinada, como en la sig. tabla 5.2:

NOTA	FRECUENCIA (Hz.)
A, LA (octava arriba).	440
G#, SOL sostenido	415
G, SOL	392
F#, FA sostenido	370
F, FA	349
E, MI	330
D#, RE sostenido	311
D, RE	294
C#, DO sostenido	277
C, DO	262
b, SI	247
a#, LA sostenido	233
a, LA	220

Tabla 5.2

- Octava musical. Dada una nota la cual tiene asociada una frecuencia, una octava musical arriba o abajo de la nota, se entiende como aumentar al doble o disminuir a la mitad la frecuencia asociada a ésta, es decir si se tiene una frecuencia de 440 Hz, una octava abajo sería 220 Hz, y una octava arriba serían 880 Hz, aproximadamente, esto hablando en una escala musical, donde cada nota está asociada a una frecuencia específica, de tal manera que podamos hablar de octavas entre notas musicales, no entre frecuencias. Decimos que una nota es octava de otra si sus respectivas frecuencias están en relación:

$$\frac{F_1}{F_2} = 2 \qquad (5.11)$$

Así pues el intervalo entre 100 y 200 Hz se define como una octava, como lo es el intervalo entre 1000 y 2000 Hz. En términos lineales la segunda octava es más larga que la primera, es por esto que la escala logarítmica abarca un amplio rango de valores y podemos usarla para medir las grandes variaciones que se presentan en la medición de las características del sonido, por ejemplo el umbral sensitivo humano es de 120 dB SPL que es 1,000,000,000,000 veces más alto que el umbral auditivo de 0 dB SPL.

- Semitonos adyacentes (en una escala musical van de una nota blanca a una nota negra o viceversa), los cuales tienen 6% de frecuencia hacia abajo (bemol) o hacia arriba (sostenido) de la nota deseada [TAL94].

5.1.4 Calidad musical (timbre).

Conceptualmente el timbre es el conjunto de factores o parámetros de una señal acústica que permiten o ayudan al oído a distinguir entre un instrumento y otro. Por ejemplo el saxofón y la trompeta producen sonidos completamente diferentes aún si estuvieran tocando la misma nota. Existen 2 factores importantes en la determinación del timbre [TAL94]:

1.- Armónicas, estos son tonos cuyas frecuencias son múltiplos exactos de la frecuencia base (llamada la frecuencia fundamental). En general las armónicas disminuyen en amplitud conforme crecen en frecuencia. En música generalmente las armónicas son conocidas como sobretonos.

2.- Las Frecuencias Transitorias, se presentan en un periodo muy corto al inicio de cada nota, del orden de centésimas de segundo, sin relación numérica con la frecuencia fundamental como la tienen las armónicas. Estas frecuencias mueren rápidamente de ahí su nombre de transitorias. Si éstas son eliminadas o distorsionadas es posible que cambie la calidad con que se percibe el sonido del instrumento que se escucha.

5.1.5 Intensidad del sonido y su efecto en la distancia.

La Intensidad del sonido decrece con la distancia. Existe una relación que nos determina perfectamente esta relación :

$$I \propto \frac{1}{d^2} \qquad (5.12)$$

Donde d es la distancia desde la fuente del sonido. Por ejemplo suponiendo que hay 1m de distancia desde la fuente, la intensidad I es de 4 $\frac{W}{m^2}$. Así a 2 m la intensidad

será de 1 $\frac{W}{m^2}$ y a 8 m. sería $\frac{1}{16}\frac{W}{m^2}$. Esta relación se conoce como ley del cuadrado inverso [TAL94].

Dos aspectos importantes debemos considerar en la ley :

- La fuente de sonido es pequeña, comparada con la distancia.
- No hay reflexión de superficies.

5.1.6 Formas de onda complejas.

En general todas las señales estan compuestas de una frecuencia fundamental y series de señales armónicas relacionadas numéricamente con la fundamental. La forma de onda periódica más simple es el seno la cual es importante porque existe como una frecuencia fundamental. Si sumamos a la frecuencia fundamental de una onda cualquiera la tercera armónica fig. 5.2, se crea una forma de onda nueva. Si la tercera armónica se defasa en el tiempo y se suma a la onda con frecuencia fundamental, se crea otra nueva forma de onda. Las dos formas de onda tendrían el mismo tono, debido a que la frecuencia fundamental es la misma, pero obviamente el timbre seria diferente por las series armónicas que están conformando a cada onda. Esto se resume en el teorema de Fourier, el cual afirma que todas las señales periódicas armónicas están compuestas por series armónicas de señales senoidales [POH84].

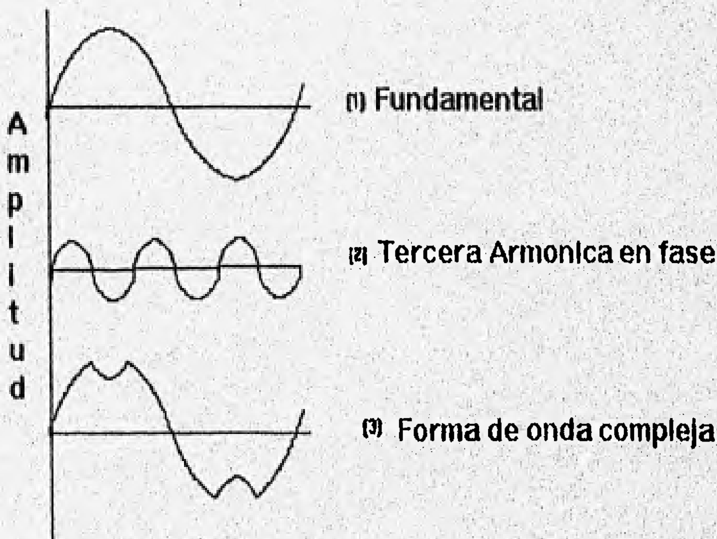


Fig. 5.2

5.2 MUESTREO DISCRETO EN EL TIEMPO.

En general la mayoría de las señales en ingeniería o ciencias resultan ser analógicas por naturaleza. La manera de procesarlas puede ser directamente como señales analógicas, o primero sometiendo a la señal a un proceso de digitalización. Esta conversión analógico-digital se realiza con dos pasos fundamentalmente : 1) Muestreo y 2) Cuantización como lo muestra la fig. 5.3:

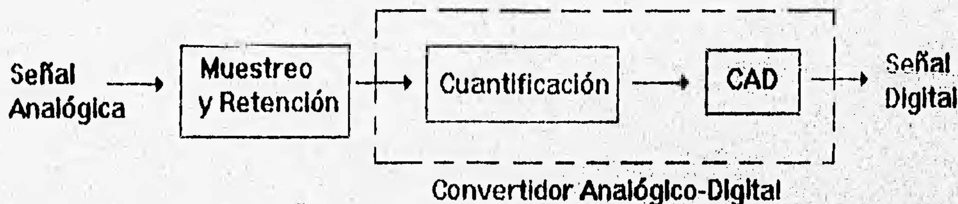


Fig. 5.3

Si un sistema digital es muestreado discretamente, ¿Qué pasa con la señal entre los tiempos de muestreo?, ¿se pierde la señal original?. Siempre y cuando se tengan las condiciones correctas no se perderá información durante el periodo de muestreo y la señal muestreada contendrá la misma cantidad de información que la señal no muestreada.

Por ejemplo: si para una señal $x(t)$, seleccionamos una frecuencia de muestreo para digitalizarla F_S, y conocemos la frecuencia máxima de la señal $x(t)(F_{MAX})$ podemos lograr condiciones óptimas para evitar pérdidas de la señal al digitalizarla haciendo que $F_S > F_{MAX}$ además de utilizar un método de interpolación apropiado. Dicha fórmula de interpolación se expresa en el siguiente teorema, llamado *el teorema de muestreo.*

Teorema de Muestreo : Si la frecuencia más grande contenida en una señal analógica $x_a(t)$ es $F_{MAX} = B$ y la señal es muestreada a una tasa de $F_S \geq 2F_{MAX} = 2B$, entonces $x_a(t)$ puede ser exactamente recuperada de las muestras, usando la función de interpolación [ZEP90]:

$$g(t) = \frac{\sin 2\pi Bt}{2\pi Bt} \qquad (5.13)$$

Así que podemos expresar $x_a(t)$ como :

$$x_a(t) = \sum_{n=-\infty}^{\infty} x_a(\tfrac{n}{F_S})g(t - \tfrac{n}{F_S}) \qquad (5.14)$$

Donde $x_a(\frac{n}{F_s}) = x(n)$ son las muestras de $x_a(t)$. Cuando la frecuencia de muestreo es $F_N = 2B = 2F_{MAX}$ recibe el nombre de **Frecuencia de Nyquist**.

El proceso de convertir una señal en el tiempo discreto a una señal digital, expresando cada valor de la muestra como un número finito de dígitos (en lugar de un infinito) es llamado cuantización.

Como una representación física, un sistema para muestrear una señal, está representado en la fig. 5.4:

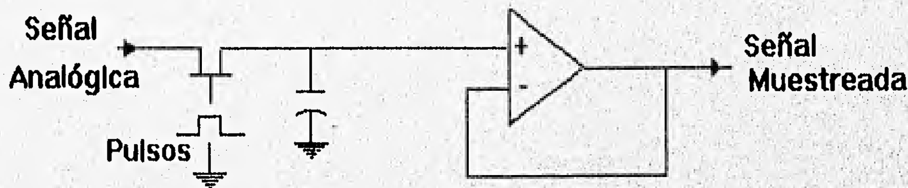
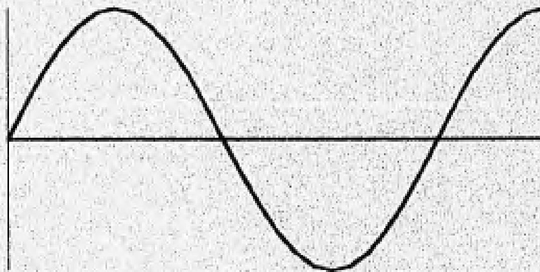


Fig. 5.4

Siempre es conveniente que en un sistema de audio digital exista un filtro paso bajas que preceda al circuito de muestreo para remover las frecuencias que estén arriba o que sobrepasen la mitad de la frecuencia de muestreo, ya que según el teorema de muestreo, la frecuencia máxima de la señal debe ser menor que la mitad de la frecuencia de muestreo, así que el filtro debe ser paso bajas tomando como frecuencia de corte la mitad de la frecuencia de muestreo. También es importante y recomendable colocar un filtro paso bajas a la salida del sistema para remover las frecuencias altas que se generan internamente en el sistema, este filtro además ayuda a recobrar la señal original, reconstruyendo la señal muestreada mediante el efecto que se conoce como escalera, el cual se muestra en la fig. 5.5 :



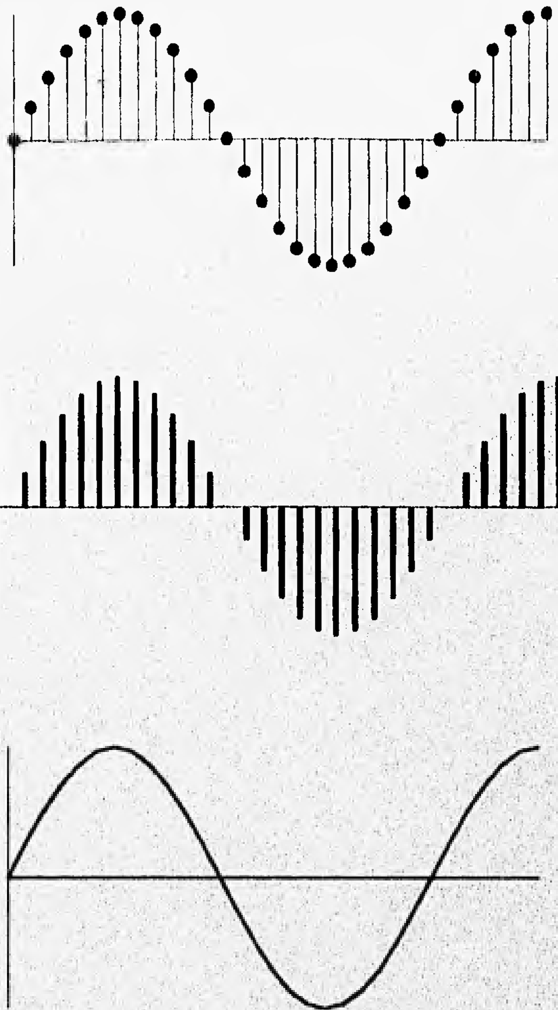


Fig. 5.5

No obstante debemos tener mucho cuidado y tratar de combatir dos tipos fundamentales de distorsión: la condición de frecuencias erróneas, donde el proceso de muestreo de la señal se hace sin respetar el teorema de muestreo enunciado anteriormente, produciéndose el efecto conocido como aliasing y el error resultante de la cuantización de la forma de onda analógica, llamado *error de cuantización* o *ruido de cuantización.*

5.3 EL FENOMENO DEL ALIASING.

Cuando la frecuencia de muestreo es menor que el doble de la frecuencia máxima de la señal analógica a digitalizar, se presenta el fenómeno conocido como aliasing. Esto

quiere decir que si la frecuencia de muestreo es apenas igual al doble de la frecuencia máxima de la señal a muestrear, solo existen dos muestras por ciclo, el mínimo necesario para representar la naturaleza bipolar de la señal, pero si se intenta muestrear a frecuencias menores que el doble de la frecuencia máxima de la señal a digitalizar, el sistemas seguirá proporcionando muestras a intervalos regulares, pero creará componentes falsas de la señal, las cuales tienen la probabilidad de aparecer dentro del ancho de banda de audio y son imposibles de distinguir de las señales legítimas. Estas componentes falsas tienen frecuencias llamadas de *traslape* cuyo valor está dado por la ecuación 5.15 [POH84]:

$$F_a = S - F \tag{5.15}$$

donde S es la frecuencia de muestreo y F es la frecuencia máxima que sobrepasa la mitad de la frecuencia de muestreo.

5.3.1 Solución al Fenómeno del Aliasing.

Para solucionar el problema del aliasing es recomendable implementar una etapa de entrada que restrinja la señal mediante un filtro paso bajas (filtro anti-aliasing, Fig. 5.6) diseñado para proporcionar una atenuación considerable a la mitad de la frecuencia de muestreo para asegurarse que la capacidad de frecuencia nunca exceda la mitad de la frecuencia de muestreo. Una vez que existe el aliasing no hay técnica que permita la eliminación de las frecuencias de aliasing del ancho de banda de audio original [POH89].

Fig. 5.6

5.4 CUANTIZACION.

Para grabar una señal de audio se tiene que almacenar información en dos dimensiones, tiempo y amplitud. La cuantización almacena la información en amplitud, mientras que el muestreo almacena la información en tiempo de la señal de audio. La cuantización se define entonces como el valor medido de la amplitud de la señal analógica muestreada en el tiempo, en otras palabras es la técnica de medir un evento analógico para formar un valor numérico, el cual en un sistema digital, normalmente se

implementa con números binarios. En términos de la cuantización el número posible de valores numéricos está determinado por el tamaño de la palabra de datos de la computadora. Con la cuantización como todos los eventos analógicos, la exactitud está limitada por la resolución del sistema. Debido al tamaño finito de las variables, se introduce un error de medición. Dicho error es similar al ruido en un sistema analógico [POHL89].

5.4.1 Razón Señal a Error.

Con un sistema de números binarios, el tamaño de la palabra de datos determina el número de intervalos de cuantización disponibles. Esto se puede determinar elevando el tamaño de la palabra a una potencia de dos, como se muestra en la tabla 5.3

$2^1=2$	$2^7 = 128$	$2^{13} = 8192$	$2^{19} = 524288$
$2^2=4$	$2^8 =256$	$2^{14}= 16384$	$2^{20}= 1048576$
$2^3=8$	$2^9 = 512$	$2^{15} = 32768$	$2^{21} = 2097152$
$2^4=16$	$2^{10}= 1024$	$2^{16}=65536$	$2^{22}=4194304$
$2^5=32$	$2^{11} = 2048$	$2^{17} = 131072$	$2^{23} = 8388608$
$2^6=64$	$2^{12} = 4096$	$2^{18} = 262144$	$2^{24} = 16777216$

Número N de intervalos de cuantización en una palabra binaria igual a $2^{n\ bits}$

Tabla 5.3

Donde N representa el número de intervalos de cuantización, cuando la palabra es de n bits. Si se tiene un número mayor de bits será mejor la aproximación, pero siempre existe un error asociado con la cuantización debido al número limitado de amplitudes que pueden ser contenidas en una palabra binaria, la cual nunca podrá representar un número infinito de posibilidades analógicas.

Existe un punto en el que el error no puede ser distinguido por el sistema auditivo humano. Al respecto los especialistas en el área han acordado que una adecuada representación se alcanza con 15 a 20 bits.

Para conocer el desarrollo de un sistema digitalizador se conoce una fórmula de aproximación señal a error, la cual es la división entre el número máximo de intervalos (N - I), que el esquema de cuantización cubre, entre el máximo intervalo de error definido. Este valor se expresa en la ecuación 5.16:

$$\text{Aproximación señal a error} = \frac{\text{número máximo de intervalos}}{\text{máximo error definido}} \quad (5.16)$$

Por ejemplo, para 16 bits:

Aproximación señal a error = $\frac{65535}{0.5}$ = 131,070 que es equivalente a 98 dB.

Esta razón de la máxima amplitud representable con el error, determina la razón señal a error (S/E) del sistema.

La razón señal a error se expresa mediante el cálculo del valor máximo de la señal, el valor de error y posteriormente formando su razón de cambio. Con un sistema de cuantización en donde cada intervalo de muestreo tiene amplitud "Q" y "n" es el número de bits de la palabra de la computadora, el máximo nivel de señal de cero a pico es $\frac{1}{2}(2^n Q) = 2^{n-1} Q$ y el valor máximo de la señal RMS es:

$$\text{Señal de Voltaje (RMS)} = \frac{2^{n-1}Q}{\sqrt{2}} \quad (5.17)$$

La señal de error se analiza mediante la función de probabilidad de error de cuantización. Cuando la señal de entrada tiene una amplitud alta y un espectro ancho, el error tiene la misma probabilidad de ser un valor entre $+\frac{Q}{2}, -\frac{Q}{2}$.

La energía contenida en el ruido se calcula realizando la integral de todos los valores de error obtenidos y multiplicando la energía contenida en un error (E) por su probabilidad P(E) dE, ecuación 5.18 .

$$\text{Energía de Error} = \int_{-\frac{Q}{2}}^{+\frac{Q}{2}} E^2 P(E) dE \text{ con } P(E) = \frac{1}{Q} \quad (5.18)$$

$$\text{Error de Voltaje (RMS)} = \frac{Q}{\sqrt{12}} \quad (5.19)$$

En términos de la razón señal a error, la ecuación es:

$$\frac{S}{E} = \frac{\frac{2^{n-1}Q}{\sqrt{2}}}{\frac{Q}{\sqrt{12}}} = 2^n \sqrt{1.5} \quad (5.20)$$

Por lo tanto la razón señal a error es igual a, $2^2 \sqrt{1.5}$ donde "n" es el número de bits.

En decibeles la ecuación 5.20 se expresa como:

$$\text{Razón señal a error} = \frac{S}{E} \approx 6.02n + 1.76dB \quad (5.21)$$

Por ejemplo: 16 bits dan una S/E de 98 dB; 15 bits 92 dB; 14 bits 86 dB.

5.4.2 Distorsión en la Cuantización.

El error de cuantización es la diferencia entre el valor analógico muestreado en el tiempo y el valor de intervalo escogido para la cuantización. En el tiempo de muestreo el valor de la amplitud tiene que ser seleccionado del intervalo de cuantización más cercano. Bajo las mejores condiciones la señal coincide con el intervalo de cuantización. En el otro caso la señal se encuentra en medio de dos intervalos. Concluyendo, el error de cuantización se encuentra limitado por $\pm\frac{1}{2}$ del intervalo en el tiempo de muestreo.

Debido a que el error de cuantización está en función de la señal original no se puede clasificar como ruido, sino más bien, como una forma de distorsión. Se puede formular un análisis que demuestre la magnitud de la distorsión resultante de la cuantización.

Si "Q" es el intervalo de cuantización también es la amplitud pico a pico del error de cuantización. El valor instantáneo del error de cuantización ($\pm e$) tiene una distribución uniforme de 0 a "Q" y su valor pico se toma como $e = \frac{Q}{2}$. La potencia del producto de la distorsión aplicada a una unidad resistiva es:

$$D = \frac{2}{Q}\int_0^{\frac{Q}{2}} e^2 de \tag{5.21}$$

$$D = \frac{2}{Q}\left(\frac{(\frac{Q}{2})^3}{3}\right) = \frac{Q^2}{12} \tag{5.22}$$

Lo anterior demuestra que la potencia de distorsión es independiente de la amplitud de la señal de entrada, pero está relacionada con el tamaño del intervalo de cuantización.

El porcentaje de distorsión se calcula a partir del numero "n" de bits en el sistema y del nivel de la señal "S" relativa a la señal máxima posible. Con "n" bits existen un número máximo de $N = 2^n$ valores y $2^n - 1$ intervalos (I). El número de intervalos en uso (I_S) es:

$$I_S = (2^n - 1)\,anti\log 10\frac{S}{20} \tag{5.23}$$

El voltaje pico a pico de la señal de nivel "S" se genera de la multiplicación del nivel de cuantización y el número de intervalos utilizados, es decir $pp = QI_S$ y el voltaje pico es $\frac{QI_S}{2}$.

Asumiendo una señal senoidal la potencia promedio a una resistencia unitaria $\frac{1}{4}(QI_S)^2$ será de $P = \frac{Q^2 I_S^2}{8}$, la potencia de distorsión es $\frac{Q^2}{12}$ y el porcentaje de distorsión (THD%) es: $\frac{D}{P}(100\%)$

$$TDH\% = \frac{\frac{Q^2}{12}}{\frac{Q^2 I_S^2}{8}} 100\% = \frac{2}{3 I_S^2} 100\% \qquad (5.24)$$

donde:

$$I_S = (2^n - 1) antilog\, 10 \frac{S}{20} \qquad (5.25)$$

Sustituyendo valores en la ecuación, el porcentaje de distorsión es nulo a 0 dB y menor a 0.1% a -70 dB.

CONCLUSIONES.

Durante este capítulo hemos tratado los aspectos más elementales de las ondas de sonido definiendo una señal de audio, sus características, unidades de medida y fenómenos que se presentan cuando se trabaja con estas señales, tratando de explicarlos desde un punto de vista matemático.

A partir de esto pudimos definir algunas características del audio digital desde el teorema de muestreo, la cuantización y la detección del error existente. Debe pues quedar claro que las señales de audio en general son analógicas y existe la necesidad de discretizarlas para poder operar dichas señales mediante algoritmos DSP, y posteriormente convertirlas a señales analógicas tomando en cuenta el error que se sufre durante esta conversión A/D y D/A.

REFERENCIAS.

[HER81] Hermosa Donate, Antonio. *El moderno órgano electrónico; circuitería y sistemas*. Marcombo, España, 1981.

[POH89] Pohlmann, Ken C. *Principles of Digital Audio*. SAMS, Estados Unidos, 1989.

[TAL94] Talbot, Smith Michael. *Audio Recording and Audio Reproduction; Practical Measures for Audio Enthusiasts*. Newnes, Gran Bretaña, 1994.

[ZEP95] Zepeda, Claudia. Tesis de licenciatura: *Sistema de Clasificación de Señales EEG Utilizando el Ambiente de Programación Visual Khoros*. Facultad de Ciencias Físico Matemáticas, BUAP, México, 1995. p. 20.

6

Diseño de un Sistema de Producción de Efectos Especiales en Señales de Audio.

INTRODUCCION.

En este capítulo se desarrollará la semblanza de la aplicación de la metodología propuesta en el diseño de un sistema de producción de efectos especiales en señales de audio. Como cabe recordar, en la metodología se incluye la redacción de varios documentos, como el de solicitud de solución del problema, la especificación, los manuales de usuario y del sistema, etc. En el cuerpo de este capítulo, así como en los apéndices B, C y D, se encuentra el texto correspondiente a estos documentos, pero se ha preferido integrarlo como explicación narrativa para ilustrar el proceso seguido durante la aplicación de la metodología que como documentos independientes.

Se ha restringido el tipo de señal de audio a la de una guitarra, con el fin de diseñar un grupo de efectos característicos para tal instrumento, aunque varios de ellos pueden ser aplicados a cualquier señal de audio.

6.1 PLANTEAMIENTO DEL PROBLEMA.

El sistema a desarrollar consiste en un productor de efectos especiales en señales de audio. La señal de audio sobre la que operará el sistema es la de una guitarra eléctrica, señal que deberá ser modificada en distintas formas a las que se conoce como efectos especiales.

6.2 ANALISIS DE LA SOLUCION.

6.2.1 Antecedentes Históricos.

La guitarra eléctrica es un instrumento musical desarrollado a partir de la guitarra acústica tradicional. Este instrumento consta de seis cuerdas colocadas y tensadas sobre un brazo y una caja de madera que proporciona la resonancia acústica amplificando el sonido producido por la vibración de las cuerdas. Los distintos tonos musicales que abarcan la cuerdas dependen de tres parámetros: el grosor, la tensión y la longitud de cada cuerda. El grosor no puede variarse y la afinación inicial de una guitarra depende de la tensión que se haga de las seis cuerdas, alcanzando una escala de tonos definida. Por lo tanto, la forma de alcanzar distintos tonos en la escala musical es variando la longitud de las cuerdas. Esto se logra "pisando" u oprimiendo las cuerdas contra el brazo en una posición definida. El brazo tiene una cierta longitud y divisiones llamadas trastes, de forma que en cualquier lugar del traste que la cuerda sea pisada, se tiene una longitud fija. Esto quiere decir que, a diferencia de los instrumentos de arco como el violín, el número de tonos es discreto.

La guitarra eléctrica fue inventada por el músico norteamericano Les Paul, quien tocaba la guitarra acústica en una orquesta. El sonido de su instrumento se veía opacado por el mayor volumen de las secciones de metales y las percusiones, de manera que Les Paul concibió la idea de amplificar eléctricamente el sonido de su instrumento mediante un micrófono. Ante la necesidad de desplazarse en el escenario y por la incomodidad evidente del micrófono de pedestal para este fin, diseñó un tipo de micrófono especial para ser montado en la guitarra al que llamó "pastilla" (en inglés: pickup). La pastilla es un transductor que transforma la vibración acústica de las cuerdas en una señal de voltaje, haciendo innecesaria la presencia de una caja de resonancia. De esta forma, la pastilla se conecta a un amplificador de potencia y a un altavoz haciendo audible la señal. El sonido de la guitarra eléctrica es completamente distinto al de la acústica, pero ese carácter distinto le dio un potencial mucho mayor al tenerse la posibilidad de modificar las características de la señal eléctrica generada por la pastilla. Les Paul hizo caso omiso de las críticas que de su instrumento hacían los puristas y comenzó el diseño y construcción de los primeros efectos para la guitarra eléctrica.

6.2.2 Tipos de Efectos Especiales para Guitarra Eléctrica.

Los efectos especiales para guitarra eléctrica pueden clasificarse en tres grandes grupos:

a) Efectos que alteran el nivel de la señal

Estos efectos, como su nombre lo indica, modifican en alguna u otra forma la amplitud o nivel de la señal de la guitarra eléctrica para producir un efecto especial. Los principales efectos de este tipo son la distorsión, el compresor y el trémolo.

b) Efectos que alteran la respuesta en frecuencia.

Estos efectos modifican o alteran la señal en función de su frecuencia. Un ejemplo clásico de estos efectos es un ecualizador gráfico, que divide a la señal en un conjunto de bandas de frecuencia, de cada una de las cuales puede alterarse el nivel para acentuar o atenuar la presencia de cada banda de frecuencias en la señal total.

c) Efectos que desplazan a la señal en el dominio del tiempo.

En estos efectos, se retrasa la señal una cierta cantidad de tiempo y se mezcla esta señal retrasada con la señal original para producir efectos como el eco, el coro y la reverberación.

6.2.3 Requerimientos del Sistema.

6.2.3.1 Señales de entrada y salida.

La señal de entrada será una señal de audio analógico captada por las pastillas de la guitarra eléctrica o por un micrófono, para poder extender la solución a otras fuentes de sonido.

- *Número de canales de entrada*: 1.
- *Ancho de banda*: se contará con un ancho de banda genérico de audio, que va de 20 a 20000 Hz.
- *Señal aleatoria.*
- *Forma de onda*: La forma de onda típica de una guitarra asemeja un diente de sierra.
- *Tamaño de la palabra de la señal digital*: Típicamente, se trabaja con 8 bits para aplicaciones de voz y con 16 bits para audio digital comercial. Para fines de simulación, se trabajará con el formato de 8 bits PCM ley μ (μ-law) utilizado en las estaciones de trabajo SUN y con el formato VOC, propio de las tarjetas de sonido Sound Blaster de Creative Labs.
- *Frecuencia de muestreo*: La frecuencia de muestreo típica para voz es 8 kHz. Para audio digital comercial es de 44 KHz.

La señal de salida deberá ser también analógica, de forma que pueda ser enviada a un amplificador de audio.

6.2.3.2 Opciones de procesamiento.

La opción de procesamiento única para este problema es audio digital.

6.2.3.3 Modos de procesamiento.

El sistema debe incluir un conjunto de efectos especiales como: distorsión, "flanger", coro (chorus), reverberación (reverb), eco (echo) y trémolo. Estos distintos efectos serán los modos de procesamiento.

6.2.3.4 Alcance.

El proceso de diseño se enfocará fundamentalmente al desarrollo de la solución algorítmica. Se utilizará una herramienta de software (Ptolemy) [BUC94] para simular y analizar los algoritmos diseñados. A partir de estos algoritmos se generará el código de alto nivel que nos permita compilar programas para ser ejecutados en una PC con una tarjeta de audio. Con fines demostrativos, se generará también el microcódigo correspondiente; sin embargo, debido a que no se cuenta con una arquitectura de procesamiento adecuada, no podrá probarse el microcódigo generado. Finalmente, debido a que la solución no alcanzará el nivel de un producto terminado, no se definirán por el momento requerimientos de no-procesamiento.

6.2.4 Especificación de la Solución.

A continuación se especificarán funcionalmente a alto nivel los efectos de audio; a partir de esta especificación se diseñarán los algoritmos de procesamiento digital de señales que realizarán estos efectos.

6.2.4.1 Distorsión.

Se le conoce en inglés como "Fuzz" y "Overdrive". Es un efecto de alteración de la amplitud o nivel que distorsiona la señal de la guitarra, lo que se logra saturando (overdriving) un amplificador hasta el punto de distorsión. El proceso básico mediante el cual los circuitos analógicos realizan esta distorsión consta de dos partes: primero se amplifica la señal de la guitarra y luego se recorta. En una variante del efecto se suavizan los bordes de la señal recortada para lograr un sonido más natural. El tipo de distorsión depende de qué tipos de recorte se realicen. Una de las variantes más usuales consiste en recortar sólo la parte positiva de la señal. La razón de esto resulta algo curiosa, pues se dice que el efecto surgió por accidente cuando uno de los bulbos de un amplificador de potencia en configuración push-pull se quemó, provocando la saturación, [AND84].

Los controles de este efecto son:

Sensitividad.- Controla qué tanta amplitud de entrada es necesaria para generar distorsión. Esto puede lograrse variando la ganancia y dejando fijo el nivel de recorte, o bien dejando fija la ganancia y variando el voltaje de recorte. Lo más usual es el primer caso.

Salida.- Controla el nivel de salida del efecto.

Puede tenerse también un ecualizador de tres bandas, llamado control de tonos, para las frecuencias bajas, medias y altas, con el objetivo de acentuar o atenuar la presencia de estos tonos en la señal distorsionada.

6.2.4.2 Trémolo.

Consiste en variar cíclicamente la amplitud de la señal de entrada. En los circuitos analógicos esto se logra mediante un amplificador controlado por voltaje (VCA) al que se le alimenta la salida de un oscilador de baja frecuencia (LFO) como voltaje de control. La ganancia de un amplificador controlado por voltaje es directamente proporcional a un voltaje de control. De esta forma, si tenemos un oscilador de baja frecuencia que da un voltaje alternante y con éste controlamos al VCA, tendremos una ganancia que varía cíclicamente. La frecuencia del oscilador debe ser baja, ya que si fuera lo suficientemente alta alcanzaría una frecuencia audible y se convertiría en un tono.

Existen básicamente dos controles para operar el trémolo: velocidad y profundidad (depth).

Velocidad.- Determina la frecuencia de oscilación del oscilador. Las frecuencias varían entre 1 y 15 Hz, lo que es inferior a las frecuencias audibles.

Profundidad.- Varía o regula la intensidad del efecto, es decir, controla qué tanto varía la amplitud de la señal de entrada.

6.2.4.3 Efectos de Desplazamiento en el Tiempo (Time Shifting).

Los efectos de desplazamiento que se realizarán son el *delay*, el *eco*, el *reverb*, el *coro* y el *flanger*. En general, se habla de que los efectos anteriores operan sobre el dominio del tiempo tomando la señal de entrada y retrasándola un cierto lapso. Al componente principal de los efectos de este tipo se le llama línea de retraso (delay-line).

Para ejemplificar el trabajo de estos efectos supongamos que tenemos uno de ellos llamado simplemente "retraso", que toma la señal de un micrófono y la retiene un cierto

período de tiempo después del cuál la proporciona como salida, la cuál será a su vez alimentada a un amplificador de potencia que impulsa un altavoz.

Si un cantante se encuentra en un cuarto con paredes de cristal para impedir que su voz se escuche en el exterior y colocamos nuestro retraso de forma que el cantante tenga el micrófono consigo dentro del cuarto y nosotros tengamos el altavoz fuera del mismo, apreciaríamos el efecto cuando el cantante comience a cantar. Nosotros veríamos que entre el momento en que él mueve sus labios emitiendo una nota y el momento en que nosotros oímos esa nota a través del altavoz existe un lapso.

Este ejemplo básico de retraso constituye la clave de funcionamiento de los efectos de time-shifting. Ahora, ¿de qué forma se generan las distintas variantes de estos efectos? Estas variantes como el eco, el coro y el flanger dependen de distintos parámetros y elementos que explicaremos a continuación.

Si deseamos un *eco* de la señal, necesitamos mezclar (o sumar) la señal original con la señal retrasada, de forma que si el cantante canta una sola nota, oigamos la nota original y la nota retrasada. Sin embargo, si deseamos contar con un efecto parecido al eco natural en donde tenemos varias repeticiones, cada una de menor volumen con respecto a la anterior, debemos realimentar la señal retrasada al efecto, de forma que se genere un segundo, un tercer y más retrasos, obteniendo una amplitud menor en cada uno al escalar la realimentación por un factor menor a la unidad.

Otra aplicación del retraso en el tiempo es el efecto de *coro (chorus)* con el que se simula la presencia de otra u otras fuentes de sonido combinando la señal original con una señal retrasada ligeramente. Cuando dos guitarristas tocan la misma melodía, existe siempre un ligero retraso de uno con respecto al otro, pero este retraso no es uniforme, es decir, el guitarrista 2 se retrasa a veces con respecto al guitarrista 1 y en ocasiones sucede al revés. Si deseamos simular esto, puede variarse el tiempo de retraso del efecto para lograr un sonido más real. Cabe hacer notar que en tiempo real nunca podremos simular que el guitarrista 2 (la señal retrasada) se adelante al guitarrista 1 (la señal original).

Cuando el retraso es muy pequeño, se logra el efecto de ida y vuelta (flanger), con el que la señal parece provenir de un avión a reacción.

Según sea el intervalo en que se encuentre el tiempo de retraso será el efecto producido, de esta forma, presentamos aquí la clasificación que hace Craig Anderton de estos efectos:

"Retrasos de 0 a 15 ms: Mezclar una señal retrasada de 0 a 15 ms con la señal original produce el efecto conocido como "flanger", un dramático efecto especial que imprime un sonido de jet a la señal de la guitarra.

"Retrasos de 10 a 25 ms: Mezclar una señal retrasada de 10 a 25 ms con la señal original produce el popular efecto de coro. Esto crea un sonido más lleno, más animado que recuerda los sonidos de dos instrumentos tocándose al mismo tiempo.

"Retrasos de 25 a 50 ms.- Aquí se comienza a entrar en los intervalos del eco, donde se percibe que la señal retrasada ocurre después de la señal original. En este rango se tiene el eco llamado "slapback", que es un eco muy cercano al sonido original.

"Retrasos de más de 50 ms.- Con los retrasos del orden de 250 ms se tiene la sensación de estar tocando el instrumento en un espacio muy grande. Con retrasos de alrededor de 50 ms se tiene la impresión de estar en un cuarto pequeño." [AND84]

Los efectos analógicos comercializados tradicionalmente se han orientado a estos intervalos de retraso, es decir, se tienen un circuito para flanger, otro para chorus y otro para eco. En ocasiones se tienen unidades de flanger/chorus en una sola caja, pero difícilmente se tiene una caja con los tres efectos. Sin embargo, de forma general, podemos encontrar los mismos controles en estos efectos:

Retraso.- Define el lapso de tiempo inicial que se retrasará la señal.

Mezcla.- Como sabemos, se debe mezclar o sumar la señal original con la señal retrasada, estos controles permiten variar qué tanta señal original y qué tanta señal retrasada se tienen a la salida.

Realimentación.- Define el factor por el que escalará la señal para la realimentación. Generalmente su valor máximo es inferior a 1, pues una realimentación unitaria prolongaría indefinidamente la señal y una realimentación mayor eventualmente saturaría los amplificadores y los altavoces. El valor mínimo es 0, con lo que se tendría una sola repetición de la señal.

Modulación.- Como mencionamos anteriormente, para los efectos de chorus y flanger es deseable modificar ligeramente el tiempo de retraso para hacer más real el efecto. Dos controles definen el monto y la frecuencia de esa variación del tiempo de retraso: frecuencia y variación. Por ejemplo, si para un flanger tenemos un tiempo de retraso inicial de 5ms, podemos darle una variación de 1 ms de forma que el tiempo de retraso varíe de 4 a 6 ms de forma cíclica, con una frecuencia de variación de 10 Hz, por ejemplo.

Control de tono.- Este control permite atenuar las frecuencias altas de la señal retrasada para simular mejor el eco natural, en donde para cada repetición se atenúan estas frecuencias.

Ahora llegamos al último efecto de este tipo: el *reverb*. Este efecto simula el sonido de un instrumento siendo tocado en un auditorio grande. Esto es distinto al eco, que repite un sonido hasta que se desvanece. Cuando nos encontramos en un auditorio, no sólo

escuchamos el sonido emitido directamente por la fuente, sino también una miríada de sonidos provenientes de la reflexión del sonido original contra el techo, los muros, el piso, las butacas e incluso el público del auditorio. Por siglos, nos hemos acostumbrado a oír el sonido de esta forma y cualquier sonido deja de parecernos natural. De esta forma, se han desarrollado dispositivos para simular este comportamiento del sonido y que son llamados reverberadores (reverberators).

Tradicionalmente se han construido reverberadores con conjuntos de resortes impulsados eléctricamente por la señal de audio y se han construido también reverbs completamente eléctricos. Los controles de estos efectos son:

Entrada.- Controla la amplitud de la señal a la entrada del efecto, de forma que se tenga una reverberación más marcada cuando se impulsa una señal mayor.

Mezcla.- Controla la suma de la señal directa con la señal procesada por el reverb.

Estructuralmente, el sistema consistirá de la solución algorítmica desarrollada como un programa visual de Ptolemy que opere los efectos diseñados sobre una señal de audio digitalizada y que reproduzca la señal de salida con el (los) efecto(s) aplicado(s).

6.3 DISEÑO DE LA SOLUCION ALGORITMICA.

A partir del análisis, se construirá la solución algorítmica y la solución software del problema utilizando una herramienta. La herramienta seleccionada es Ptolemy, dadas sus capacidades de programación visual, de generación de código y de extensibilidad.

Con los modelos definidos en la fase anterior, se diseñaron las soluciones algorítmicas de la distorsión, el trémolo, el delay, el eco y el reverb. A partir de ellas, se construyeron los programas visuales en Ptolemy para simular el comportamiento de las soluciones. A continuación se detallarán los principales aspectos de la construcción y simulación de cada efecto.

6.3.1 Distorsión.

La solución para este efecto consiste en una amplificación y un recorte. La construcción del programa visual en Ptolemy requirió utilizar un amplificador a la entrada, además elementos de control como un comparador y un multiplexor. Del comparador se obtiene el valor de selección del multiplexor; de esta forma, si comparamos la señal de entrada con una constante, obtendremos un valor lógico que determinará la salida del multiplexor. El programa visual se presenta en la figura 6.1.

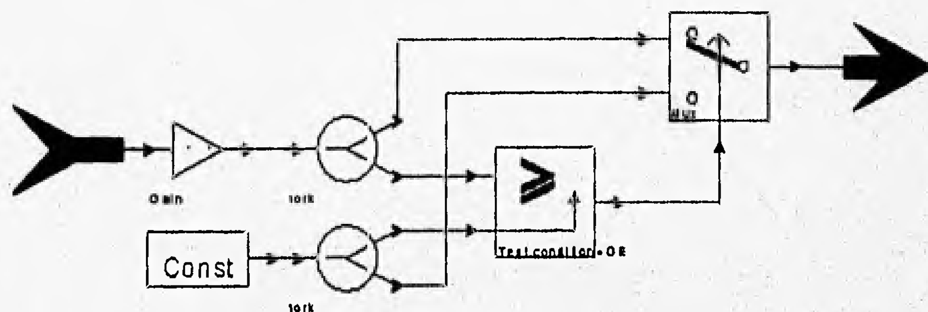


Fig. 6.1

La respuesta del algoritmo se simuló alimentando una señal senoidal y graficando la salida (fig. 6.2).

Fig. 6.2

La gráfica de la respuesta se presenta en la figura 6.3. Puede observarse el recorte de la señal en 0.7, que fue la constante que se definió como nivel de recorte.

Salida de la distorsion a entrada senoidal (recorte a 0.7)

Y

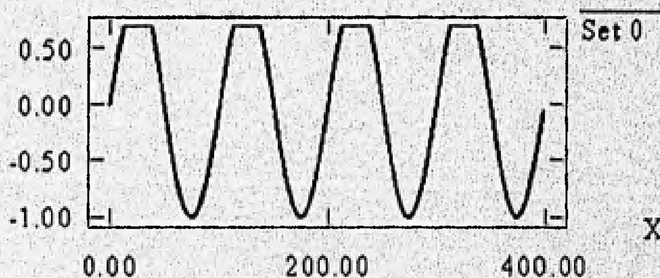


Fig. 6.3

6.3.2 Trémolo.

Este efecto consiste en la variación cíclica de la amplitud de la señal de audio de entrada. Para lograr esto, es necesario multiplicar la señal de entrada por una señal periódica de baja frecuencia. Se ha elegido utilizar una señal senoidal como moduladora. La salida *y* estará definida por la expresión: $y(n)=x(n)*(1+\alpha*lfo(n))$, donde $lfo(n)$ es el valor de la senoidal de baja frecuencia en el tiempo n y α es un factor real entre 0 y 1 que define la profundidad de la variación en la amplitud de la señal de entrada. De esta forma, si α =0, la señal de salida tendrá la misma amplitud de la señal de entrada, mientras que α=1, la amplitud de la señal de salida variará hasta 2 veces el nivel original, y nunca será menor a éste. El algoritmo representado en el programa visual correspondiente se presenta en la figura 6.4.

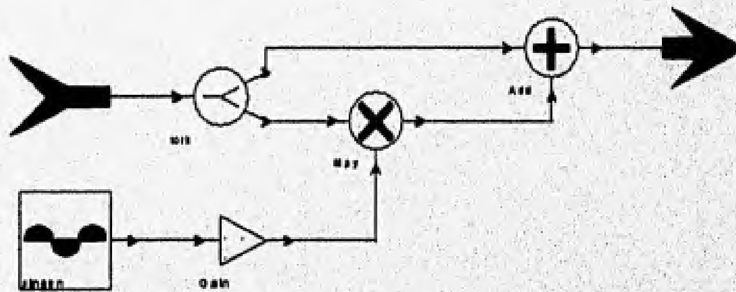


Fig. 6.4

La simulación se realizó con una entrada constante de amplitud igual a 1, la frecuencia de la moduladora a 10 Hz y un factor α de 0.5 (fig. 6.5). La gráfica de la salida (fig. 6.6) permite observar la variación cíclica de la señal de entrada.

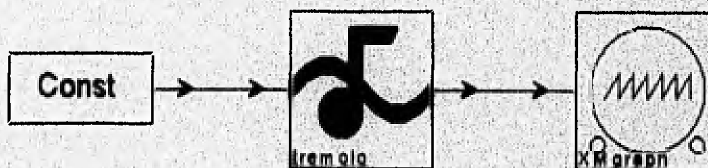


Fig. 6.5

Salida del tremolo a entrada escalon (v=10 Hz, p=0.5)

Y

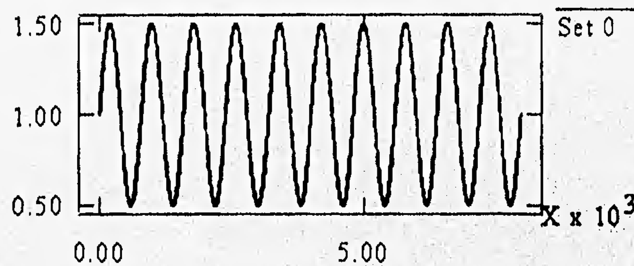


Fig. 6.6

6.3.3 Delay.

La solución de este efecto es la suma de la señal original con la señal retrasada. La representación de este algoritmo ya construido en Ptolemy se presenta en la figura 6.7.

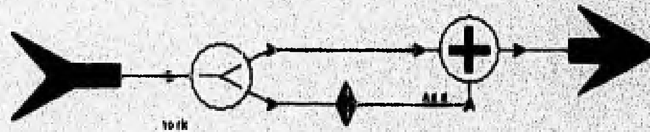


Fig. 6.7

Para simular el comportamiento de la solución, se conectó a la entrada del efecto una señal impulso y a la salida un bloque de graficación (fig. 6.8).

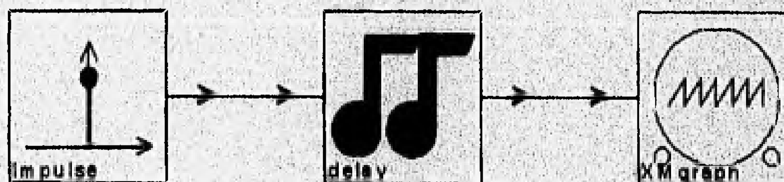


Fig. 6.8

La salida de la ejecución de este programa (fig. 6.3) muestra un impulso seguido de otro impulso de igual magnitud pero retrasado en el tiempo. La simulación se ejecutó para 1000 muestras, siendo el retraso igual a 100 muestras.

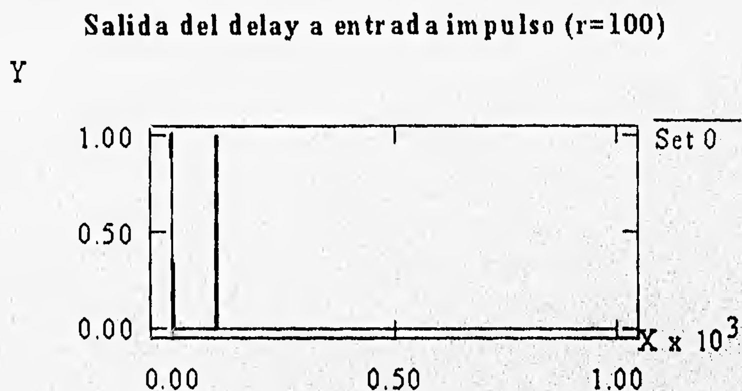


Fig. 6.9

6.3.4 Eco.

El eco, según el análisis, consiste de la suma de la señal original con la señal retrasada, salida que se realimentará para lograr el efecto de repeticiones sucesivamente menores en amplitud. La representación de la solución construida en Ptolemy se da en la figura 6.10.

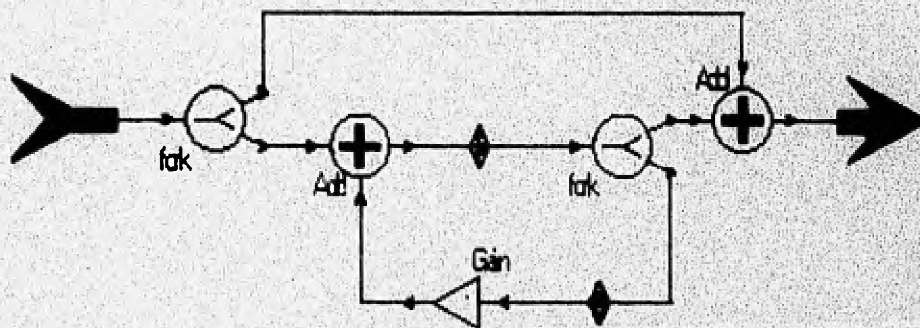


Fig. 6.10

Para simular su comportamiento, se conectaron como entrada una señal impulso y como salida un graficador (fig. 6.11).

Fig. 6.11

En la salida de la simulación (fig. 6.12) puede observarse un impulso seguido de impulsos retrasados que decrecen exponencialmente.

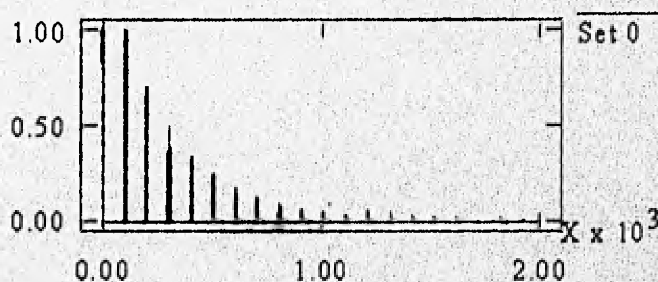


Fig. 6.12

6.3.5 Reverb.

Para la construcción del reverb, se tomó el algoritmo diseñado por Schroeder y citado por Blesser [BLE78]. Este sistema de reverberación consiste de cuatro líneas paralelas de retraso realimentado con valores de retraso en los intervalos de 30 a 45 ms., seguidas de dos filtros paso-todo (allpass filters) en cascada. Por último, se suma a toda la estructura la señal original. Para construir este algoritmo en la herramienta, fue necesario crear en primer lugar un bloque para el retraso realimentado (fig. 6.13), a partir del cual se construyó el filtro paso-todo (fig. 6.14). De esta forma, al crear elementos genéricos, podemos reutilizarlos en nuestros diseños sin necesidad de crearlos cada vez que sean requeridos.

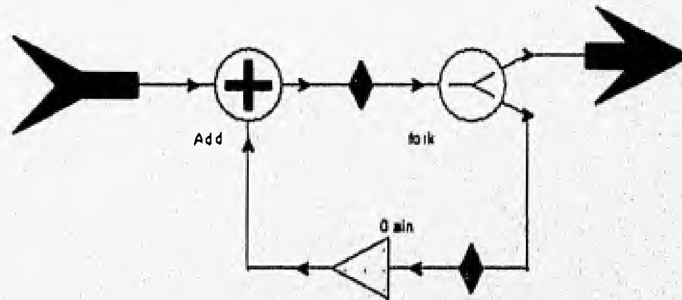


Fig. 6.13

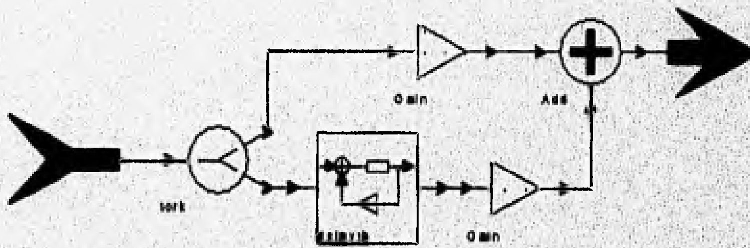


Fig. 6.14

Así, se construyó el algoritmo completo de reverberación (fig. 6.15).

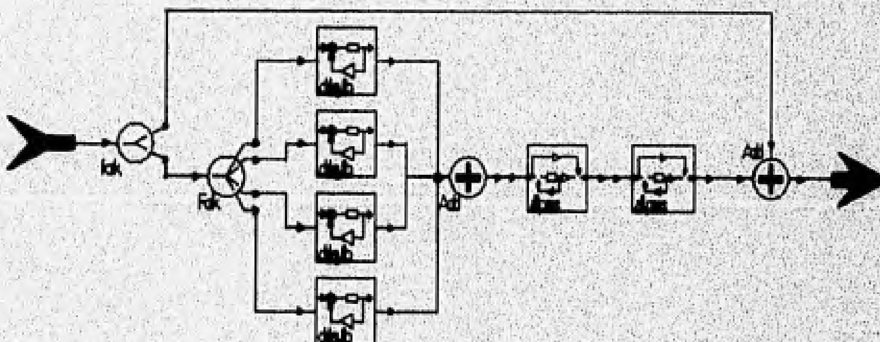


Fig. 6.15

Con el fin de evaluar el comportamiento del reverb, se alimentó a su entrada una señal impulso y se graficó la salida (figs. 6.16 y 6.17). Los valores de los retrasos se fijaron en 30, 35, 40 y 45 ms. para las líneas paralelas y en 5 y 15 ms para los filtros paso-todo Puede observarse en la salida una secuencia de impulsos escalados y distribuidos no-uniformemente. Para lograr respuestas distintas, es necesario sólamente variar los parámetros de retraso y de realimentación (en esta simulación, se tienen valores de retraso que son múltiplos de otros, lo que provoca repeticiones simultáneas cuya suma alcanza una amplitud mayor a la de la entrada).

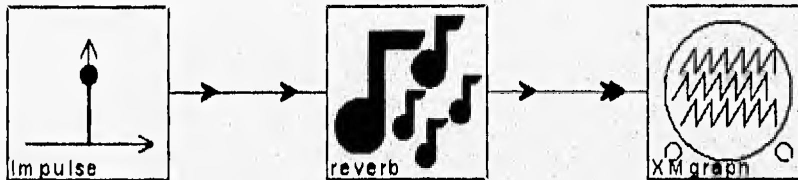


Fig. 6.16

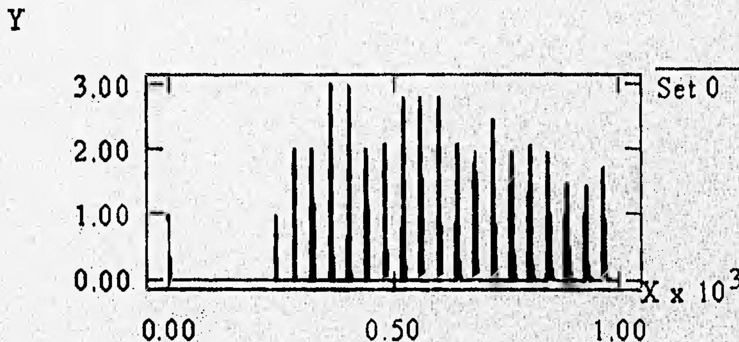


Fig. 6.17

6.3.6 Flanger/Chorus.

Este efecto consiste en la suma de la señal original con la salida una línea de retraso variable. La solución algorítmica se presenta en la fig. 6.18. Como se dijo anteriormente, los intervalos en el retraso definen el tipo de efecto: retrasos menores (0 a 15 ms) y realimentación proporcionan el sonido parecido al motor de un avión a reacción característico del flanger. Retrasos un poco mayores (15 a 25 ms), generan la sensación de estar ante la presencia de más de una fuente de sonido (coro). Ptolemy no cuenta con un elemento gráfico que corresponda a un retraso variable, por lo que no es posible construir un programa visual para simular el algoritmo ni sintetizar el

código a alto nivel. Sin embargo, a partir de la solución algorítmica representada, se realizará la solución software que permita desempeñar el algoritmo y que podrá después integrarse al ambiente gráfico.

Fig. 6.18

6.4 DISEÑO DE LA SOLUCION SOFTWARE.

El siguiente nivel de diseño de la metodología se refiere al diseño de la solución software al problema, es decir, la construcción del código en lenguaje de alto nivel necesaria para realizar las tareas DSP definidas en la solución algorítmica. Para los efectos de trémolo, delay, eco y los bloques necesarios del reverb (retraso realimentado y filtro paso todo), se utilizó el dominio de generación automática de código soportado por Ptolemy. En cuanto a la distorsión, el dominio de generación de código no contaba con los elementos de control utilizados en la simulación del algoritmo (comparador y multiplexor). Por lo tanto, el código correspondiente a este efecto será diseñado sin el auxilio de la herramienta.

Cabe aquí mencionar algunos aspectos básicos de la generación de código en esta herramienta. Los esfuerzos del grupo de desarrollo de Ptolemy de la Universidad de California en Berkeley se han orientado sobre todo a la síntesis automática de código para un sistema interconstruído (embedded) [PIN95]. Esto implica la idea de que en una estación de trabajo se tenga instalada la herramienta, un compilador y se tengan además conectadas una o más tarjetas DSP (DSP cores), con sus correspondientes procesadores dedicados, sus elementos de memoria y sus convertidores A/D y D/A. Todo esto soportado por el sistema operativo y los recursos propios de la estación de trabajo (dispositivos de entrada y salida, memoria, etc.). De esta forma, cuando se ejecuta un programa visual creado dentro del dominio de generación de código de Ptolemy, se sintetizará el código, se compilará, se ligará, se cargará en memoria y se ejecutará como un programa completo. Si el código generado está destinado al procesador DSP de una tarjeta existente, Ptolemy realizará además el control en tiempo real de la arquitectura.

Sin embargo, no todas las aplicaciones están destinadas a un sistema interconstruído de tal naturaleza; de hecho, muchos de los posibles usuarios o beneficiarios de un

sistema DSP requerirán solamente un algoritmo, un programa o una arquitectura específicas que puedan integrarse a otros elementos para conformar un producto terminado.

Teniendo esto en mente, hemos realizado la síntesis de código con el objetivo de que los programas sean más generales y que puedan ser incluidos en sistemas de software de distintas plataformas. Además, el análisis del código generado por la herramienta para ejemplos sencillos puede ser de gran utilidad para entender las bases de la semántica y en el diseño de programas propios que implanten bloques no soportados aún por ésta u otras herramientas, como es el caso del flanger del sistema de producción de efectos especiales en señales de audio.

Los programas visuales creados para la generación de código se construyeron de la forma más general posible, para que el código generado pueda ser tomado como un bloque completo. De acuerdo con las tendencias de modularidad y de encapsulamiento de las metodologías de desarrollo de software como la estructurada y la orientada a objetos, es deseable contar con bloques autocontenidos en lugar de grandes procedimientos. Por ejemplo, según se puede apreciar en el inciso 6.3.5, el reverb incluye varios elementos del mismo tipo, como retrasos realimentados y filtros paso todo. Por lo anterior, se generó el código de estos elementos, que pueden ser reutilizados en otros algoritmos, en lugar de generar el código de la configuración completa del reverb. De hecho, para el filtro paso todo, que incluye dentro de su configuración un retraso realimentado, se sustituyó este bloque por un amplificador.

Según la estructura altamente jerárquica de Ptolemy, un programa visual debe ser un "universo" (debe tener por lo menos una entrada y una salida) para poder ser ejecutado; por lo tanto, se conectó como entrada una constante y como salida un "hoyo negro" (blackhole), que es un elemento que consume muestras de entrada pero no genera muestras de salida (equivale a "aterrizar" la salida).

Los programas visuales se presentan en las figuras 6. 19 a 6.23 (trémolo, delay, eco, retraso realimentado y filtro paso todo).

En el apéndice B se presenta el código generado a partir de la ejecución de estos programas visuales. También en ese apéndice se comentan las características generales del código generado por Ptolemy para este tipo de diseños.

Con el objeto de presentar y utilizar estos efectos en una plataforma distinta, como parte de un sistema de software y diseñar y probar el código de los efectos que no fue posible realizar en Ptolemy (distorsión y flanger/chorus), se realizó una adaptación del código generado por Ptolemy a lenguaje C++. Mediante esta adaptación será posible procesar archivos de audio digital en una computadora PC a través de una tarjeta Sound-Blaster, arquitectura más accesible y común que un sistema interconstruido basado en una estación de trabajo con Ptolemy instalado.

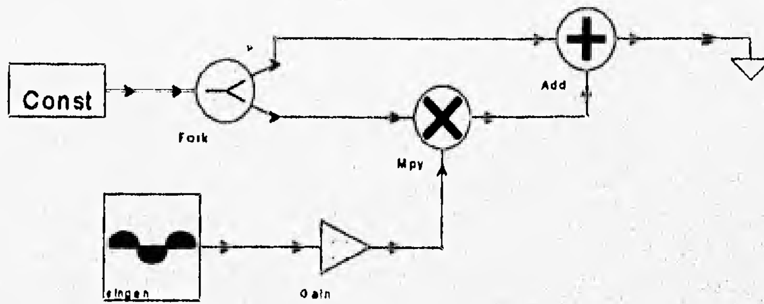


Fig. 6.19

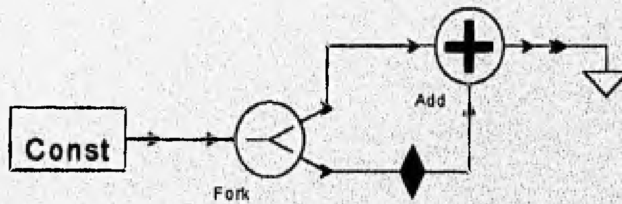


Fig. 6.20

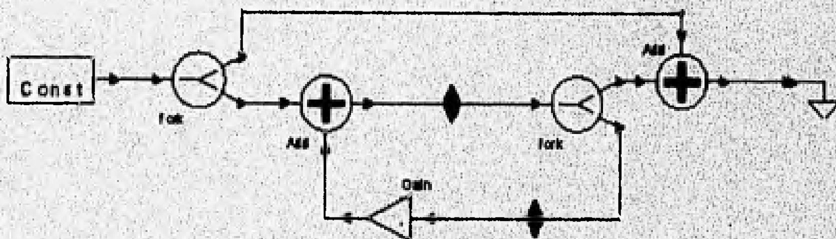


Fig. 6.21

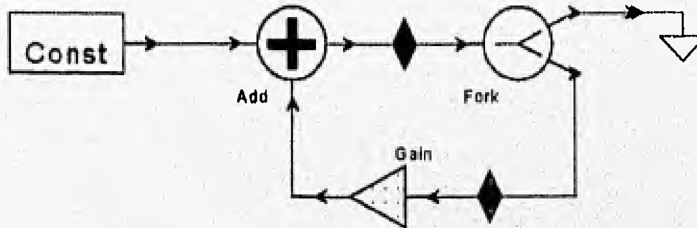


Fig. 6.22

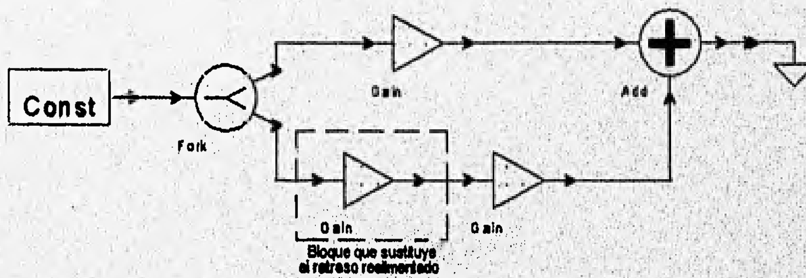


Fig. 6.23

En el apéndice C se presenta el código fuente de los programas realizados en C++, precedido de la estructura general de los mismos, así como la documentación necesaria para utilizarlos en el procesamiento de archivos de audio digital para la tarjeta Sound Blaster Pro.

6.5 DISEÑO DE LA SOLUCION MICROCODIGO.

El nivel de alcance en el proceso de diseño e implantación lo constituye la solución microcódigo. Según se especificó en el alcance, la realización de esta actividad será realizada con fines ilustrativos. El apéndice E muestra el uso de una herramienta como Ptolemy en la generación automática de código de bajo nivel a partir de la representación de los algoritmos en un programa visual.

6.6 PRUEBA Y REFINAMIENTO DEL PROTOTIPO.

La prueba de la solución generada visualmente en Ptolemy para los efectos de delay, eco, distorsión, trémolo y reverb se realizó conectando a la entrada del bloque correspondiente una estrella que lee un archivo de audio en formato au (codificado según el algoritmo PCM ley-μ) y genera como salida una muestra entera de ocho bits. La galaxia del efecto procesa la muestra y la salida se conecta a la entrada de una estrella que la codifica nuevamente a formato au, reproduciendo el archivo resultante en el altavoz de la estación de trabajo. En la figura 6.24, se muestra como ejemplo el universo construido para probar el eco.

Fig. 6.24

En la verificación del flanger/chorus, se utilizó un archivo de audio en formato voc (característico de las tarjetas Sound Blaster), el cual fue procesado con el programa diseñado para tal efecto. El archivo de salida fue reproducido mediante el software propio de la tarjeta Sound Blaster Pro.

La validación final de los efectos, tanto los generados en Ptolemy como el flanger/chorus generado en C++, está definida por la percepción auditiva, puesto que las simulaciones de los programas visuales (gráficas de salida) se habían comportado conforme a lo previsto en la especificación y hasta este punto la solución podía considerarse como correcta. A continuación se presentan los comentarios que arroja esta validación:

El eco, el delay, el trémolo y el reverb cumplen satisfactoriamente las especificaciones, al desempeñar el efecto sonoro deseado sin presentar ruido ni distorsionar la señal. Los parámetros definidos para cada uno de los efectos anteriores permiten generar distintas variaciones en el sonido producido por éstos, las cuales serán determinadas por el usuario.

En cuanto al flanger, el sonido producido en realidad se asemeja al motor de un avión a reacción o jet, y gracias a los distintos parámetros del efecto, se pueden generar muchas variantes del sonido producido. Sin embargo, se detectaron pequeños ruidos semejantes a un "click", los cuales pueden ser producidos por la naturaleza variable del retraso, que no varía en forma continua sino discreta.

La distorsión, al provenir de una acentuada amplificación de la señal de entrada, aumenta también considerablemente el nivel del ruido a la salida. Por lo tanto, se aprecia la distorsión pero no se tiene una señal muy "limpia" (en términos musicales). Para mejorar el efecto, podemos agregar al algoritmo un filtro de atenuación del ruido antes de la amplificación, además de agregar una etapa que redondee o suavice los bordes de la señal recortada.

6.6 OPERACIÓN.

Si bien los efectos diseñados no constituyen un producto terminado, pueden ser utilizados como parte de un sistema de software y con fines de simulación. Los programas para plataforma PC se encuentran documentados y probados. La guía para su utilización, así como una explicación de su estructura y funcionamiento se encuentra en el apéndice C.

Por otra parte, los efectos que fueron construidos en Ptolemy se encuentran integrados en una paleta propia. Se han creado iconos propios para ellos y constituyen una pequeña pero útil herramienta en la simulación de sistemas de procesamiento de audio digital. En la figura 6.25 se muestra la paleta que contiene los iconos de las galaxias creadas. Puede observarse que, además del delay, el eco, el trémolo, la distorsión y el reverb, se tienen los iconos del retraso realimentado y el filtro paso todo, que pueden ser reutilizados en otros diseños. En el apéndice D se muestran los parámetros definidos para cada galaxia y su relación con los bloques del programa visual que representa al algoritmo.

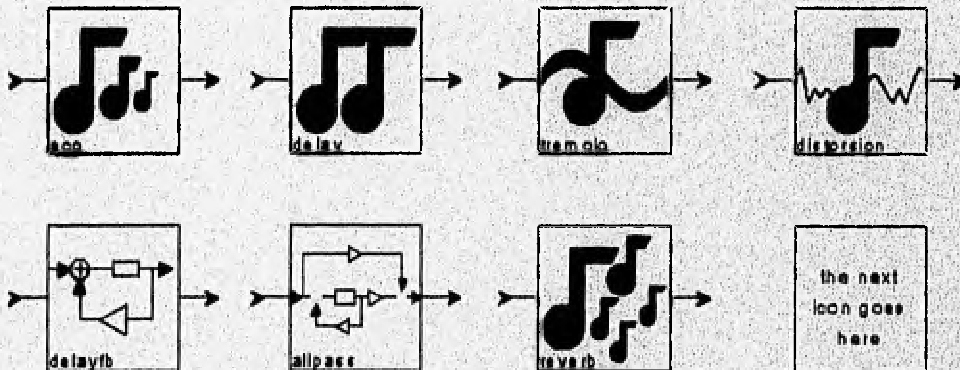


Fig. 6.25

CONCLUSIONES.

El objetivo del capítulo 6 se cumplió satisfactoriamente, ya que a lo largo de éste, pudimos ver como el uso de una metodología de desarrollo de sistemas DSP es de suma importancia, desde plantear el problema en lenguaje natural, definir especificaciones y requerimientos, conviniendo que puede tener solución mediante un sistema DSP, hasta el uso de herramientas de software, que permitieron simular el algoritmo y generar el código de manera sencilla y eficiente.

Por otro lado, la realización de este capítulo fue sumamente provechosa y enriquecedora, ya que con la construcción de los programas visuales y la generación de código de cada efecto, pudimos conocer y aprovechar mucho mejor las bondades y recursos de una herramienta como Ptolemy, dejando además preparados los algoritmos para su reuso en otras aplicaciones. Sin embargo, algunos problemas externos, como la falta de un compilador propio en la instalación de Ptolemy, nos impidieron desarrollar código para crear la estrella del flanger, además de que la falta de algunos elementos en el dominio de generación de código de alto nivel (CGC) nos impidieron generar el código de la distorsión.

REFERENCIAS.

[AND84] Anderton, Craig. *Guitar Gadgets*. Amsco, Estados Unidos, 1984.

[BLE78] Blesser, Barry y Kates, James M. "Digital Processing in Audio Signals" en *Applications of Digital Signal Processing*. Alan V. Oppenheim, Editor. Prentice-Hall, Estados Unidos, 1978.

[BUC94] Buck, Joseph et. al. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". Department of Electrical Engineering and Computer Science, University of California, Berkeley, Estados Unidos, 1994.

[PIN95] Pino, Jose Luis et. al. "Software Synthesis for DSP using Ptolemy". *Journal of VLSI Signal Processing*, vol. 9, pp. 7-21. Kluwer Academic Publishers, Estados Unidos, 1995.

Conclusiones y Perspectivas.

Los algoritmos del Procesamiento Digital de Señales constituyen un conjunto de herramientas que mejoran y amplían la recepción, el almacenamiento, la modificación y la transmisión de la información contenida en señales en el tiempo discreto. La realización práctica de estos algoritmos requiere del diseño e implantación de programas en alto y bajo nivel, de arquitecturas de procesamiento e incluso el desarrollo de un nuevo circuito integrado. A esta realización la llamamos *Sistema de Procesamiento Digital de Señales* o *Sistema DSP*.

A pesar del desarrollo que en forma particular han seguido los elementos que constituyen el desarrollo de un sistema DSP (algoritmos, programas a alto y bajo nivel, arquitecturas de procesamiento), el diseñador se enfrenta ante la carencia de las ligas metodológicas que le permitan desempeñar el proceso de desarrollo de un sistema DSP en forma organizada y coherente. Esto quiere decir que existe una discontinuidad (que en ocasiones parece abismal) entre la transición del algoritmo matemático expresado en el pizarrón y el programa que desempeñe ese algoritmo, por mencionar un ejemplo.

Por otra parte, se han creado una gran variedad de herramientas de software que permiten desempeñar automáticamente muchas de las labores que tradicionalmente habían sido realizadas manualmente, y que consumían gran parte del tiempo y los recursos asignados a un proyecto.

Teniendo como base estos conceptos, en este trabajo se ha propuesto una Metodología de Diseño de Sistemas de Procesamiento Digital de Señales a partir de la revisión de los siguientes elementos:

- las bases teóricas de los sistemas discretos y el procesamiento digital de señales
- las metodologías de desarrollo de software y de diseño de arquitecturas de procesamiento
- las herramientas de software de desarrollo dedicadas a los sistemas DSP

Esta metodología sigue un proceso en el que inicialmente se plantea un problema a ser solucionado. Posteriormente, se analiza ese problema con el fin de contar con una especificación de la solución, tomando en cuenta los requerimientos y el entorno del problema. A la fase de análisis sigue el diseño de la solución, que hemos integrado bajo el paradigma del prototyping o desarrollo de prototipos, en donde a partir de la especificación se construye un prototipo de solución que se va evaluando y mejorando hasta ajustarse a los requerimientos definidos en el análisis. En esta fase, es de vital importancia el aprovechamiento de las capacidades de las herramientas de software, tanto en la simulación de los algoritmos como en la construcción de la solución en sus

distintos alcances (algorítmica, software, microcódigo y hardware). Una vez conseguida una solución satisfactoria, se genera la documentación necesaria para la operación de la misma. Estas características hacen que la metodología pueda ampliar su campo de acción, realizando las adaptaciones adecuadas, a otros campos relacionados, tales como las comunicaciones, el control y el reconocimiento de patrones, por mencionar algunos.

Para probar la validez y utilidad de la metodología, se aplicó en el diseño e implantación de un sistema de producción de efectos especiales en señales de audio digital. De esta forma, a partir del planteamiento del problema, se desarrollaron las actividades de la fase de análisis, comenzando con una investigación sobre los principios del audio digital y de los efectos de audio. Los principios de audio digital nos permitieron entender los elementos y conceptos fundamentales de un esquema de procesamiento de audio digital. Por otra parte, la investigación sobre efectos especiales de audio y la forma en que tradicionalmente se han realizado con circuitos analógicos, permitió generar una especificación a alto nivel de la solución al problema planteado. Esta especificación hizo posible una concepción clara y sencilla de la solución que algorítmicamente había que dar al problema. Una vez diseñados los algoritmos básicos, se hizo uso de una herramienta de desarrollo (Ptolemy) que nos sirvió para simular los algoritmos diseñados en un ambiente visual, validando rápidamente el correcto funcionamiento de los algoritmos y detectando en forma temprana los posibles errores de diseño, tanto en forma teórica como perceptual.

Una vez evaluada la solución algorítmica, se utilizó la herramienta para construir las bases del siguiente nivel de alcance: la solución software. A partir de programas visuales, se sintetizó el código en alto nivel (C) para cada efecto. El código generado por la herramienta fue utilizado para construir una aplicación final de software independiente de la herramienta de desarrollo y destinada a una plataforma distinta, en la cual se debieron considerar los elementos básicos de una aplicación de este tipo (manejo de archivos, interacción con el usuario, etc.). Para la construcción de este programa, se siguió la metodología orientada a objetos y el lenguaje C++, comprobándose la portabilidad del código generado por Ptolemy. Se seleccionó la metodología orientada a objetos porque dos de sus conceptos más importantes, como el encapsulamiento y el polimorfismo, hacen casi transparente la integración de código en objetos autocontenidos.

Como última actividad de la fase de diseño, se utilizó Ptolemy en la generación del microcódigo de algunos de los efectos de audio. Esta actividad se realizó con el fin de mostrar la potencialidad de la herramienta, sin tener por objetivo la implantación física (hardware) de la solución microcódigo (microsoftware).

De la realización de este trabajo podemos concluir que:

- La tendencia actual y futura indica la presencia de herramientas de software cada vez más poderosas aplicables en el desarrollo de sistemas DSP. Estas herramientas tienen como uno de sus objetivo primordiales la automatización e integración de

actividades que han consumido gran cantidad de tiempo y recursos, como la programación a alto nivel y la programación de microcódigo. Ante esta tendencia, podemos decir que las actividades se están especializando, es decir, el desarrollo de algoritmos más eficientes y numéricamente más robustos, el desarrollo de software, la programación de microcódigo, el diseño de arquitecturas y de circuitos integrados se lleva a cabo en grupos cada vez más especializados y cuyo trabajo se integra en las herramientas.

♦ La utilización de la metodología y de las herramientas de desarrollo permiten la creación de mejores soluciones con un uso más eficiente de los recursos materiales y humanos. Esto se debe a que durante un proceso ordenado se generan menos errores en la transición de un paso de la realización al siguiente, aumentando la rapidez y economía en el desarrollo. Asimismo, facilita la correcta especificación de la solución y sus restricciones y proporciona la documentación necesaria para una operación más fácil y eficiente.

♦ La metodología *facilita* el diseño de la solución, ya que se parte de un análisis que proporciona una especificación y una definición de requerimientos.

♦ La solución generada siguiendo la metodología es *mejor*, pues el uso de una herramienta permite la detección temprana de errores o mejoras en el diseño.

♦ La solución desarrollada mediante la aplicación de la metodología y el uso de una herramienta de desarrollo es fácilmente adaptable e integrable en una aplicación independiente de la herramienta, contando con la documentación adecuada.

♦ Las herramientas de desarrollo pueden desempeñar un papel fundamental en el aprendizaje y capacitación teórica y práctica de los alumnos de ingeniería en asignaturas relacionadas al análisis de sistemas y señales, control, comunicaciones, procesamiento digital de señales, y procesamiento digital de imágenes, entre otras.

♦ El desarrollo de herramientas innovadoras que automaticen actividades características del diseño e implantación de distintos tipos de sistemas, abre amplias y variadas líneas de investigación para los ingenieros en computación, como interfaces gráficas, sistemas heterogéneos y la síntesis de código y microcódigo a partir de representaciones visuales (no sólo para algoritmos y arquitecturas de procesamiento, sino para diversos tipos de aplicaciones como sistemas de bases de datos o compiladores).

♦ En particular, las herramientas de desarrollo orientadas al procesamiento de señales y las comunicaciones, como Ptolemy, Matlab/Simulink o Khoros, basan su operación en la representación gráfica de un algoritmo. Dentro de la metodología propuesta, esto implica el haber concebido la solución algorítmica. En la actualidad se trabaja para que las herramientas incluyan el auxilio, e incluso la automatización, en las fases de planteamiento y análisis del problema, así como en la generación automática de la solución algorítmica. En estas perspectivas, la inteligencia artificial, el

reconocimiento de patrones y el procesamiento de lenguaje natural tienen un papel decisivo.

Apéndice A:

Guía de Utilización de Ptolemy.

INTRODUCCION.

Este apéndice tiene como finalidad proporcionar al lector una guía práctica del uso de Ptolemy. Parte de la información aquí resumida se obtuvo del manual del usuario [PUG95], que constituye la documentación práctica más importante, además de la experiencia adquirida durante el trabajo con el software durante la realización de esta tesis.

Ptolemy es una herramienta de diseño para sistemas heterogéneos que soporta simultáneamente diversos modos de cómputo para la simulación y desarrollo de prototipos. Entre sus aplicaciones más importantes se encuentran:

- diseño de redes multimedia
- software interconstruido de tiempo real
- diseño paralelo de software y hardware
- control y procesamiento de llamadas en redes de telecomunicaciones
- desarrollo rápido de prototipos de servicios de telecomunicaciones
- simulación de hardware de modos múltiples (mixed-mode)
- sistemas de procesamiento de señales y control

Está diseñado siguiendo los lineamientos de la programación orientada a objetos y permite la programación visual en una interfaz gráfica de usuario. Fue desarrollado por el Departamento de Ingeniería Eléctrica y Ciencias de la Computación de la Universidad de California en Berkeley. Trabaja sobra la plataforma UNIX (fue desarrollado en las estaciones de trabajo Sparc de Sun Microsystems), y la interfaz gráfica es soportada por el sistema X windows.

A.1 INFORMACIÓN SOBRE ADQUISICIÓN DEL SOFTWARE E INSTALACION.

Ptolemy puede ser adquirido por tres medios: transferencia vía ftp, World Wide Web y correo convencional. La dirección electrónica para obtenerlo vía ftp es: `ptolemy.eecs.berkeley.edu`. La dirección de WWW es: `http://ptolemy.eecs.berkeley.edu/`. Para obtener información sobre la

adquisición mediante el correo convencional, la dirección de email es: `ilpsoftware@eecs.berkeley.edu`.

Los requerimientos para la instalación completa de Ptolemy son:

- 110 MBytes de almacenamiento secundario
- 8 Megabytes de memoria RAM física
- una estación de trabajo de alguna de las siguientes arquitecturas: HP, SGI Irix5.2, Sun 4 con Solaris 2.4 o SunOS 4.1.3

A.2 CONCEPTOS BASICOS.

La característica fundamental que destaca a Ptolemy de otras herramientas de su tipo es la heterogeneidad, es decir, la posibilidad de trabajar con distintos modelos de cálculo, como el SDF (Sinchronous Data Flow) o Flujo de Datos Síncrono, utilizado para los sistemas DSP, el DE (Discrete Event) o Eventos Discretos, utilizado en la modelización de redes de comunicaciones. De esta forma, en el modelo SDF un programa visual estará destinado a la simulación, mientras en el dominio CG (Code Generation), el objetivo será la síntesis de código.

Por otra parte, siguiendo uno de los conceptos fundamentales de la programación orientada a objetos, Ptolemy posee una estructura altamente jerárquica. De esta forma, se tiene una clase fundamental: el bloque (block), de la que se derivan la estrella (star), la galaxia (galaxy) y el universo (universe). Una estrella es un elemento atómico o indivisible de Ptolemy dentro de un dominio. Un conjunto de estrellas define a una galaxia, que si es ejecutable se convierte en un universo. Los datos de distintos tipos se denominan partículas (particles). Además, un universo creado en un dominio puede ser incluido en un segundo universo creado en otro dominio mediante el concepto de agujero de gusano (wormhole), abstracción que permite la heterogeneidad y la convivencia entre los distintos modos de cálculo.

A.3 LA INTERFAZ GRÁFICA DE USUARIO Y LOS DEMOS DE PTOLEMY.

A.3.1 Inicio.

Ptolemy incluye una interfaz gráfica de usuario denominada pigi (Ptolemy Interactive Graphical Interface) soportada por X windows. Una manera adecuada de conocer las potencialidades y este ambiente de trabajo de Ptolemy es conocer las demostraciones (demos) incluidos con el software. Para ello, es necesario iniciar una sesión bajo el nombre de usuario `ptolemy` en la estación de trabajo donde se encuentre instalado el

software, cambiarse al directorio `ptolemy/demo` y desde ahí ejecutar pigi en el background.

```
cd $PTOLEMY/demo

pigi &
```

Aparecerán tres ventanas: una correspondiente a la consola del vem (editor gráfico interactivo), una ventana de bienvenida y una ventana llamada init.pal. Esta última es una paleta que contiene iconos correspondientes a los demos de los distintos dominios (fig. A.1). Se podrá observar que estos iconos tienen un borde púrpura, lo que indica que son paletas con más iconos dentro. Si el icono tiene un borde negro, se trata de un universo (programa visual ejecutable); si el borde es verde, se trata de una galaxia, y si el borde es azul, se trata de una estrella, que es el elemento básico con que se construyen las aplicaciones.

En la paleta de demos del dominio SDF se encuentran más paletas, correspondientes a demos llamados básicos, demos de procesamiento de señales, de procesamiento de imágenes, etc. Dentro de los demos básicos (fig. A.2) se encuentra uno llamado sinMod, que modula una señal senoidal con otra senoidal. Para tener acceso a él es necesario abrir la ventana de la paleta de demos SDF, luego abrir la ventana de los demos básicos (basic), y finalmente abrir la ventana del universo sinMod (fig. A.3). Puede observarse la estructura altamente jerárquica que sigue la interfaz gráfica.

Para abrir una ventana, tiene que hacerse focus sobre el icono correspondiente y oprimir la tecla i (*look-inside*). Esta tecla es el acelerador de una opción del menú de comandos pigi asociado a cada ventana. Los comandos más importantes de este menú se muestran en la tabla A.1. Este menú aparece oprimiendo shift y el botón medio del mouse.

Cuando se abre un universo, la ventana correspondiente se llama plantilla (facet), y el contenido es el programa visual que define la aplicación. El contenido de un universo son estrellas y/o galaxias. En el caso del modulador senoidal (sinMod), se tienen dos galaxias: la que genera una señal senoidal y la que constituye el modulador, además de una estrella que grafica la salida. Las galaxias se denominan así porque su contenido son estrellas, pero no están destinadas a ser ejecutadas. Por ejemplo, el generador senoidal está constituido por una estrella que genera una secuencia creciente de números (una rampa), a la que se ha conectado una función no lineal que computa el seno del número que tenga a la entrada. Si se desea observar el contenido de esta galaxia, debe hacerse focus sobre el icono y ejecutar el comando look-inside.

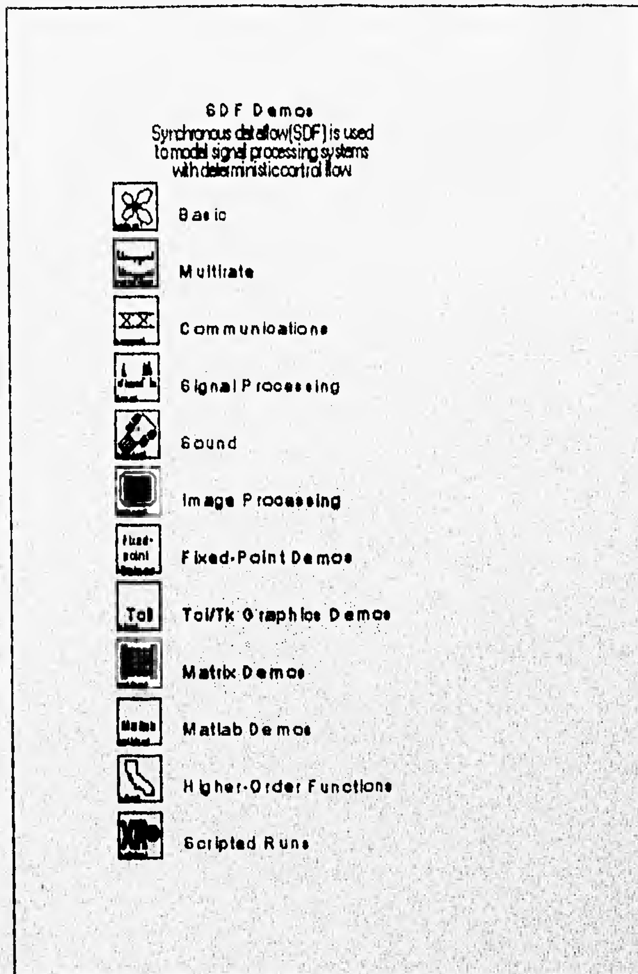


Fig. A.1

A.3.2 Ejecución y cambio de parámetros.

Para correr el universo, debe ejecutarse el comando *run* (acelerador: R). Aparecerá una caja de diálogo llamada panel de control de la ejecución, donde para el dominio SDF podrá especificarse el número de iteraciones que será ejecutado cada bloque. Como salida de la ejecución tendremos la gráfica de una onda senoidal modulada por otra senoidal (fig. A.4). Para cambiar los parámetros de una estrella o galaxia, debe hacerse focus sobre el icono que la represente y ejecutar *edit-params* (acelerador: tecla e), tras lo cual aparecerá una caja de diálogo donde podrán especificarse nuevos valores para los parámetros. Por ejemplo, en el caso de este demo, algunos de los parámetros que pueden cambiarse son la frecuencia de muestreo, la frecuencia y la fase del generador senoidal.

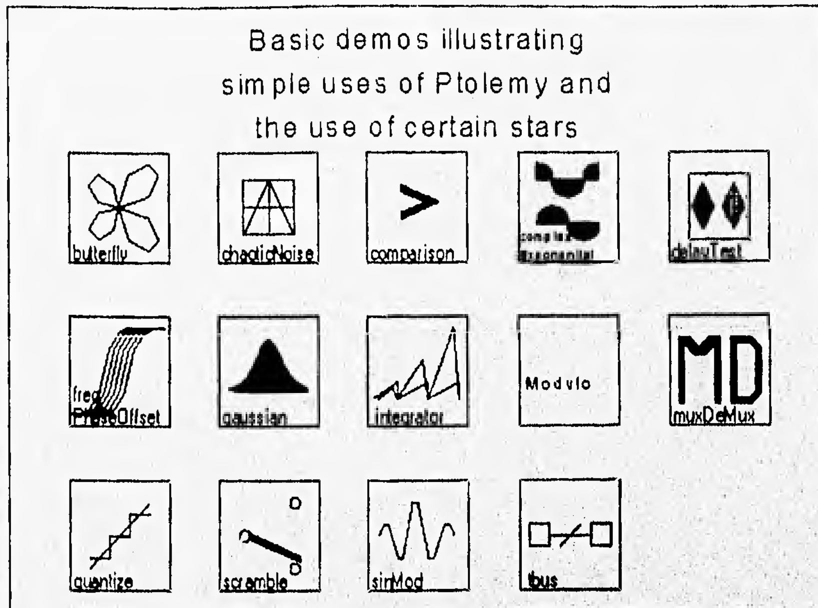


Fig. A.2

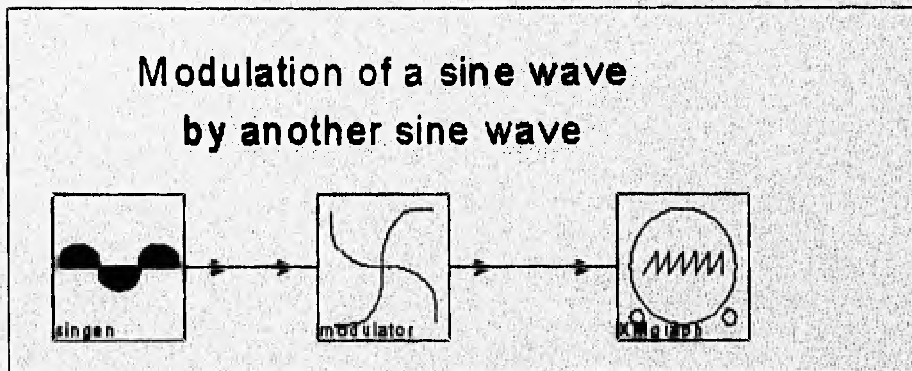


Fig. A.3

Menú	Encabezado	Comando	Tecla	Descripción
pigi	Edit	edit-params	e	cambia los parámetros de una estrella, una galaxia o un universo
	Window	look-inside	i	abre un icono para ver su definición
		open-palette	O	abre una de las paletas de bloques
		open-facet	F	abre una paleta arbitraria (galaxia, universo, etc)
	Exec	run	R	ejecuta un universo
	Other	man	M	abre la página del manual correspondiente a una estrella
		profile	,	despliega un breve resumen del funcionamiento de una estrella
		show-name	n	despliega el nombre de un icono

Tabla A.1

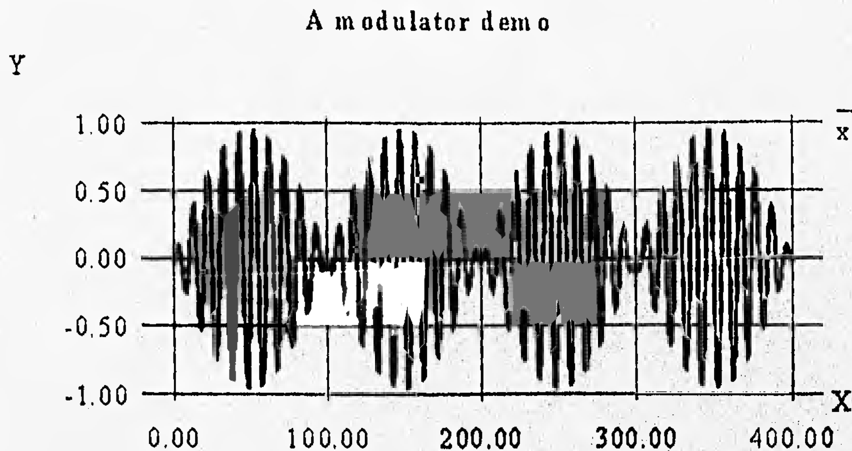


Fig. A.4

Para obtener documentación básica sobre una estrella o una galaxia, debe hacerse focus sobre ellas y ejecutarse los comandos *man* y *profile*. El primero despliega el texto del manual correspondiente al bloque, mientras el segundo explica su funcionamiento básico.

De la misma forma pueden explorarse los distintos demos con que cuenta Ptolemy.

A.4 CREACIÓN DE UNIVERSOS.

A.4.1 Inicio.

Para explicar el procedimiento de creación de universos, se dará un resumen de los pasos generales y las indicaciones básicas para crear un universo que sume dos señales senoidales y despliegue gráficamente el resultado.

Como podrá recordarse, para ejecutar los demos se ejecutó pigi desde el directorio `ptolemy/demo`. Para crear un universo propio, deberá crearse un directorio donde se almacenarán nuestras aplicaciones y desde ahí ejecutar pigi. Por ejemplo:

```
md ptolemy/examples
cd ptolemy/examples
pigi &
```

Cuando se ejecuta pigi, siempre abre la ventana init.pal que se encuentre en el directorio desde donde se ejecutó. Para un directorio nuevo, pigi crea una nueva ventana init.pal.

A.4.2 Ejemplo de creación de universos.

La ventana abierta en estos momentos se llama init.pal. A partir de ella, se abrirá una plantilla (facet) nueva para crear la aplicación. Para realizar ésto debe ejecutarse el comando *open-facet* del menú pigi y darse el nombre del facet (sumasen, por ejemplo). El facet es el espacio de trabajo donde se crearán los programas visuales. El procedimiento general para realizar esto es el siguiente:

Se abren las paletas que contengan los elementos necesarios para realizar el programa. Las paletas principales del dominio SDF (el más robusto soportado por Ptolemy y el de mayor uso para aplicaciones DSP), así como el tipo de iconos (estrellas y galaxias) que contienen, se detallan en la siguiente lista:

Fuentes de señales (Signal Sources).- Son las posibles entradas a un sistema: señal seno, impulso, rampa, escalón, constante, etc.

Salidas (Signal Sinks).- Son los terminadores de los programas visuales: graficadores, histogramas, reproductores de audio, etc.

Aritmética (Arithmetics).- Son operadores aritméticos básicos: suma, multiplicación, amplificación, integración, etc.

Funciones no-lineales (Non-linear).- Incluye funciones como seno, coseno, sinc, exponencial, etc.

Lógica (Logic).- Operadores lógicos: and, or, xor, nand.

Control (Control).- Manipulan el flujo de las simulaciones: divisores (forks), conmutadores, interruptores, etc.

Procesamiento de Señales (Signal Processing).- Existen algoritmos DSP programados como correlaciones, autocorrelaciones, FFT's, FT, Filtros FIR, Filtros IIR, etc.

Una vez abiertas las paletas necesarias, deben crearse los elementos necesarios para el programa visual en nuestro facet de trabajo. Para la creación y manipulación de estos elementos se hace uso del menú llamado vem. Los principales elementos de este menú se presentan en la tabla A.2.

Para nuestro ejemplo, necesitaremos dos señales senoidales como entrada. Para crear la primera es necesario en primer lugar dibujar un punto en nuestro facet; ésto se realiza haciendo click con el botón izquierdo del mouse en la posición donde se desea

crear el elemento. Posteriormente, habrá que hacer focus sobre la paleta de Signal Sources (fig. A.5), colocar el apuntador del mouse sobre el icono llamado singen y oprimir la tecla c (comando create del menú vem). Aparecerá entonces el icono de la senoidal en nuestro facet. De la misma forma, podremos crear la segunda senoidal, el sumador (Add de la paleta Arithmetic, fig. A.6) y el graficador (Xgraph de la paleta Signal Sinks, fig. A.7). El diseño, como luciría hasta este momento, se presenta en la figura A.8.

Menú	Encabezado	Comando	Tecla	Descripción
vem	System	open-window	o	abre una nueva vista
		close-window	cntrl-d	cierra una ventana
		save-window	S	salva la plantilla (facet)
	Display	zoom-in	z	acercamiento
		zoom-out	Z	alejamiento
		show-all	f	reescalamiento automático del diseño
	Undo	undo	U	deshacer un número de operaciones previas
	Edit	create	c	crea un objeto en el facet
	Selection	delete-objects	D	elimina un objeto seleccionado
		select-objects	s	selecciona uno o más objetos

Tabla A.2

Cada elemento gráfico tiene por lo menos una entrada o una salida, las que es necesario conectar para completar el programa. Esto se realiza haciendo click sobre la flecha de salida de un elemento, sosteniendo el botón y arrastrando el apuntador hasta que éste se encuentre colocado en la entrada del siguiente icono. De esta forma se dibujará una línea cuya creación habrá que confirmar mediante el comando create de vem. El programa visual completo aparece en la figura A.9.

La ejecución del programa visual se realiza mediante el comando *run* de pigi (debe recordarse que aparecerá una caja de diálogo donde habrá que especificar el número de veces que se ejecutará cada estrella). Para nuestro ejemplo, la salida del mismo (fig. A.10) sería una gráfica con una senoidal cuya variable dependiente va de 2 a -2. Esto sucede porque hemos sumado dos senoidales de la misma frecuencia. Para apreciar un cambio en esta salida, podemos cambiar la frecuencia de alguna de las señales de entrada mediante la ejecución del comando *edit-params* de pigi.

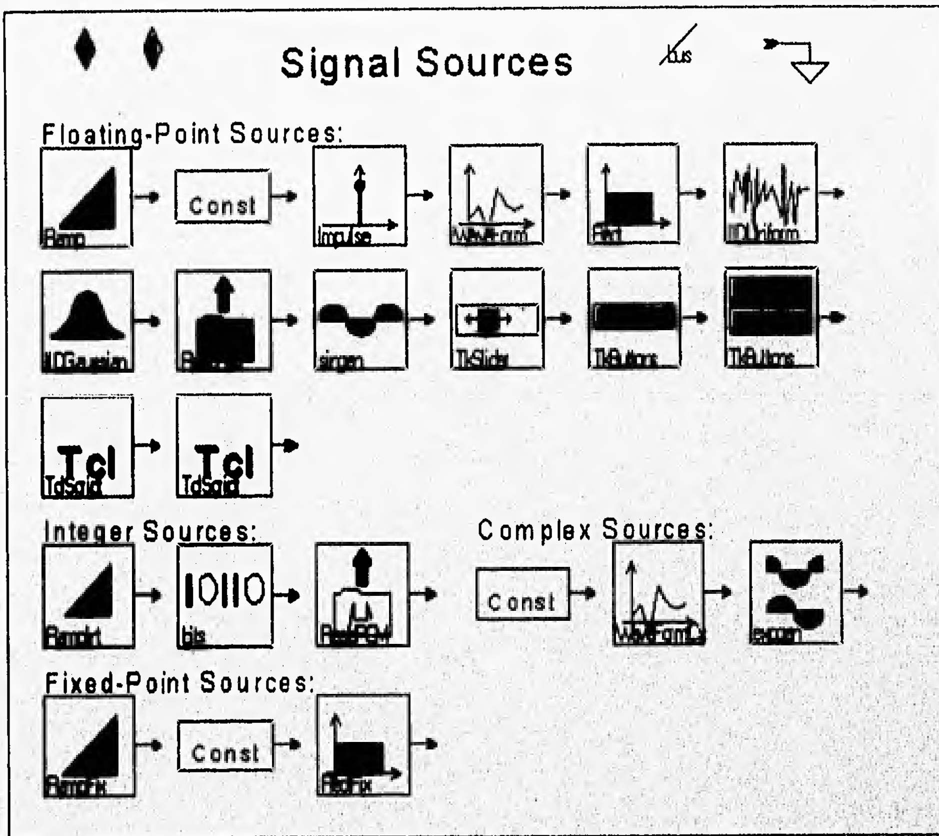


Fig. A.5

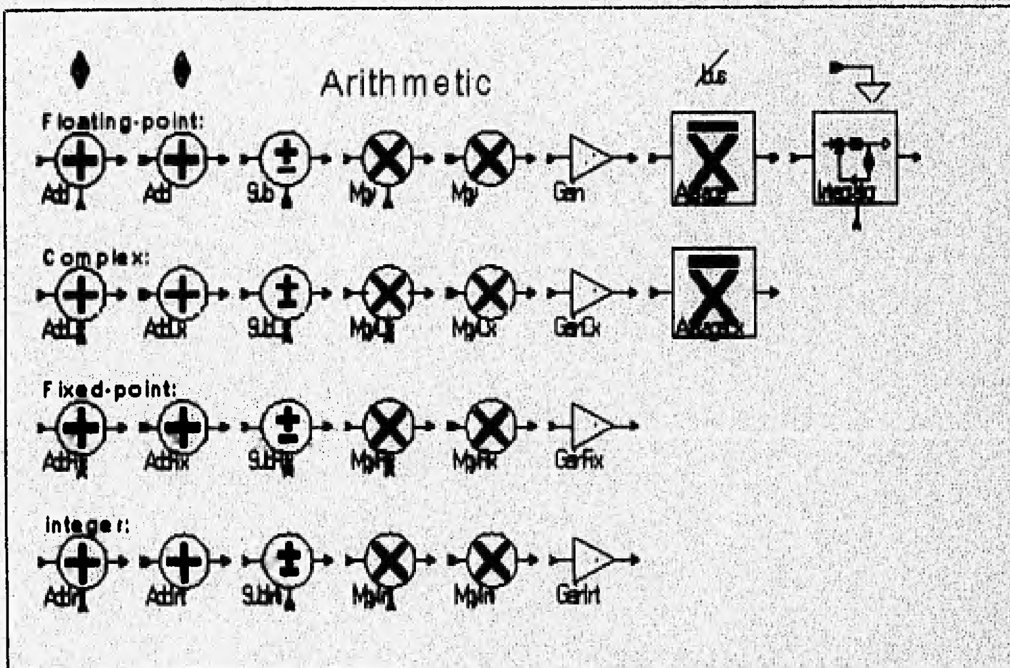


Fig A.6

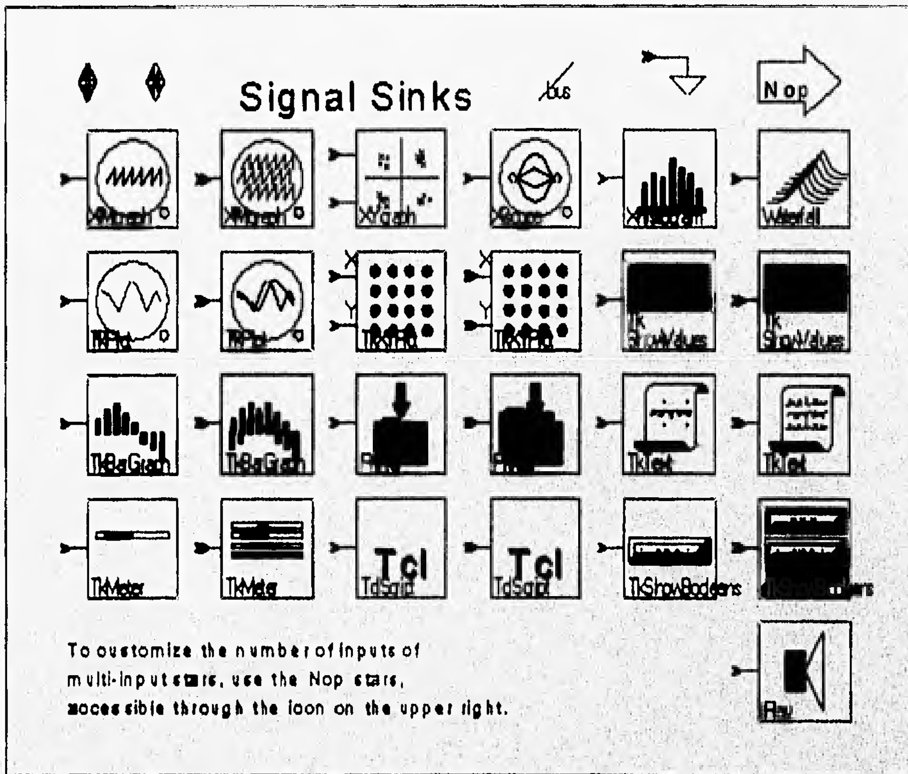


Fig. A.7

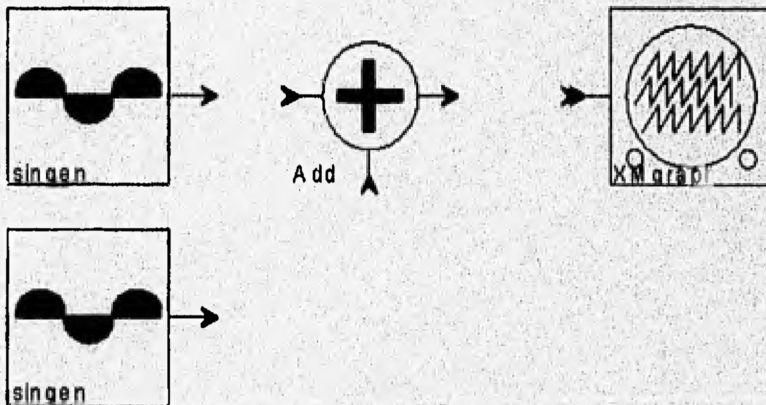


Fig. A.8

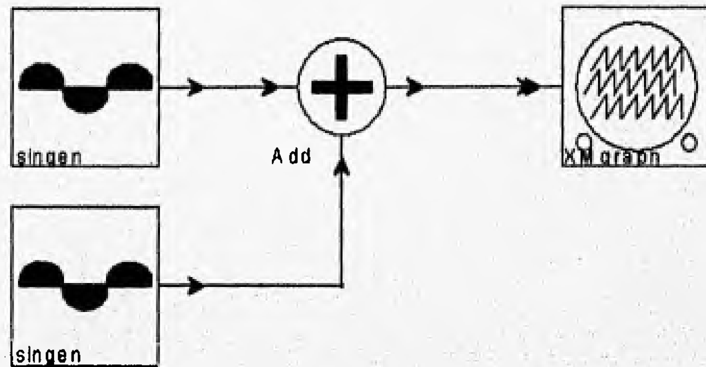


Fig. A.9

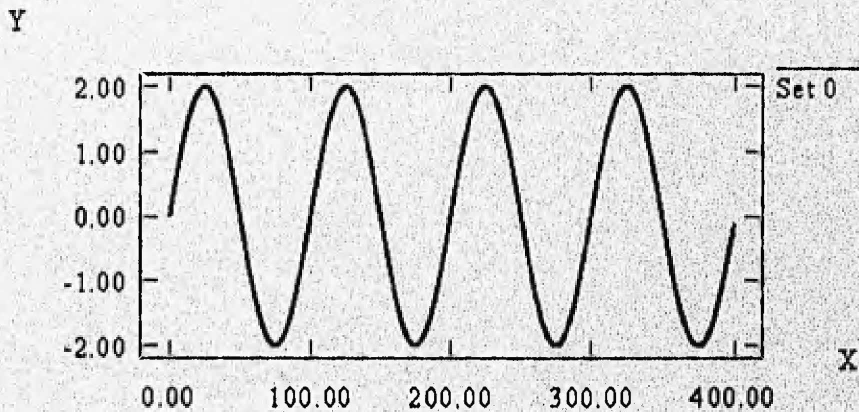


Fig. A.10

Dentro de las estrellas de fuentes de señales, se tiene una que lee un archivo de audio en formato .au (codificado en PCM ley-μ y usado en las estaciones de trabajo de SUN Microsystems) y da como salida las muestras enteras decodificadas, que pueden ser procesadas con las demás estrellas del dominio de simulación. Existe, asimismo, una estrella que toma como entrada valores lineales y los convierte nuevamente al formato .au, para después reproducir el archivo generado en el altavoz de la estación de trabajo. En la figura A.11 se presenta un universo construido con dichos elementos. En este universo, la salida de la estrella que lee el archivo .au es amplificada y luego

alimentada a la estrella que codifica y reproduce; podemos decir que se trata de un amplificador digital básico.

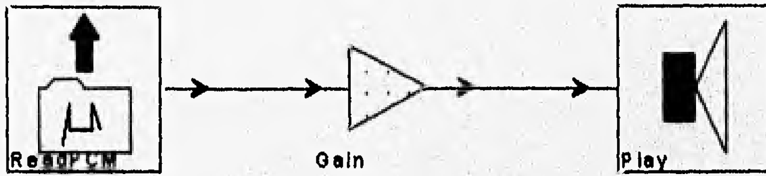


Fig. A.11

REFERENCIAS.

[PUM95] *Ptolemy User's Manual.* College of Engineering; Department of Electrical Engineering and Computer Science, University of California at Berkeley, Estados Unidos, 1995.

Apéndice B:
Programas Fuente Generados por Ptolemy.

INTRODUCCION.

En este apéndice se presenta el código de los programas en lenguaje C generados por la herramienta de software Ptolemy a partir de los programas visuales presentados en el capítulo 6. Algunas de las características generales de los programas que es importante mencionar son las siguientes:

- Los programas presentan un encabezado donde se indica el nombre del usuario, la fecha, el objetivo (target) utilizado y el nombre del universo ejecutado para sintetizar el código.

- Se tienen dos directivas de preproceso para la inclusión de un header que contiene las definiciones de las clases de los objetos y la definición de un tipo para el manejo de números complejos.

- Los programas se generan bajo una función `main`; esta opción puede ser modificada editando el objetivo (target) del dominio. Cada programa consiste de un ciclo `for`, cuyo límite está determinado por el número de muestras que se hayan especificado al correr la simulación (ver apéndice A.3.1).

Analizando los programas generados por Ptolemy, podemos observar que los comandos relacionan los distintos bloques mediante sus entradas y sus salidas. De esta forma, la mayoría de las variables tienen nombres como `input_x`, o `output_x`, donde `x` es un entero correspondiente al orden de creación del objeto en el programa visual. Sin embargo, se ha aprovechado el hecho de que la mayoría de las entradas a un bloque son salidas de uno o más bloques para generar un código más eficiente. Así, la mayoría de las líneas de código definen una salida como función de una o más salidas. Las siguientes son observaciones generales sobre el código que generan los bloques más usuales y que utilizamos en nuestros programas visuales. Ptolemy genera, además, comentarios para identificar el código generado por cada bloque o estrella:

- La salida de un amplificador (Gain) está dada por la multiplicación de su ganancia por la salida del bloque anterior conectada como su entrada (`output_2=0.7*output_1)`, donde 0.7 es el valor de la ganancia del amplificador.

- La salida de un bloque sumador (Add) es directamente la suma de las dos entradas al bloque, que son a su vez salidas de otros bloques (`output_3=output_4+output_1`). Lo mismo sucede con un multiplicador (Mpy) (`output_3=output_4*output_1`).

- El código de un separador o divisor de señales (Fork) es simplemente una asignación que iguala entradas o salidas.

- Para una constante (Const), se asigna el valor de la misma a un identificador de salida.

- La salida de un generador senoidal consiste de un valor que se va incrementando en cierta cantidad con cada iteración (Ramp), valor al que se le aplica la función seno (Sin) para obtener la salida.

Otros elementos que merecen especial atención son los retrasos y la realimentación.

- Ptolemy implanta un retraso de *n* muestras mediante un arreglo de *n+1* muestras, que va controlando a través de las iteraciones con tres índices. Para ejemplificarlo, procedamos a detallar a alto nivel lo que sucede con un retraso de 2 muestras, que es un caso muy sencillo.

Llamemos `delay` al arreglo, e `indice1`, `indice2` e `indice3` a los índices de control. La declaración para el arreglo, considerando que debe ser capaz de almacenar 3 muestras sería:

```
float delay[3];
```

Los índices se inicializan de la siguiente forma:

```
indice1=0;
indice2=0;
indice3=1;
```

En el tiempo 0, se almacena la entrada actual en la posición definida por `indice1`, el cual se incrementa y se reinicializa en caso de llegar al límite del arreglo.

```
delay[indice1]=entrada;
indice1+=1;
if (indice1>=3)
   indice1-=3;
```

Si el algoritmo consiste en la suma de la señal de entrada más la señal retrasada dos muestras, la salida en el tiempo `i=0` está dada por:

```
salida=delay[indice2]+delay[indice3];
indice2+=1;
```

```
if (indice2>=3)
   indice2-=3;
indice3+=1;
if (indice3>=3)
   indice3-=3;
```

Puede observarse que `indice2` tiene el mismo valor que tenía `indice1` antes de incrementarse, por lo que en la posición `indice2` se tiene la muestra de entrada. El `indice3` se encuentra una posición más adelante, cuyo contenido por el momento es 0.

De la misma forma, dos muestras más adelante (`i=2`), se almacenará la muestra de entrada en `delay[indice1]` (`delay[2]`), y se obtendrá la salida sumando `delay[indice2]` (`delay[2]`, la muestra recién leída) con `delay[indice3]`, que contiene la muestra almacenada en `i=0`, pues en ese momento `indice3` es igual a 0, después de haberse reinicializado en la iteración anterior.

- En el caso de la realimentación, el código correspondiente consiste en asignar a la entrada del bloque de la línea principal de retraso la suma de la entrada con la salida del bloque de realimentación. El proceso implica un cierto orden: primero se debe generar la salida de la línea de realimentación a partir de la salida total; posteriormente se suma la salida de la realimentación a la entrada general y se almacena esta suma en la línea de retraso; finalmente, se genera la salida total a partir de la línea de retraso (y sumando la entrada, como en el caso del eco, inciso B.3).

A continuación, se presenta el código generado por Ptolemy a partir de los programas visuales de las figuras 6.19 a 6.23.

B.1 CODIGO GENERADO PARA EL TREMOLO.

El programa visual a partir del cual se sintetizó el código, es el de la (Cap. 6, figura 6.18). El número de iteraciones fue 8000, lo necesario para procesar un segundo de señal de voz digitalizada a 8 kHz. La frecuencia de la señal moduladora es de 10 Hz. Esta señal se genera obteniendo el seno (Sin) de un valor que crece con cada iteración (Ramp). La entrada es una constante (Const) igual a 3.0. Posteriormente, se trata de sumas (Add), una multiplicación de señales (Mpy) y amplificadores (Gain).

```
/* User:      ptolemy
   Date:      Tue Apr 30 17:07:14 1996
   Target:    default-CGC
   Universe:  codtremolo */
#include <math.h>

#if !defined(NO_FIX_SUPPORT)
/* include definitions for the fix runtime library */
#include "CGCrtlib.h"
#endif

#if !defined(COMPLEX_DATA)
#define COMPLEX_DATA 1
  typedef  struct  complex_data  {  double  real;  double  imag;  }
complex;
#endif
/* main function */
main() {
    int i_8;
double output_0;
double output_1;
double output_4;
double value_9;
double output_5;
double output_6;
double output_7;
value_9=0.0;
output_0 = 0.0;
output_1 = 0.0;
output_4 = 0.0;
output_5 = 0.0;
output_6 = 0.0;
output_7 = 0.0;
for (i_8=0; i_8 < 8000; i_8++) {
     {  /* star codtremolo.singen1.Ramp1 (class CGCRamp) */
     output_5 = value_9;
     value_9 += 0.00785398163397447;
     }
     {  /* star codtremolo.singen1.Sin1 (class CGCSin) */
     output_6 = sin(output_5);
```

```
	}
	{  /* star codtremolo.Gain1 (class CGCGain) */
output_7 = 0.5 * output_6;
	}
	{  /* star codtremolo.Const1 (class CGCConst) */
	output_4 = 3.0;
	}
	{  /* star codtremolo.Fork.output=21 (class CGCFork) */
	}
	{  /* star codtremolo.Mpy.input=21 (class CGCMpy) */
	output_0 = output_4 *
output_7;
	}
	{  /* star codtremolo.Add.input=21 (class CGCAdd) */
	output_1 = output_4 +
output_0;
	}
	{  /* star codtremolo.BlackHole1 (class CGCBlackHole) */
/* This star generates no code */
	}
} /* end repeat, depth 0*/

}
```

B.2 CODIGO GENERADO PARA EL DELAY.

Este código fue generado para 2 muestras y un retraso de una muestra. La entrada fue una constante igual a 2.0 (Cap. 6, fig. 6.20). El programa consiste del procedimiento de implantación de retrasos mencionado anteriormente. La salida total es la suma de la entrada con la salida de la línea de retraso.

```
/* User:      ptolemy
   Date:      Mon Apr 29 20:37:31 1996
   Target:    default-CGC
   Universe:  coddelay */

#if !defined(NO_FIX_SUPPORT)
/* include definitions for the fix runtime library */
#include "CGCrtlib.h"
#endif

#if !defined(COMPLEX_DATA)
#define COMPLEX_DATA 1
  typedef struct complex_data { double real; double imag; }
complex;
#endif
/* main function */
main() {
    int i_7;
double output_0[2];
int output_4;
double output_1;
int input_1_5;
int input_2_6;
{int i; for(i=0;i<2;i++) output_0[i] = 0.0;}
output_4 = 0;
output_1 = 0.0;
input_1_5 = 0;
input_2_6 = 1;
for (i_7=0; i_7 < 10; i_7++) {
     {  /* star coddelay.Const1 (class CGCConst) */
     output_0[output_4] = 2.0;
     output_4 += 1;
     if (output_4 >= 2)
          output_4 -= 2;
     }
     {  /* star coddelay.Fork.output=21 (class CGCFork) */
     }
     {  /* star coddelay.Add.input=21 (class CGCAdd) */
     output_1 = output_0[input_1_5] +
output_0[input_2_6];
     input_1_5 += 1;
     if (input_1_5 >= 2)
```

```
            input_1_5 -= 2;
        input_2_6 += 1;
        if (input_2_6 >= 2)
            input_2_6 -= 2;
        }
        {  /* star coddelay.BlackHole1 (class CGCBlackHole) */
/* This star generates no code */
        }
} /* end repeat, depth 0*/

}
```

B.3 CODIGO GENERADO PARA EL ECO.

Se sintetizó este código a partir del programa visual (Cap. 6, figura 6.21), ejecutado para 10 muestras, con 5 muestras de retraso y realimentación de 0.7. El detalle más importante a observar consiste en la realimentación. Como puede apreciarse, en primer lugar se obtiene la salida de la realimentación (`output_7`), que en este caso es el valor de la ganancia del amplificador multiplicado por el valor de salida del delay (`output_4[input_10]`).

```
output_7=0.7*output_4[input_10];
```

Lo que se almacena en la linea de retraso es la entrada constante con valor 3.0 (`output_6`) **más la realimentación** (`output_7`):

```
output_4[output_8]=output_6+output_7;
```

por ultimo, la salida final (`output_5`) **es la suma de una entrada constante de valor 3.0** (`output_6`) **con la salida del delay** (`output_4[input_1_9]`).

```
output_5 = output_4[input_1_9] + output_6;
```

```
/* User:      ptolemy
   Date:      Thu May  9 20:32:19 1996
   Target:    default-CGC
   Universe:  codeco */

#if !defined(NO_FIX_SUPPORT)
/* include definitions for the fix runtime library */
#include "CGCrtlib.h"
#endif

#if !defined(COMPLEX_DATA)
#define COMPLEX_DATA 1
  typedef struct complex_data { double real; double imag; }
complex;
#endif
/* main function */
main() {
    int i_11;
double output_4[2];
int output_8;
double output_5;
int input_1_9;
double output_6;
int input_10;
double output_7;
{int i; for(i=0;i<2;i++) output_4[i] = 0.0;}
```

```
output_8 = 0;
output_5 = 0.0;
input_1_9 = 1;
output_6 = 0.0;
input_10 = 0;
output_7 = 0.0;
for (i_11=0; i_11 < 10; i_11++) {
     {  /* star codeco.Gain1 (class CGCGain) */
output_7 = 0.7 * output_4[input_10];
     input_10 += 1;
     if (input_10 >= 2)
          input_10 -= 2;
     }
     {  /* star codeco.Const1 (class CGCConst) */
     output_6 = 3.0;
     }
     {  /* star codeco.Fork.output=21 (class CGCFork) */
     }
     {  /* star codeco.Add.input=22 (class CGCAdd) */
     output_5 = output_4[input_1_9] + output_6;
     input_1_9 += 1;
     if (input_1_9 >= 2)
          input_1_9 -= 2;
     }
     {  /* star codeco.BlackHole1 (class CGCBlackHole) */
/* This star generates no code */
     }
     {  /* star codeco.Add.input=21 (class CGCAdd) */
     output_4[output_8] = output_6 + output_7;
     output_8 += 1;
     if (output_8 >= 2)
          output_8 -= 2;
     }
     {  /* star codeco.Fork.output=22 (class CGCFork) */
     }
} /* end repeat, depth 0*/

}
```

B.4 CODIGO GENERADO PARA EL RETRASO REALIMENTADO.

Este código se sintetizó a partir del programa visual (Cap. 6, figura 6.22), para 10 iteraciones, una línea de retraso de 5 muestras, realimentación de 0.5 y entrada constante de 3.0. El procedimiento es muy parecido al del eco, pero en este caso no se realiza la suma de la entrada constante para generar la salida total `(output_0[output_5])`, que viene directamente de la línea de retraso.

```
/* User:      ptolemy
   Date:      Tue Apr 30 17:22:28 1996
   Target:    default-CGC
   Universe:  coddelayfb */

#if !defined(NO_FIX_SUPPORT)
/* include definitions for the fix runtime library */
#include "CGCrtlib.h"
#endif

#if !defined(COMPLEX_DATA)
#define COMPLEX_DATA 1
  typedef struct complex_data { double real; double imag; }
complex;
#endif
/* main function */
main() {
    int i_8;
double output_0[6];
int output_5;
int input_6;
double output_1;
double output_2;
int input_1_7;
{int i; for(i=0;i<6;i++) output_0[i] = 0.0;}
output_5 = 0;
input_6 = 0;
output_1 = 0.0;
output_2 = 0.0;
input_1_7 = 1;
for (i_8=0; i_8 < 10; i_8++) {
     {  /* star coddelayfb.Gain1 (class CGCGain) */
output_1 = 0.5 * output_0[input_6];
     input_6 += 1;
     if (input_6 >= 6)
          input_6 -= 6;
     }
     {  /* star coddelayfb.Const1 (class CGCConst) */
```

```
     output_2 = 3.0;
     }
     {  /* star coddelayfb.BlackHole1 (class CGCBlackHole) */
/* This star generates no code */
     input_1_7 += 1;
     if (input_1_7 >= 6)
          input_1_7 -= 6;
     }
     {  /* star coddelayfb.Add.input=21 (class CGCAdd) */
     output_0[output_5] = output_2 +
output_1;
     output_5 += 1;
     if (output_5 >= 6)
          output_5 -= 6;
     }
     {  /* star coddelayfb.Fork.output=21 (class CGCFork) */
     }
} /* end repeat, depth 0*/

}
```

B.5 CODIGO GENERADO PARA EL FILTRO PASO-TODO.

Este código es muy simple, pues se trata sólo de sumas y amplificadores. Cabe recordar, según el programa visual (Cap. 6, figura 6.23), que se ha sustituido el bloque correspondiente al retraso realimentado con un amplificador. El objetivo de esto es lograr programas modulares autocontenidos reutilizables y que puedan incorporarse a sistemas de software desarrollados mediante metodologías apropiadas. Se definieron 10 iteraciones y una entrada constante igual a 3.0.

```
/* User:    ptolemy
   Date:      Tue Apr 30 17:42:51 1996
   Target:    default-CGC
   Universe:  codallpass */

#if !defined(NO_FIX_SUPPORT)
/* include definitions for the fix runtime library */
#include "CGCrtlib.h"
#endif

#if !defined(COMPLEX_DATA)
#define COMPLEX_DATA 1
  typedef struct complex_data { double real; double imag; }
complex;
#endif
/* main function */
main() {
    int i_7;
double output_2;
double output_3;
double output_4;
double output_5;
double output_6;
output_2 = 0.0;
output_3 = 0.0;
output_4 = 0.0;
output_5 = 0.0;
output_6 = 0.0;
for (i_7=0; i_7 < 10; i_7++) {
     {  /* star codallpass.Const1 (class CGCConst) */
     output_6 = 3.0;
     }
     {  /* star codallpass.Fork.output=21 (class CGCFork) */
     }
     {  /* star codallpass.Gain1 (class CGCGain) */
output_2 = 1.0 * output_6;
     }
     {  /* star codallpass.Gain2 (class CGCGain) */
output_3 = 2.0 * output_6;
     }
```

```
     {  /* star codallpass.Gain3 (class CGCGain) */
output_4 = 3.0 * output_3;
     }
     {  /* star codallpass.Add.input=21 (class CGCAdd) */
     output_5 = output_2 +
output_4;
     }
     {  /* star codallpass.BlackHole1 (class CGCBlackHole) */
/* This star generates no code */
     }
} /* end repeat, depth 0*/

}
```

Apéndice C:
Programa de Producción de Efectos Especiales en Señales de Audio Digital.

INTRODUCCION.

En el presente apéndice se presenta el desarrollo del programa creado para procesar archivos de audio digital. Dicho programa fue desarrollado bajo los conceptos de la programación orientada a objetos usando el lenguaje C++. Se tomaron en cuenta los programas fuente generados por Ptolemy para el delay, el eco, el tremolo y el reverb, mientras que en el caso de la distorsión y el flanger se implantaron algoritmos propios. Se describirán en forma jerárquica los objetos diseñados, sus variables de instancia y sus funciones miembro, hasta llegar al código de cada efecto, agregando una explicación de su funcionamiento. Al final, se presentará el código completo de un programa que procesa datos de audio digital en formato .VOC, propio de las tarjetas de audio Sound Blaster de la plataforma PC.

El formato .VOC de Cretive Labs para archivos de datos de audio digital consta de un encabezado y de un conjunto de bloques de datos cuyas características serán resumidas en el inciso C. Por el momento, es importante mencionar que la tarjeta Sound Blaster Pro, modelo utilizado en el desarrollo de este programa, maneja datos enteros de 8 bits. De esta forma, los valores pueden ir de 0 a 255, siendo 128 el nivel equivalente a nivel de amplitud 0. Tomando en cuenta esta información elemental, podemos proceder a describir el proceso de diseño del programa.

C.1 CLASE "EFECTO".

Se creó una clase primaria llamado efecto, a partir de la cual se derivarán las demás. Las variables de instancia son una entrada y una salida enteras, así como la frecuencia de muestreo, también entera. El primer método (inicializar) solicita y asigna la frecuencia de muestreo de los datos de entrada. El otro método (obtensalida), suma la muestra de entrada con la muestra de salida generada por el efecto (el 128 se debe al offset característico del formato de 8 bits) y, por último, limita el valor de la salida para que no exceda los límites del formato. Ambos métodos son virtuales, de forma que puedan ser omitidos (overriden) en las clases heredadas. El código de esta clase se muestra en el listado C.1.

Listado C.1

```
class efecto {
public:
   int entrada;
   int salida;
   int frecuencia;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void efecto::inicializar(void)
{
   printf("Frecuencia de muestreo [Hz]: ");
   scanf("%d",&frecuencia);
};

void efecto::obtensalida(void)
{
   salida=128+(entrada-128)+(salida-128);
   if (salida>255)
      salida=255;
   if (salida<0)
      salida=0;
};
```

C.2 CLASE "EFECTORETRASO".

Para los efectos que utilizan una linea de retraso se utiliza un arreglo donde se almacenan las muestras de retraso entre la muestra de entrada y la siguiente muestra de salida. El objetivo de esta clase es definir este arreglo para manejar efectos como delay, eco y flanger. Las variables de instancia definidas son el arreglo de 8000 posiciones enteras y un índice que lo barre, además de los metodos inicializar y obtensalida, al igual que la clase efecto de la cual se deriva o hereda. El método inicializar solicita el valor del retraso en milisegundos, que junto con la frecuencia de muestreo, la cual se obtiene al ejecutar el metodo inicializar de la clase efecto, servirá para calcular el límite del arreglo. Finalmente, inicializa el arreglo con el valor de 128 (nivel de amplitud 0). El método obtensalida incrementa el índice y lo reinicializa si llega al final del arreglo, invocando antes el mismo método de la clase efecto. En el listado C.2 presentamos el código de clase.

Listado C.2

```
class efectoretraso: public efecto {
public:
   int arreglo[8000];
```

```
   int limite,indice;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void efectoretraso::inicializar(void)
{
   int i;
   float tiempo;

   efecto::inicializar();
   printf("Tiempo de retraso (ms): ");
   scanf("%f",&tiempo);
   limite=(frecuencia*tiempo)/1000;
   for (i=0;i<limite;i++)
      arreglo[i]=128;
   indice=0;
};

void efectoretraso::obtensalida(void)
{
   efecto::obtensalida();
   indice++;
   if (indice==limite)
      indice=0;
};
```

C.3 CLASE "EFECTORETRASOPEQ".

Clase derivada de efecto para el conjunto de retrasos que se utilizan en el reverb. Los retrasos típicos del reverb son menores a los retrasos del delay, el eco y el flanger. La única diferencia con respecto a efectoretraso es la inicialización del arreglo (en este caso esta limitada a 2000 muestras). Esto es necesario para no saturar la memoria al crear los 6 retrasos (4 realimentados y 2 filtros paso-todo) necesarios para el reverb. El método inicializar y el método obtensalida son idénticos a los métodos de la clase efecto. En el listado C.3 se muestra el código de este efecto.

Listado C.3

```
class efectoretrasopeq: public efecto {
public:
   int arreglo[2000];
   int limite,indice;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};
```

```
void efectoretrasopeq::inicializar(void)
{
   int i;
   float tiempo;

   printf("Tiempo de retraso (ms): ");
   scanf("%f",&tiempo);
   limite=(frecuencia*tiempo)/1000;
   for (i=0;i<limite;i++)
      arreglo[i]=128;
   indice=0;
};

void efectoretrasopeq::obtensalida(void)
{
   efecto::obtensalida();
   indice++;
   if (indice==limite)
      indice=0;
};
```

C.4 CLASE "DELAY".

Clase derivada de efectoretraso para el delay. Hereda íntegramente la inicializacion y agrega un metodo particular obtensalida, el cual obtiene una salida parcial extrayendo el dato más antiguo del arreglo, donde almacenará la muestra de entrada, posteriormente ejecuta el método ontensalida de efectoretraso, el cual a su vez ejecuta el método obtensalida de efecto, que realiza la suma de la entrada más la salida con offset. Este código se muestra en el listado C.4

Listado C.4

```
class delay:public efectoretraso {
public:
   virtual void obtensalida(void);
};

void delay::obtensalida(void)
{
   salida=arreglo[indice];
   arreglo[indice]=entrada;
   efectoretraso::obtensalida();
};
```

C.5 CLASE "ECO".

Clase derivada de efectoretraso para el eco. Adiciona la definición de una variable de instancia para la ganancia de realimentación, la cual es pedida en el método inicializar del eco. El método obtensalida genera una salida parcial del retraso que multiplica por la ganancia de realimentacion. Este valor, sumado a la entrada, se almacena en el arreglo, e inmediatamentese ejecuta el metodo obtensalida de efectoretraso que suma entrada más salida. Código mostrado en el listado C.5.

Listado C.5

```
class eco:public efectoretraso {
public:
   float realimentacion;

   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void eco::inicializar(void)
{
   efectoretraso::inicializar();
   printf("Realimentacion (0-1): ");
   scanf("%f",&realimentacion);
};

void eco::obtensalida(void)
{
   float retro;

   salida=arreglo[indice];
   retro=128.0+realimentacion*(salida-128.0);
   arreglo[indice]=128+(entrada-128)+(retro-128);
   efectoretraso::obtensalida();
};
```

C.6 CLASE "FLANGER".

Clase derivada de efectoretraso para el flanger/chorus. Se trata de un retraso realimentado ciclicamente variable. Para lograr la variación, se introduce un segundo índice (indice2) del cual se extraerá la salida y que estará separado de indice por un número variable (barrido). Este número se incrementará o decrementará en uno cada determinado número de muestras (bandera); (paso) determina si barrido se incrementa o se decrementa, mientras que (conteo) lleva la cuenta de muestras que (barrido) ha tenido un valor, es decir, barrido conservará su valor un determinado número de muestras definido por bandera; cuando conteo alcanza a bandera, barrido decrementa

o incrementa su valor según lo especifique paso. Cuando (barrido) alcanza a alguno de sus limites (barrido_max o barrido_min), debe cambiarse la dirección del incremento (paso=-paso). En el método inicializar del flanger se piden la tasa de variación (en ms por segundo), la ganancia de retroalimentación y se inicializan las variables de instancia (paso, barrido_max, barrido_min, indice2, barrido, conteo, bandera).

Listado C.6

```
class flanger: public efectoretraso {
public:
int indice2,barrido,conteo,bandera,paso,barrido_max,barrido_min;
   float realimentacion;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void flanger::inicializar(void)
{
   float variacion;

   efectoretraso::inicializar();
   printf("Tasa de variacion del retraso (ms/s): ");
   scanf("%f",&variacion);
   printf("Realimentacion (0-1): ");
   scanf("%f",&realimentacion);
   paso=1;
   barrido_max=limite-1;
   barrido_min=0;
   indice2=0;
   barrido=0;
   conteo=0;
   bandera=1000/variacion;
};

void flanger::obtensalida(void)
{
   float retro;

   if ((indice+barrido)>(limite-1))
      indice2=barrido-((limite-1)-indice)-1;
   else
      indice2=indice+barrido;

   salida=arreglo[indice2];
   retro=128.0+realimentacion*(salida-128.0);
   arreglo[indice]=128+(entrada-128)+(retro-128);
   efectoretraso::obtensalida();
   conteo++;
   if (conteo==bandera){
```

```
        conteo=0;
        barrido=barrido+paso;
        if (barrido==barrido_max||barrido==barrido_min)
        paso=-paso;
    }
};
```

C.7 CLASE "DISTORSION".

La distorsión se construyó definiendo la clase distorsion derivada directamente de efecto. Se declaran las variables de instancia amplificacion y recorte. En la inicialización se leen los valores de estas variables. En el método obtensalida se genera la salida amplificando la muestra de entrada (multiplicándola por amplificacion) y recortando su parte positiva (comparando el valor contra recorte). El código se presenta en el listado C.7.

Listado C.7

```
class distorsion: public efecto{
public:
   float amplificacion;
   int recorte;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void distorsion::inicializar(void)
{
   printf("Amplificacion: ");
   scanf("%f",&amplificacion);
   printf("Recorte (0-127): ");
   scanf("%d",&recorte);
};

void distorsion::obtensalida(void)
{
   salida=128+amplificacion*(entrada-128);
   if (salida>128+recorte)
      salida=recorte;
   if (salida<0)
      salida=0;
};
```

C.8 CLASE "TREMOLO".

Clase derivada de efecto para el trémolo. En el método inicializar se obtiene la frecuencia (velocidad) de la señal moduladora y el factor por el que se multiplicará la amplitud (profundidad), se inicializa una variable de instancia (rampa) que servirá de argumento para obtener el valor de la señal moduladora en cada muestra; a esta rampa se le suma un incremento también calculado en este método a partir de la frecuencia de muestreo y la frecuencia de la moduladora. En el método obtensalida, se le aplica a la rampa la funcion seno (lfo) y se modula la muestra de entrada para obtener la salida, la modulacion consiste en la suma de uno mas lfo por profundidad (parcial), posteriormente se obtiene la salida total multiplicando (modulando) la entrada por parcial y realizando el recorte necesario. En el listado C.8 se muestra el código de este efecto

Listado C.8

```
class tremolo: public efecto {
public:
   float rampa,incremento,profundidad;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void tremolo::inicializar(void)
{
   int velocidad;

   efecto::inicializar();
   printf("Velocidad (Hz): ");
   scanf("%d",&velocidad);
   printf("Profundidad (0-1): ");
   scanf("%f",&profundidad);
   rampa=0.0;
   incremento=2.0*PI*velocidad/frecuencia;
};

void tremolo::obtensalida(void)
{
   float lfo,parcial;

   lfo=sin(rampa);
   rampa=rampa+incremento;
   parcial=1+lfo*profundidad;
   salida=128+(entrada-128)*parcial;
   if (salida>255)
      salida=255;
   if (salida<0)
      salida=0;};
```

C.9 CLASE "DELAYFB".

Clase derivada de efectoretrasopeq para los retrasos realimentados que forman parte del reverb. En el método inicializar se agrega una realimentación y se omite la lectura de la frecuencia de muestreo. Para obtener la salida, en el método obtensalida se genera una salida parcial que procesa la ganancia de retroalimentación por la salida (retro) e iguala la entrada a 0 (128, tomando en cuenta el offset de los datos de 8 bits del formato .VOC), para después mandar a ejecutar el método obtensalida de la clase efectoretraso que suma entrada más salida. El listado C.9 muestra el código.

Listado C.9

```
class delayfb:public efectoretrasopeq {
public:
   float realimentacion;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void delayfb::inicializar(void)
{
   efectoretrasopeq::inicializar();
   printf("Realimentacion (0-1): ");
   scanf("%f",&realimentacion);
};

void delayfb::obtensalida(void)
{
   float retro;

   salida=arreglo[indice];
   retro=128.0+realimentacion*(salida-128.0);
   arreglo[indice]=128+(entrada-128)+(retro-128);
   entrada=128;
   efectoretrasopeq::obtensalida();
};
```

C.10 CLASE "ALLPASS".

Clase derivada de delayfb para los filtros paso-todo. Puede decirse que es una version especializada del delayfb. En el método inicializar se piden ganancias para la entrada y la salida de la linea de retraso. En el método obtensalida, después de amplificar la salida y la entrada por las ganancias pedidas, se suman. En el listado C.10 se muestra el código correspondiente.

Listado C.10

```
class allpass:public delayfb {
public:
   float ganent,gansal;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void allpass::inicializar(void)
{
   delayfb::inicializar();
   printf("Ganancia entrada: ");
   scanf("%f",&ganent);
   printf("Ganancia salida: ");
   scanf("%f",&gansal);
};

void allpass::obtensalida(void)
{  int aux;

   aux=entrada;
   delayfb::obtensalida();
   entrada=128+ganent*(aux-128);
   salida=128+gansal*(salida-128);
   efecto::obtensalida();
};
```

C.11 CLASE "REVERB".

Clase derivada de efecto para el reverb. El reverb consta de 4 retrasos realimentados (delayfb) en paralelo seguidos de 2 filtros paso-todo (allpass) en serie. En el método inicializar se crean los retrasos a partir de dos arreglos de objetos del tipo correspondiente y se solicitan sus valores. En el método obtensalida, la salida se obtiene manejando la entrada al efecto como entrada al bloque paralelo de delayfb's, cuyas salidas serán sumadas (parcial1). El resultado servirá de entrada a los allpass conectados en serie o cascada, al final de la cual se tendrá la salida del efecto. En el listado C.11 se muestra el código respectivo.

Listado C.11

```
class reverb:public efecto{
public:
   delayfb *retrasofb[4];
   allpass *pasotodo[2];
   virtual void inicializar(void);
   virtual void obtensalida(void);
```

```
};

void reverb::inicializar(void)
{
   int i,j;
   float tiempo;

   efecto::inicializar();

   for(i=0;i<4;i++){
      retrasofb[i]=new delayfb;
      printf("Parametros del retraso realimentado %d\n",i+1);
      retrasofb[i]->frecuencia=frecuencia;
      retrasofb[i]->inicializar();
   };

   for(i=0;i<2;i++){
      pasotodo[i]=new allpass;
      printf("Parametros del filtro paso-todo %d\n",i+1);
      pasotodo[i]->frecuencia=frecuencia;
      pasotodo[i]->inicializar();
  };

};

void reverb::obtensalida(void)
{
   int i,parcial1;


   for (i=0;i<4;i++){
      retrasofb[i]->entrada=entrada;
      retrasofb[i]->obtensalida();
   };


parcial1=128+(retrasofb[0]->salida-128)+(retrasofb[1]->salida-12
8)+(retrasofb[2]->salida-128)+(retrasofb[3]->salida-128);
   pasotodo[0]->entrada=parcial1;
   pasotodo[0]->obtensalida();
   pasotodo[1]->entrada=pasotodo[0]->salida;
   pasotodo[1]->obtensalida();
   salida=pasotodo[1]->salida;
   efecto::obtensalida();
};
```

C.12 EL FORMATO .VOC.

El formato .VOC consta de los siguientes elementos principales [STO93]:

- **Bytes $00-$13 (0-19 decimal).**

Estos primeros 19 bytes contienen el texto "Creative Voice File" seguido del valor $1A.

- **Bytes $14-$15 (20-21).**

Contienen el offset a partir del cual se encuentran los datos, comenzando con el valor menos significativo. Para la versión 1.10 del formato, el valor de estos bytes es $1A00, dado que el encabezado tiene precisamente 1A bytes de longitud.

- **Bytes $16-$17 (22-23).**

Contienen el número de la versión del formato, también comenzando con el valor menos significativo. Para la versión 1.10, el byte $17 contiene el valor $01 (1 decimal) y el byte $16 contiene el valor $0A (10 decimal).

- **Bytes $18-$19 (24-25).**

Estos bytes sirven para validar la versión del formato contenida en los dos bytes anteriores. Los bytes $18-$19 contienen el complemento del número de la versión sumado a $1234, comenzando con el byte menos significativo.

Al encabezado siguen los bloques de datos, de los cuales existen 9 tipos. En este resumen se presentará la estructura de los dos tipos de bloques más importantes y que fueron considerados en el desarrollo del programa.

- **Bloque 0 (terminador).**

Es el bloque que marca el fin de un archivo de datos en formato .VOC. Su estructura consta únicamente de un byte que define el tipo de bloque 0.

- **Bloque 1 (bloque nuevo de datos).**

Es el bloque utilizado más frecuentemente; contiene muestras lineales de datos directamente reproducibles en la tarjeta de audio. En primer lugar, contiene un byte que especifica su tipo (1); posteriormente contiene tres bytes que definen la longitud del bloque de datos, que se clcula a partir de la fórmula:

```
longitud=byte1+byte2*256+byte3*65535
```

El quinto bloque contiene una constante a partir de la cual se calcula la frecuencia de muestreo según la fórmula:

```
frec=-1000000 DIV (constante-256)
```

El sexto byte indica el factor de compresión de los datos. El 0 indica que los datos no están comprimidos, mientras que el valor máximo, 3, indica que por cada 8 bytes de datos originales, se tienen 2 bytes comprimidos.

Finalmente, se tienen los bytes de muestras de audio.

C.13 CODIGO COMPLETO DEL PROGRAMA DE PRODUCCION DE EFECTOS ESPECIALES EN SEÑALES DE AUDIO.

A continuación se presenta el código completo del programa incluyendo algunas otras clases definidas para otros fines de manejo de archivos y conversión de formatos. Este programa fue compilado con Turbo C++ de Borland. Toma como argumentos de entrada los nombres de los archivos de entrada y de salida, en ese orden. Una vez que verifica la existencia y corrección del archivo de entrada en formato .VOC, solicita al usuario el tipo de efecto con el que se procesarán los datos de entrada para generar el archivo de salida. Dependiendo de esta selección, el usuario debe introducir los valores de los parámetros correspondientes al efecto deseado. El programa generará entonces el archivo de salida con el mismo formato.

```
// efectos.cpp, 10-05-96, 17:05
// Productor de efectos especiales en señales de audio
// Entrada: archivo de datos en formato .VOC
// Salida: archivo de datos en formato .VOC
// @ 1996, Ricardo Arroyo y Julio Orozco

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <math.h>
# define PI 3.14159 // Define una constante PI
# define USHORT unsigned short


// ***************************************************************************
// ***************************************************************************

// Estructura para el encabezado del formato .VOC.

typedef struct
{
    char nombre[20];
    unsigned short off;
    unsigned short ver;
    unsigned short id;
} ENCABEZADOVOC;

// ***************************************************************************
// ***************************************************************************

// Clase para manejar archivos.
```

```
// Variables de instancia: nombre, apuntador, encabezado (del formato .VOC)
// y estado (para casos de error y para agregar otros tipos de archivo).

// Clase para archivo.

class archivo {
public:
   char *nombre;
   FILE *fp;
   int estado;
   ENCABEZADOVOC encabezado;

   void abrelectura(void);
   void abreescritura(void);
   int lee(void);
   void escribe(int);
};

// Abre un archivo para lectura binaria, validando el formato .VOC.

void archivo::abrelectura(void)
{
   char cadena[80];
   unsigned short ra;

   strcpy(cadena,nombre);
   strcat(cadena,".VOC");
   if((fp=fopen(cadena,"rb"))==NULL)
   {
      strcpy(cadena,nombre);
      if ((fp=fopen(cadena,"rb"))==NULL)
      {
       printf("Error al abrir archivo .VOC de entrada [%s]\n",nombre);
       estado=1;
       exit(1);
      }
   }
   fread(&encabezado,1,sizeof(ENCABEZADOVOC),fp);
   for(ra=0;ra<19;ra++)
      cadena[ra]=encabezado.nombre[ra];
   cadena[ra]=0;
                                            if((strcmp(cadena,"Creative         Voice
File")!=0)||(encabezado.nombre[ra]!=0x1A))
    {
      printf("No es un archivo Creative Voice (.VOC)\n");
      estado=1;
    }
    if (estado!=1)
    {
                                         printf("Creative     Voice     File
V%u.%02u\n",(encabezado.ver>>8)&0xFF,encabezado.ver&0xFF);
       printf("Los datos comienzan en el offset %08Xh\n",encabezado.off);
       printf("Codigo de identificacion del archivo %04Xh, ",encabezado.id);
       if((((~encabezado.ver)+0x1234)&0xFFFF)==encabezado.id)
        printf("valido\n");
       else printf("invalido (ignorado)\n");
    }
   strcpy(cadena,nombre);
   strcat(cadena,".VOC");
   if ((fp=fopen(cadena,"rb"))==NULL)
```

```
   {
      strcpy(cadena,nombre);
      fp=fopen(cadena,"rb");
   }

};

// Abre un archivo para escritura binaria.

void archivo::abreescritura(void)
{
   fp=fopen(nombre,"wb");
};

// Lee un dato entero del archivo

int archivo::lee(void)
{
   int c;
   c=getc(fp);
   return c;
};

// Escribe un dato entero en el archivo.

void archivo::escribe(int c)
{
   putc(c,fp);
};

// ***********************************************************************
// ***********************************************************************

// Clase generica para definir a partir de ella todos los efectos de audio.
// Tiene como variables una entrada y una salida enteras, ademas de la
// frecuencia de muestreo.

// Sus metodos tienen la funcion de inicializar el efecto y obtener
// la muestra de salida a partir de la muestra de entrada.

class efecto {
public:
   int entrada;
   int salida;
   int frecuencia;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

// Solicita la frecuencia de muestreo

void efecto::inicializar(void)
{
   printf("Frecuencia de muestreo [Hz]: ");
   scanf("%d",&frecuencia);
};

// Mezcla o suma la muestra de entrada con la muestra de salida.
// Realiza un offset de 128, siendo los datos de 8 bits (de 0 a 255)
// y recorta el valor de la muestra de salida para que no exceda estos
// limites.
```

```
void efecto::obtensalida(void)
{
   salida=128+(entrada-128)+(salida-128);
   if (salida>255)
      salida=255;
   if (salida<0)
      salida=0;
};

// *************************************************************************
// *************************************************************************

// Clase derivada de efecto para definir los efectos basados en lineas de
// retraso (delay, eco, y flanger).

// Define un arreglo para el retraso, el limite del arreglo y un indice
// para su recorrido.

class efectoretraso:public efecto {
public:
      int arreglo[8000];
      int limite,indice;
      virtual void inicializar(void);
      virtual void obtensalida(void);
};

// Solicita el tiempo de retraso en (ms), calcula el limite a partir de
// este dato y de la frecuencia de muestreo; inicializa el arreglo y el
// indice.

void efectoretraso::inicializar(void)
{
   float tiempo;
   int i;

   efecto::inicializar();
   printf("Tiempo de retraso (ms): ");
   scanf("%f",&tiempo);
   limite=(frecuencia*tiempo)/1000;
   for (i=0;i<limite;i++)
      arreglo[i]=128;
   indice=0;
};

// Incrementa el indice y lo reinicializa en caso de haber llegado al
// limite.

void efectoretraso::obtensalida(void)
{
   efecto::obtensalida();
   indice++;
   if (indice==limite)
      indice=0;
};

// *************************************************************************
// *************************************************************************

// Clase derivada de efecto para el conjunto de retrasos que se utilizan
// en el reverb. Los retrasos tipicos del reverb son menores a los retrasos
```

```
// del delay, el eco y el flanger. La única diferencia con respecto a
// efectoretraso es la inicialización del arreglo (en este caso esta limitada
// a 2000 muestras). Esto es necesario para no saturar la memoria al crear
// los 6 retrasos (4 realimentados y 2 filtros paso-todo) necesarios para
// el reverb.

class efectoretrasopeq: public efecto {
public:
   int arreglo[2000];
   int limite,indice;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void efectoretrasopeq::inicializar(void)
{
   int i;
   float tiempo;

   printf("Tiempo de retraso (ms): ");
   scanf("%f",&tiempo);
   limite=(frecuencia*tiempo)/1000;
   for (i=0;i<limite;i++)
      arreglo[i]=128;
   indice=0;
};

void efectoretrasopeq::obtensalida(void)
{
   efecto::obtensalida();
   indice++;
   if (indice==limite)
      indice=0;
};

// ************************************************************************
// ************************************************************************

// Clase derivada de efectoretraso para el delay.
// Hereda integramente la inicializacion y obtiene su salida parcial
// extrayendo el dato mas antiguo del arreglo, donde almacenara la muestra
// de entrada.

class delay:public efectoretraso {
public:
   virtual void obtensalida(void);
};

void delay::obtensalida(void)
{
   salida=arreglo[indice];
   arreglo[indice]=entrada;
   efectoretraso::obtensalida();
};

// ************************************************************************
// ************************************************************************

// Clase derivada de efectoretraso para el eco.
// Adiciona la definicion de una ganancia de realimentacion.
```

```
class eco:public efectoretraso {
public:
   float realimentacion;

   virtual void inicializar(void);
   virtual void obtensalida(void);
};

// Solicita la ganancia de realimentacion

void eco::inicializar(void)
{
   efectoretraso::inicializar();
   printf("Realimentacion (0-1): ");
   scanf("%f",&realimentacion);
};

// Obtiene una salida parcial del retraso que multiplica por la ganancia
// de realimentacion. este valor, sumado a la entrada, se almacenan en el
// arreglo.

void eco::obtensalida(void)
{
   float retro;

   salida=arreglo[indice];
   retro=128.0+realimentacion*(salida-128.0);
   arreglo[indice]=128+(entrada-128)+(retro-128);
   efectoretraso::obtensalida();
};

// ************************************************************************
// ************************************************************************

// Clase derivada de efectoretraso para el flanger/chorus.
// Se trata de un retraso realimentado ciclicamente variable.
// Para lograr la variacion, se introduce un segundo indice (indice2)
// del cual se extraera la salida y que estara separado de indice por
// un numero variable (barrido). Este numero se incrementara o decrementara
// en uno cada determinado numero de muestras (bandera); (paso) determina
// si barrido se incrementa o se decrementa, mientras que (conteo) lleva
// la cuenta de muestras que (barrido) ha tenido un valor; cuando (conteo)
// alcanza a bandera, (barrido) debe incrementarse o decrementarse.
// Cuando (barrido) alcanza a alguno de sus limites (barrido_max y
// barrido_min), debe cambiarse la direccion del incremento (paso=-paso).

class flanger: public efectoretraso {
public:
   int indice2,barrido,conteo,bandera,paso,barrido_max,barrido_min;
   float realimentacion;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void flanger::inicializar(void)
{
   float variacion;

   efectoretraso::inicializar();
   printf("Tasa de variacion del retraso (ms/s): ");
   scanf("%f",&variacion);
```

```
    printf("Realimentacion (0-1): ");
    scanf("%f",&realimentacion);
    paso=1;
    barrido_max=limite-1;
    barrido_min=0;
    indice2=0;
    barrido=0;
    conteo=0;
    bandera=1000/variacion;
};

void flanger::obtensalida(void)
{
    float retro;

    if ((indice+barrido)>(limite-1))
       indice2=barrido-((limite-1)-indice)-1;
    else
       indice2=indice+barrido;

    salida=arreglo[indice2];
    retro=128.0+realimentacion*(salida-128.0);
    arreglo[indice]=128+(entrada-128)+(retro-128);
    efectoretraso::obtensalida();
    conteo++;
    if (conteo==bandera){
       conteo=0;
       barrido=barrido+paso;
       if (barrido==barrido_max||barrido==barrido_min)
       paso=-paso;
    }
};

//**************************************************************************
//**************************************************************************

// Clase derivada de efecto para la distorsion.
// Declara una variable para la amplificacion y una para el recorte.

// En la inicializacion se leen estas variables y en el metodo obtensalida
// se genera la muestra de salida multiplicando la muestra de entrada por
// el valor de amplificacion y luego recortando la parte positiva. Esto
// se logra igualando el valor de salida al valor de recorte.

class distorsion: public efecto{
public:
    float amplificacion;
    int recorte;
    virtual void inicializar(void);
    virtual void obtensalida(void);
};

void distorsion::inicializar(void)
{
    printf("Amplificacion: ");
    scanf("%f",&amplificacion);
    printf("Recorte (0-127): ");
    scanf("%d",&recorte);
};

void distorsion::obtensalida(void)
```

```
{
   salida=128+amplificacion*(entrada-128);
   if (salida>128+recorte)
      salida=recorte;
   if (salida<0)
      salida=0;
};

// ***********************************************************************
// ***********************************************************************

// Clase derivada de efecto para el tremolo.
// Se obtiene en la inicializacion la frecuencia (velocidad) de la señal
// moduladora y el factor por el que se multiplicara la amplitud
// (profundidad) de esta para sumarlo a 1 y luego multiplicar el resultado
// por la muestra de entrada.
// Se genera una rampa sumando sucesivamente (incremento), el cual es
// determinado mediante la frecuencia de muestreo y la frecuencia de la
// moduladora. Al valor de la rampa se le aplica la funcion seno y se
// modula la muestra de entrada para obtener la salida.

class tremolo: public efecto {
public:
   float rampa,incremento,profundidad;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void tremolo::inicializar(void)
{
   int velocidad;

   efecto::inicializar();
   printf("Velocidad (Hz): ");
   scanf("%d",&velocidad);
   printf("Profundidad (0-1): ");
   scanf("%f",&profundidad);
   rampa=0.0;
   incremento=2.0*PI*velocidad/frecuencia;
};

void tremolo::obtensalida(void)
{
   float lfo,parcial;

   lfo=sin(rampa);
   rampa=rampa+incremento;
   parcial=1+lfo*profundidad;
   salida=128+(entrada-128)*parcial;
   if (salida>255)
      salida=255;
   if (salida<0)
      salida=0;
};

// ***********************************************************************
// ***********************************************************************

// Clase derivada de efectoretrasopeq para los retrasos realimentados que
// forman parte del reverb.
```

```
// Se agrega una realimentacion y se omite en la inicializacion la entrada
// de la frecuencia de muestreo.
// Para obtener la salida, se iguala la salida a 0 (128, tomando en cuenta
// el offset de los datos de 8 bits del formato .VOC).

class delayfb:public efectoretrasopeq {
public:
   float realimentacion;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void delayfb::inicializar(void)
{
   efectoretrasopeq::inicializar();
   printf("Realimentacion (0-1): ");
   scanf("%f",&realimentacion);
};

void delayfb::obtensalida(void)
{
   float retro;

   salida=arreglo[indice];
   retro=128.0+realimentacion*(salida-128.0);
   arreglo[indice]=128+(entrada-128)+(retro-128);
   entrada=128;
   efectoretrasopeq::obtensalida();
};

// ************************************************************************
// ************************************************************************

// Clase derivada de delayfb para los filtros paso-todo.
// Puede decirse que es una version especializada del delayfb, donde se
// agregan ganancias para la entrada y la salida de la linea de retraso
// que despues de ser amplificadas seran sumadas.

class allpass:public delayfb {
public:
   float ganent,gansal;
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void allpass::inicializar(void)
{
   delayfb::inicializar();
   printf("Ganancia entrada: ");
   scanf("%f",&ganent);
   printf("Ganancia salida: ");
   scanf("%f",&gansal);
};

void allpass::obtensalida(void)
{  int aux;

   aux=entrada;
   delayfb::obtensalida();
   entrada=128+ganent*(aux-128);
   salida=128+gansal*(salida-128);
```

```
   efecto::obtensalida();
};

// ***********************************************************************
// ***********************************************************************

// Clase derivada de efecto para el reverb.
// El reverb consta de 4 retrasos realimentados (delayfb) en paralelo
// seguidos de 2 filtros paso-todo (allpass) en serie.
// En la inicializacion se crean los retrasos a partir de dos arreglos
// de objetos del tipo correspondiente y se solicitan sus valores.
// La salida se obtiene manejando la entrada al efecto como entrada al
// bloque paralelo de delayfb's, cuyas salidas seran sumadas. El resultado
// servira de entrada a los allpass conectados en serie o cascada, al final
// de la cual se tendra la salida del efecto.

class reverb:public efecto{
public:
   delayfb *retrasofb[4];
   allpass *pasotodo[2];
   virtual void inicializar(void);
   virtual void obtensalida(void);
};

void reverb::inicializar(void)
{
   int i,j;
   float tiempo;

   efecto::inicializar();

   for(i=0;i<4;i++){
      retrasofb[i]=new delayfb;
      printf("Parametros del retraso realimentado %d\n",i+1);
      retrasofb[i]->frecuencia=frecuencia;
      retrasofb[i]->inicializar();
   };

   for(i=0;i<2;i++){
      pasotodo[i]=new allpass;
      printf("Parametros del filtro paso-todo %d\n",i+1);
      pasotodo[i]->frecuencia=frecuencia;
      pasotodo[i]->inicializar();
  };

};

void reverb::obtensalida(void)
{
   int i,parcial1;


   for (i=0;i<4;i++){
      retrasofb[i]->entrada=entrada;
      retrasofb[i]->obtensalida();
   };


parcial1=128+(retrasofb[0]->salida-128)+(retrasofb[1]->salida-128)+(retrasofb
[2]->salida-128)+(retrasofb[3]->salida-128);
   pasotodo[0]->entrada=parcial1;
```

```
   pasotodo[0]->obtensalida();
   pasotodo[1]->entrada=pasotodo[0]->salida;
   pasotodo[1]->obtensalida();
   salida=pasotodo[1]->salida;
   efecto::obtensalida();
};

// **********************************************************************
// **********************************************************************

// Clase para el proceso de archivos.
// Tanto la entrada como la salida de este objeto seran archivos de datos
// Se tendra ademas una variable que defina el tipo de proceso a realizar
// El metodo principal es el que procesa los datos del archivo de entrada
// con formato voc y los almacena en el archivo de salida con el mismo for-
// mato. Posteriormente, pueden agregarse distintos formatos de archivos.

class proceso {
public:
   archivo entrada,salida;
   int tipo;
   void vocavoc (void);
};

void proceso::vocavoc(void)
{
   char dtype;
   char cual;
   unsigned long dlen;
   unsigned char tc;
   unsigned char pt;
   unsigned char ca;
   unsigned long la;
   unsigned short ra;
   efecto *efecto1;

// Creacion virtual del objeto a partir de la opcion dada por el usuario.

   printf("Efecto:\n (1) Delay; (2) Eco; (3) Flanger; (4) Tremolo; (5) Reverb:
");
   cual=getchar();
   if (cual=='1')
      efecto1=new delay;
   if (cual=='2')
      efecto1=new eco;
   if (cual=='3')
      efecto1=new flanger;
   if (cual=='4')
      efecto1=new tremolo;
   if (cual=='5')
      efecto1=new reverb;

// Llamada polimorfica del metodo de inicializacion del efecto.

   efecto1->inicializar();

   // Lectura del encabezado del archivo de entrada y escritura del mismo
   // en el archivo de salida.

   fread(&entrada.encabezado,1,sizeof(ENCABEZADOVOC),entrada.fp);
   salida.encabezado=entrada.encabezado;
```

```
fwrite(&salida.encabezado,1,sizeof(ENCABEZADOVOC),salida.fp);

// Ciclo de proceso de los bloques de datos del formato .VOC

while(1)
  {
    fread(&dtype,1,1,entrada.fp);
    fwrite(&dtype,1,1,salida.fp);
    if (dtype==0)
       break;
    dlen=0;
    fread(&dlen,1,3,entrada.fp);
    printf("\nTipo de Bloque: %u,  ",(USHORT)dtype);
    fwrite(&dlen,1,3,salida.fp);
    if(dtype==0)
    {
       printf("Terminador\n");
       break;
    }
    if(dtype==1)
    {
       printf("Datos lineales\n");
       printf("Longitud del bloque: %lu\n",dlen);
       fread(&tc,1,1,entrada.fp);
       printf("Constante de tiempo %u, ",(USHORT)tc);
       ra=1000000L/(256L-tc);
       printf("%u Hz\n",ra);
       fwrite(&tc,1,1,salida.fp);
       fread(&pt,1,1,entrada.fp);
       fwrite(&pt,1,1,salida.fp);
       printf("Tipo de compresion %u:  ",(USHORT)pt);
       switch(pt)
       {
          case 0: printf("no comprimido a 8 bits\n"); break;
          case 1: printf("comprimido a 4 bits comprimido\n"); break;
          case 2: printf("comprimido a 2.6 bits\n"); break;
          case 3: printf("comprimido a 2 bits\n"); break;
          case 4: printf("1 canal\n"); break;
          case 5: printf("2 canales\n"); break;
          case 6: printf("3 canales\n"); break;
          case 7: printf("4 canales\n"); break;
          case 8: printf("5 canales\n"); break;
          case 9: printf("6 canales\n"); break;
          case 10: printf("7 canales\n"); break;
          default: printf("Desconocido\n"); break;
       }
       for(la=2;la<dlen;la++)
       {
          // El dato leido del archivo (ca)se asignara como entrada al
          // efecto. Se ejecutara el metodo que obtiene la salida para
          // el efecto creado y se asignara a ca para ser escrito en el
          // archivo de salida.

          fread(&ca,1,1,entrada.fp);
          efecto1->entrada=ca;
          efecto1->obtensalida();
          ca=efecto1->salida;
          fwrite(&ca,1,1,salida.fp);
       }
    }
  }
```

```
};

archivo arch1,arch2;
proceso proc1;

// La funcion main se limita a asignar sus parametros de entrada a los
// nombres de dos objetos archivos, abrrir el primero para lectura, el
// segundo para escritura, asignarlos como entrada y salida de un objeto
// de proceso (proc1) y llamar el proceso vocavoc del mismo.

main(int argc, char *argv[])
{
   int i,c;

   if(argc==1)
      {
      printf("Archivo .VOC no especificado\n");
      exit(1);
      }

    arch1.nombre=argv[argc-2];
    arch2.nombre=argv[--argc];

    arch1.abrelectura();

    if (arch1.estado==1)
       exit(1);

    arch2.abreescritura();

    proc1.entrada=arch1;
    proc1.salida=arch2;

    proc1.vocavoc();
}
```

REFERENCIAS.

[STO93] Stolz, Axel. *The Sound Blaster Book.* Abacus, Estados Unidos, 1993. pp. 154-157.

Apéndice D:
Paleta de Efectos de Audio en Ptolemy.

INTRODUCCION.

En este apéndice se presentan las características generales de la paleta de efectos de audio construida en Ptolemy. La paleta incluye, como se aprecia en la figura D.1, los iconos correspondientes a las galaxias que realizan el eco, el delay, el trémolo, la distorsión, el retraso realimentado, el filtro paso todo y el reverb.

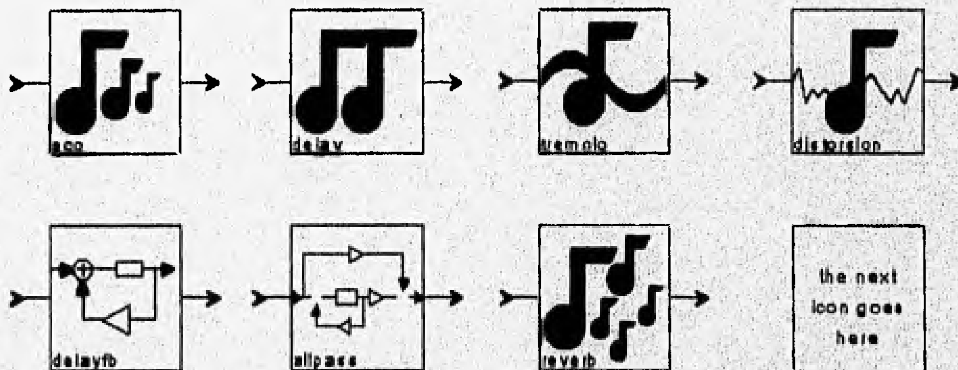


Fig. D.1

Para cada galaxia se diseñó y asoció un icono que fuera representativo de la función que realiza la galaxia. Por ejemplo, para el eco el icono consta de una nota musical seguida de varias notas más pequeñas; la primera nota es de color negro, mientras que las notas más pequeñas son de color azul. Esto da la idea de que al sonido original sigue una secuencia de notas cuya amplitud decrece exponencialmente.

Estas galaxias pueden utilizarse según los lineamientos generales de pigi, la interfaz gráfica de Ptolemy (ver apéndice A.3). Esto quiere decir que pueden incluirse como bloques en otros programas visuales de Ptolemy y que pueden modificarse sus parámetros visualmente. De hecho, las simulaciones presentadas en el capítulo 6 fueron creadas tomando las galaxias de esta paleta.

A continuación se presenta una revisión de cada galaxia, haciendo especial énfasis en los parámetros definidos para cada una y la relación de dichos parámetros con los bloques que conforman la galaxia.

D.1 ECO.

La galaxia se presenta en la figura D.2.

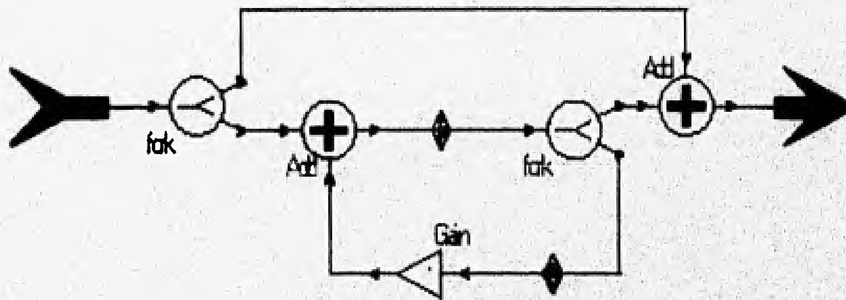


Fig. D.2

Parámetros:

- *realimentación*.- con este parámetro modificamos el valor de la ganancia del amplificador en la retroalimentación, el valor debe estar entre 0-1.
- *retraso*.- el valor de este parámetro indica las muestras de retraso entre la señal original y la repetición de la señal, el valor está definido en función de la frecuencia de muestreo. Por ejemplo, para una frecuencia de muestreo de 8 kHz, y un retraso de medio segundo, el valor debe ser 4000.

D.2 DELAY.

La galaxia del delay se muestra en la figura D.3.

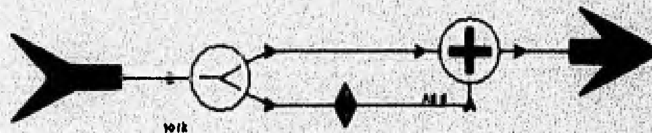


Fig. D.3

Parámetros:

- *retraso*.- el valor de este parámetro indica las muestras de retraso entre la señal original y la repetición de la señal.

D.3 TREMOLO.

En el diagrama D.4 se muestra la galaxia del tremolo.

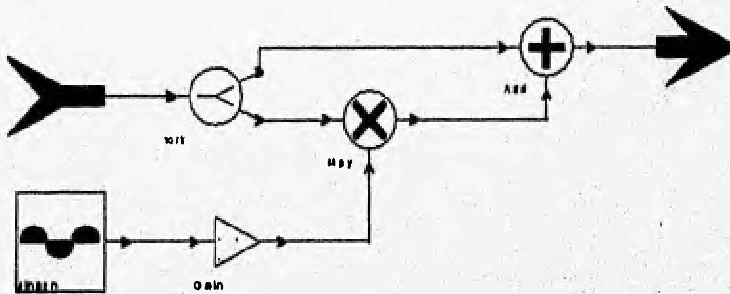


Fig. D.4

Parámetros:

- *profundidad*.- define la ganancia del amplificador conectado a la salida del generador senoidal y por lo tanto, el intervalo de variación de la amplitud de la señal de entrada; este valor debe estar entre 0-1.
- *velocidad*.- frecuencia de oscilación en Hz. del generador senoidal. El valor debe estar en el intervalo de 0 a 15 Hz., para que se encuentre por debajo del intervalo de frecuencias audibles.

D.4 DISTORSION.

La galaxia de la distorsión se muestra en la figura D.5.

Parámetros:

- *sensitividad*.- es el valor de la constante de recorte que define el valor de recorte de la señal; este valor depende de la amplitud de la señal de entrada.
- *ganancia*.- es la ganancia de amplificación previa al recorte; este valor depende de cuánto se desee distorsionar la señal.

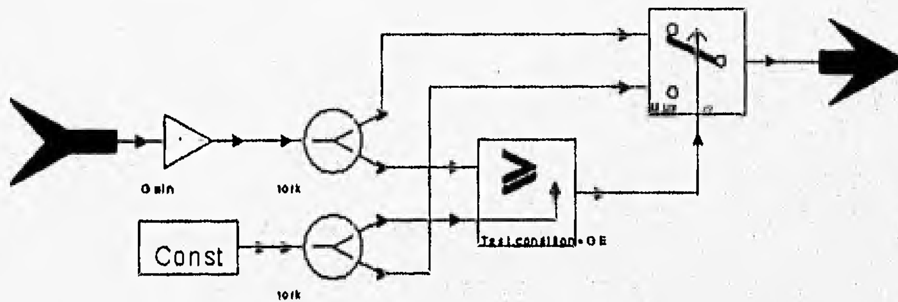


Fig. D.5

D.5 RETRASO REALIMENTADO (DELAYFB).

En la figura D.6 vemos la galaxia del retraso realimentado.

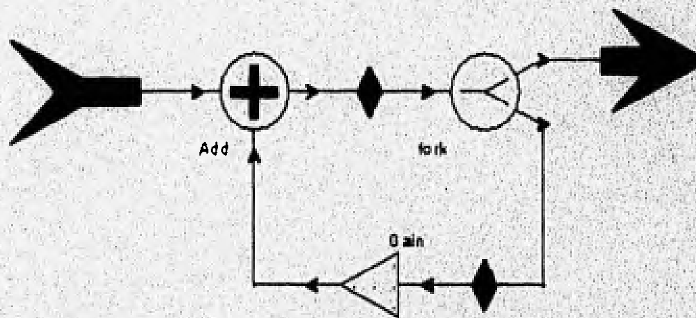


Fig. D.6

Parámetros:

- *retraso*.- el valor de este parámetro indica las muestras de retraso entre la señal original y la repetición de la señal.
- *realimentación*.- es la ganancia del amplificador en la realimentación. rango del valor entre 0-1.

D.6 FILTRO PASO TODO (ALLPASS).

En la figura D.7 presentamos nuevamente la galaxia del filtro paso todo. Cabe observar que el bloque de retraso se compone de un retarso realimentado construido en una galaxia aparte (figura D.6).

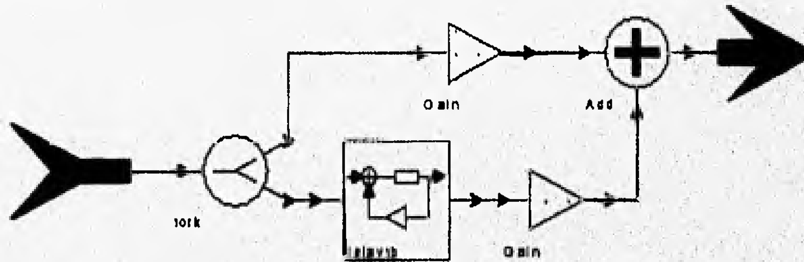


Fig. D.7

Parámetros:

- *retrasoallpass.*- es el retraso que se transfiere como parámetro al delayfb
- *gananciadir.*- es la ganancia de la señal de entrada previa a la suma.
- *gananciaret.*- es la ganancia del bloque de realimentación previa a la suma.
- *realimentación.*- es la ganancia que se transfiere como parámetro al delayfb.

D.7 REVERB.

La figura D.8 muestra la galaxia del reverb. Puede observarse que el reverb esta construido con cuatro delayfb, los cuales por default tienen 0.7 de realimentación y cuyos retrasos son los parámetros retraso1 al 4 del reverb. Los dos parámetros restantes se refieren al retraso de los bloques allpass. Se ha decidido no incluir las ganancias de los bloques allpass para no tener demasiados parámetros a controlar. Por lo tanto, internamente (es decir para cada bloque), se han fijado las realimentaciones a 0.7 y las ganancias previas a la suma a 1.0.

Para los siguientes retrasos, el valor (numero de muestras) debe especificarse en función de la frecuencia de muestreo.

Parámetros:

- *retraso1*
- *retraso2*
- *retraso3*
- *retraso4*
- *retrasoallpass1*
- *retrasoallpass2*

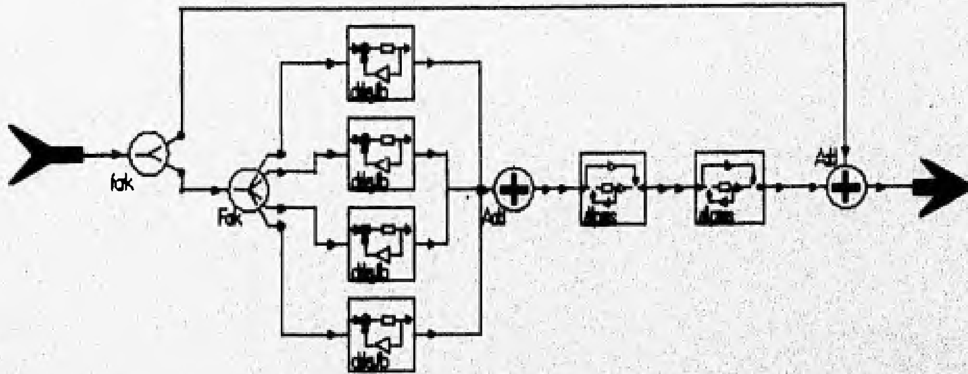


Fig D.8

Apéndice E:
Microcódigo Generado por Ptolemy.

INTRODUCCION.

La generación automática de microcódigo es una de las características más innovadoras de Ptolemy. En la actualidad, Ptolemy incluye dominios que sintetizan el código en ensamblador para arquitecturas basadas en los procesadores digitales de señales 56000 y 96000 de Motorola. Estos dominios incluyen los mismos tipos básicos de estrellas que el dominio SDF de simulación, así como nuevas estrellas, todas diseñadas específicamente para optimizar los tipos de datos y operaciones que pueden realizar los DSP's.

De esta forma, en lugar de contar con un generador senoidal en la paleta de entradas (Signal Sources), se cuenta con un generador de tonos, al que directamente se le puede definir una amplitud. Otro ejemplo que resulta importante mencionar es el caso de los delays. En el dominio SDF, los delays no son estrellas, sino modificadores de los arcos que conectan las estrellas, y es el scheduler (organizador de la ejecución del programa visual) el que se encarga de desempeñar la función de retraso. Sin embargo, en el dominio de generación de código para el DSP 56000, la necesidad de inicializar y manejar un buffer de memoria, implica la creación de una estrella específica para los delays.

Por otra parte, debido a que Ptolemy está orientado a sistemas interconstruídos, los targets (objetivos) de este tipo de dominios incluyen opciones de ensamblado, carga y ejecución del microcódigo en el caso de contar con una arquitectura basada en un DSP 56000 o 96000 conectada a la estación de trabajo. Asimismo, se incluyen paletas con estrellas o galaxias que permiten el control en tiempo real de esta arquitectura, incluyendo entradas y salidas asíncronas en sus puertos.

En el cuerpo de este apéndice, se presentan los programas visuales y el código generado a partir de ellos en el dominio CG56 para el delay, el eco, el trémolo y el reverb. Los listados tienen un encabezado donde se incluyen los datos de creación del código. Posteriormente, se tiene el código de inicialización de cada bloque y luego el código que desempeñará la función del mismo.

E.1 DELAY.

En la figura E.1 se presenta el programa visual creado para generar el microcódigo. Puede observarse el uso de una estrella específica para la línea de retraso.

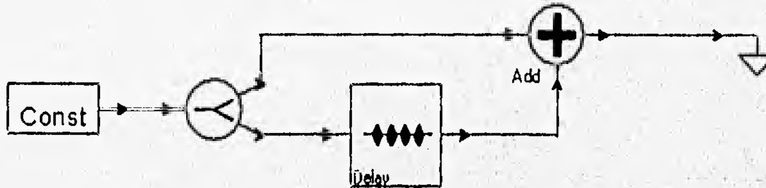


Fig. E.1

En el listado E.1 se muestra el microcódigo generado.

Listado E.1

```
        org p:
; User:        ptolemy
; Date:        Fri May 17 18:27:33 1996
; Target:      default-CG56
; Universe:    mcoddelay
        ori    #03,mr         ;disable interrupts
        include '/disk3/ptolemy/ptolemy/lib/cg56/intequlc.asm'
        include '/disk3/ptolemy/ptolemy/lib/cg56/ioequlc.asm'
; initialization code from star mcoddelay.Const1 (class CG56Const)
        org    x:0
        dc     0.299999952316284
        org    p:
; initialization code from star mcoddelay.Fork.output=21 (class AnyAsmFork)
; initialization code from star mcoddelay.Delay1 (class CG56Delay)
; initialization for porthole mcoddelay.Delay1.input
        org    x:0
        dc     0.0
        org p:
; initialize delay star
; pointer to internal buffer
        org    y:5
        dc     0
        org    p:
; initialization code from star mcoddelay.Add.input=21 (class CG56Add)
; initialization for porthole mcoddelay.Add.input=21.input
        org    x:0
        dc     0.0
; initialization for porthole mcoddelay.Add.input=21.input
        org    x:2
        dc     0.0
        org p:
; initialization code from star mcoddelay.BlackHole1 (class AnyAsmBlackHole)
; initialization for porthole mcoddelay.BlackHole1.input
        org    x:1
        dc     0.0
        org p:
```

```
      do    #10,LOOP_0
; code from star mcoddelay.Const1 (class CG56Const)
; code from star mcoddelay.Fork.output=21 (class AnyAsmFork)
; code from star mcoddelay.Delay1 (class CG56Delay)
        move    x:0,x1
        move    y:5,r0
        move    #5-1,m0
        move    y:(r0),y0
        move    x1,y:(r0)+
        move    r0,y:5
        move    y0,x:2
        move    #-1,m0
; code from star mcoddelay.Add.input=21 (class CG56Add)
      move  x:0,x0       ; 1st input -> x0
      move  x:2,a        ; 2nd input -> a
      add   x0,a
      move  a,x:1        ; this move saturates
; code from star mcoddelay.BlackHole1 (class AnyAsmBlackHole)
      nop
; prevent two endloops in a row
LOOP_0
;  ---------------------- Symmetric memory map:
;  ---------------------- x memory map:
;    Loc  0,  length  1,  port  mcoddelay.Fork.output=21(input),  type  ANYTYPE
(circular)
;  Loc 1, length 1, port mcoddelay.BlackHole1(input), type ANYTYPE (circular)
;  Loc 2, length 1, port mcoddelay.Add.input=21(input)
;  ---------------------- y memory map:
;  Loc 0, length 5, state mcoddelay.Delay1(delayBuf), type FIXARRAY (circular)
;  Loc 5, length 1, state mcoddelay.Delay1(delayBufStart), type INT
```

E.2 ECO.

La figura E.2 presenta el programa visual, donde se aprecia nuevamente el uso de la estrella de la línea de retraso. Para la realimentación se utiliza un retraso convencional. El código generado se presenta en el listado E.2.

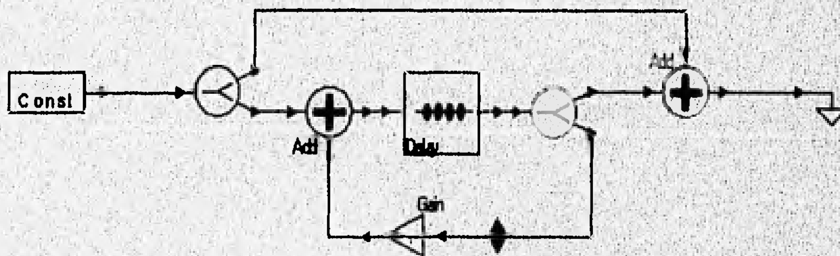


Fig. E.2

Listado E.2

```
     org p:
; User:       ptolemy
; Date:       Fri May 17 18:23:09 1996
; Target:     default-CG56
; Universe:   mcodeco
     ori    #03,mr      ;disable interrupts
     include '/disk3/ptolemy/ptolemy/lib/cg56/intequlc.asm'
     include '/disk3/ptolemy/ptolemy/lib/cg56/ioequlc.asm'
; initialization code from star mcodeco2.Const1 (class CG56Const)
     org    x:0
     dc     0.299999952316284
     org    p:
; initialization code from star mcodeco2.Fork.output=21 (class AnyAsmFork)
; initialization code from star mcodeco2.Add.input=21 (class CG56Add)
; initialization for porthole mcodeco2.Add.input=21.input
     org    x:0
     dc     0.0
; initialization for porthole mcodeco2.Add.input=21.input
     org    x:3
     dc     0.0
     org p:
; initialization code from star mcodeco2.Fork.output=22 (class AnyAsmFork)
; initialization for porthole mcodeco2.Fork.output=22.input
     org    x:1
     dc     0.0
     org p:
; initialization code from star mcodeco2.Gain1 (class CG56Gain)
; initialization for porthole mcodeco2.Gain1.input
     org    x:1
     dc     0.0
     org p:
; initialization code from star mcodeco2.Add.input=22 (class CG56Add)
; initialization for porthole mcodeco2.Add.input=22.input
     org    x:1
     dc     0.0
; initialization for porthole mcodeco2.Add.input=22.input
     org    x:0
     dc     0.0
     org p:
; initialization code from star mcodeco2.BlackHole1 (class AnyAsmBlackHole)
; initialization for porthole mcodeco2.BlackHole1.input
     org    x:2
     dc     0.0
     org p:
; initialization code from star mcodeco2.Delay1 (class CG56Delay)
; initialization for porthole mcodeco2.Delay1.input
     org    x:4
     dc     0.0
     org p:
; initialize delay star
; pointer to internal buffer
     org    y:5
     dc     0
     org    p:
     do     #10,LOOP_0
; code from star mcodeco2.Gain1 (class CG56Gain)
     move   x:1,x1
     move   #0.5,y1
     mpyr   x1,y1,a
     move   a,x:3
```

```
; code from star mcodeco2.Const1 (class CG56Const)
; code from star mcodeco2.Fork.output=21 (class AnyAsmFork)
; code from star mcodeco2.Add.input=21 (class CG56Add)
        move  x:0,x0        ; 1st input -> x0
        move  x:3,a         ; 2nd input -> a
        add   x0,a
        move  a,x:4         ; this move saturates
; code from star mcodeco2.Delay1 (class CG56Delay)
          move    x:4,x1
          move    y:5,r0
          move    #5-1,m0
          move    y:(r0),y0
          move    x1,y:(r0)+
          move    r0,y:5
          move    y0,x:1
          move    #-1,m0
; code from star mcodeco2.Fork.output=22 (class AnyAsmFork)
; code from star mcodeco2.Add.input=22 (class CG56Add)
        move  x:1,x0        ; 1st input -> x0
        move  x:0,a         ; 2nd input -> a
        add   x0,a
        move  a,x:2         ; this move saturates
; code from star mcodeco2.BlackHole1 (class AnyAsmBlackHole)
        nop
; prevent two endloops in a row
LOOP_0
;  --------------------- Symmetric memory map:
;  --------------------- x memory map:
;   Loc 0, length 1, port mcodeco2.Fork.output=21(input), type ANYTYPE
(circular)
;   Loc 1, length 1, port mcodeco2.Fork.output=22(input), type ANYTYPE
(circular)
;  Loc 2, length 1, port mcodeco2.BlackHole1(input), type ANYTYPE (circular)
;  Loc 3, length 1, port mcodeco2.Add.input=21(input
;  Loc 4, length 1, port mcodeco2.Delay1(input), type FIX
;  --------------------- y memory map:
;  Loc 0, length 5, state mcodeco2.Delay1(delayBuf), type FIXARRAY (circular)
;  Loc 5, length 1, state mcodeco2.Delay1(delayBufStart), type INT
```

E.3 TREMOLO.

La figura E.3 muestra el programa visual creado para la generación del microcódigo correspondiente al trémolo. El detalle más importante a observar es el uso de una estrella llamada *tone* que genera una senoidal o una cosenoidal (según se defina en sus parámetros), cuya amplitud también es un parámetro modificable por el usuario. El código se muestra en el listado E.3.

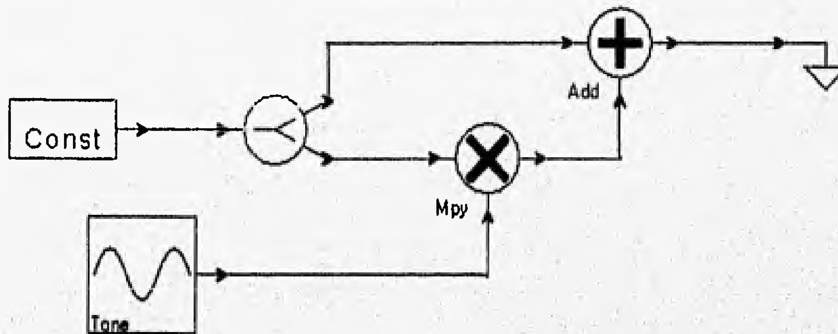


Fig. E.3

Listado E.3

```
      org p:
; User:       ptolemy
; Date:       Fri May 17 18:33:23 1996
; Target:     default-CG56
; Universe:   mcodtremolo
      ori   #03,mr      ;disable interrupts
      include '/disk3/ptolemy/ptolemy/lib/cg56/intequlc.asm'
      include '/disk3/ptolemy/ptolemy/lib/cg56/ioequlc.asm'
; initialization code from star mcodtremolo.Const1 (class CG56Const)
      org   x:0
      dc    0.299999952316284
      org   p:
; initialization code from star mcodtremolo.Fork.output=21 (class AnyAsmFork)
; initialization code from star mcodtremolo.Tone1 (class CG56Tone)
; initialization for state mcodtremolo.Tone1.state1
      org   y:0
      dc    0.0
; initialization for state mcodtremolo.Tone1.state2
      org   x:2
      dc    0.0626665353775024
      org p:
; initialization code from star mcodtremolo.Mpy.input=21 (class CG56Mpy)
; initialization for porthole mcodtremolo.Mpy.input=21.input
      org   x:0
      dc    0.0
; initialization for porthole mcodtremolo.Mpy.input=21.input
      org   x:3
      dc    0.0
      org p:
; initialization code from star mcodtremolo.Add.input=21 (class CG56Add)
; initialization for porthole mcodtremolo.Add.input=21.input
      org   x:0
      dc    0.0
; initialization for porthole mcodtremolo.Add.input=21.input
      org   x:4
      dc    0.0
      org p:
; initialization code from star mcodtremolo.BlackHole1 (class AnyAsmBlackHole)
; initialization for porthole mcodtremolo.BlackHole1.input
```

```
      org   x:1
      dc    0.0
      org p:
      do    #10,LOOP_0
; code from star mcodtremolo.Tone1 (class CG56Tone)
      move  x:2,x1
      move  y:0,a
      move  #0.992114663124084,x0
      mac   -x1,x0,a    x1,x:3
      neg   a
      mac   x1,x0,a           x1,y:0
      move  a,x:2
; code from star mcodtremolo.Const1 (class CG56Const)
; code from star mcodtremolo.Fork.output=21 (class AnyAsmFork)
; code from star mcodtremolo.Mpy.input=21 (class CG56Mpy)
      move  x:0,x0        ; 1st input -> x0
        move      x:3,y0          ; 2nd input -> y0
      mpy   x0,y0,a
      move  a,x:4
; code from star mcodtremolo.Add.input=21 (class CG56Add)
      move  x:0,x0       ; 1st input -> x0
      move  x:4,a        ; 2nd input -> a
      add   x0,a
      move  a,x:1        ; this move saturates
; code from star mcodtremolo.BlackHole1 (class AnyAsmBlackHole)
      nop
; prevent two endloops in a row
LOOP_0
; --------------------- Symmetric memory map:
; --------------------- x memory map:
;  Loc 0, length 1, port mcodtremolo.Fork.output=21(input), type ANYTYPE
(circular)
;   Loc 1, length 1, port mcodtremolo.BlackHole1(input), type ANYTYPE
(circular)
; Loc 2, length 1, state mcodtremolo.Tone1(state2), type FIX
; Loc 3, length 1, port mcodtremolo.Mpy.input=21(input
; Loc 4, length 1, port mcodtremolo.Add.input=21(input
; --------------------- y memory map:
; Loc 0, length 1, state mcodtremolo.Tone1(state1), type FIX
```

E.4 REVERB.

En la paleta de estrellas de procesamiento de señales del dominio CG56 existen dos estrellas que implantan directamente los filtros paso-todo (allpass) y de peine (comb) o retraso realimentado, que forman parte de un algoritmo de reverberación (ver Cap 6.3.5). Estas estrellas requieren dos entradas: la primera es la señal de entrada y la segunda es el tiempo de retraso; para esta aplicación, el tiempo de retraso lo constituyen constantes. Integrando estos elementos en el programa visual de la figura E.4, se sintetizó el código del listado E.4.

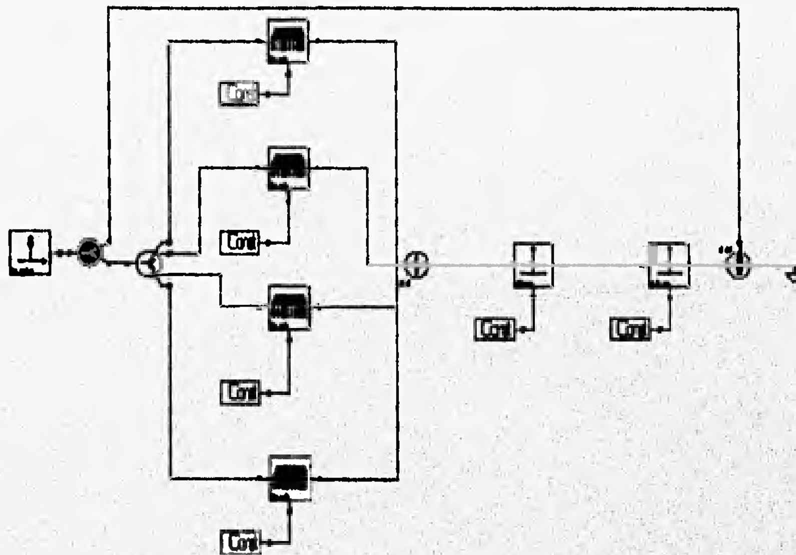


Fig. E.4

Listado E.4

```
     org p:
; User:       ptolemy
; Date:       Mon May 20 15:04:04 1996
; Target:     default-CG56
; Universe:   mcodreverb
     ori    #03,mr       ;disable interrupts
     include '/disk3/ptolemy/ptolemy/lib/cg56/intequlc.asm'
     include '/disk3/ptolemy/ptolemy/lib/cg56/ioequlc.asm'
; initialization code from star mcodreverb.Impulse1 (class CG56Impulse)
; initialization for state mcodreverb.Impulse1.pulse
     org    y:10
     dc     0.99999988079071
     org p:
; initialization code from star mcodreverb.Fork.output=41 (class AnyAsmFork)
; initialization for porthole mcodreverb.Fork.output=41.input
     org    x:1
     dc     0.0
     org p:
; initialization code from star mcodreverb.Comb1 (class CG56Comb)
; initialization for state mcodreverb.Comb1.state
     org    y:11
     dc     0.0
; initialization for porthole mcodreverb.Comb1.input
     org    x:1
     dc     0.0
     org p:
; initialize comb
; pointer to internal buffer
     org    y:12
     dc     0
```

```
     org   p:
; initialization code from star mcodreverb.Comb2 (class CG56Comb)
; initialization for state mcodreverb.Comb2.state
     org   y:13
     dc    0.0
; initialization for porthole mcodreverb.Comb2.input
     org   x:1
     dc    0.0
     org p:
; initialize comb
; pointer to internal buffer
     org   y:14
     dc    16
     org   p:
; initialization code from star mcodreverb.Comb3 (class CG56Comb)
; initialization for state mcodreverb.Comb3.state
     org   y:15
     dc    0.0
; initialization for porthole mcodreverb.Comb3.input
     org   x:1
     dc    0.0
     org p:
; initialize comb
; pointer to internal buffer
     org   y:26
     dc .  32
     org   p:
; initialization code from star mcodreverb.Comb4 (class CG56Comb)
; initialization for state mcodreverb.Comb4.state
     org   y:27
     dc    0.0
; initialization for porthole mcodreverb.Comb4.input
     org   x:1
     dc    0.0
     org p:
; initialize comb
; pointer to internal buffer
     org   y:28
     dc    48
     org   p:
; initialization code from star mcodreverb.Add1 (class CG56Add)
; initialization for porthole mcodreverb.Add1.input
     org   x:6
     dc    0.0
; initialization for porthole mcodreverb.Add1.input
     org   x:7
     dc    0.0
; initialization for porthole mcodreverb.Add1.input
     org   x:8
     dc    0.0
; initialization for porthole mcodreverb.Add1.input
     org   x:9
     dc    0.0
     org p:
; initialization code from star mcodreverb.Const1 (class CG56Const)
     org   x:2
     dc    0.0010000467300415
     org   p:
; initialization code from star mcodreverb.Const2 (class CG56Const)
     org   x:3
     dc    0.0010000467300415
```

```
     org   p:
; initialization code from star mcodreverb.Const3 (class CG56Const)
     org   x:4
     dc    0.0010000467300415
     org   p:
; initialization code from star mcodreverb.Const4 (class CG56Const)
     org   x:5
     dc    0.0010000467300415
     org   p:
; initialization code from star mcodreverb.Allpass1 (class CG56Allpass)
; initialization for porthole mcodreverb.Allpass1.input
     org   x:10
     dc    0.0
     org p:
; initialize allpass
; pointer to internal buffer
       org     y:29
       dc      64
       org     p:
; initialization code from star mcodreverb.Allpass2 (class CG56Allpass)
; initialization for porthole mcodreverb.Allpass2.input
     org   x:12
     dc    0.0
     org p:
; initialize allpass
; pointer to internal buffer
       org     y:30
       dc      80
       org     p:
; initialization code from star mcodreverb.BlackHole1 (class AnyAsmBlackHole)
; initialization for porthole mcodreverb.BlackHole1.input
     org   x:0
     dc    0.0
     org p:
; initialization code from star mcodreverb.Const5 (class CG56Const)
     org   x:11
     dc    0.0010000467300415
     org   p:
; initialization code from star mcodreverb.Const6 (class CG56Const)
     org   x:13
     dc    0.0010000467300415
     org   p:
; initialization code from star mcodreverb.Add.input=21 (class CG56Add)
; initialization for porthole mcodreverb.Add.input=21.input
     org   x:14
     dc    0.0
; initialization for porthole mcodreverb.Add.input=21.input
     org   x:1
     dc    0.0
     org p:
; initialization code from star mcodreverb.Fork.output=21 (class AnyAsmFork)
; initialization for porthole mcodreverb.Fork.output=21.input
     org   x:1
     dc    0.0
     org p:
     do    #10,LOOP_0
; code from star mcodreverb.Const1 (class CG56Const)
; code from star mcodreverb.Const2 (class CG56Const)
; code from star mcodreverb.Const3 (class CG56Const)
; code from star mcodreverb.Const4 (class CG56Const)
; code from star mcodreverb.Const5 (class CG56Const)
```

```
; code from star mcodreverb.Const6 (class CG56Const)
; code from star mcodreverb.Impulse1 (class CG56Impulse)
    clr   b      y:10,a
    move  a,x:1
    move  b,y:10
; code from star mcodreverb.Fork.output=21 (class AnyAsmFork)
; code from star mcodreverb.Fork.output=41 (class AnyAsmFork)
; code from star mcodreverb.Comb1 (class CG56Comb)
      move    x:1,b
      move    y:12,r0
      move    #>10-1,m0
      move    y:(r0),a
; go ahead and output the result
      move    a,x:9
; oldest data sample is now in a. filter it with the LPF
      move    y:11,y0
      move    #>0.5,x0
      mac     x0,y0,a
; store the state
      move    a,y:11  a,x0
; multiply it by the feedback constant, which is time*(1-pole)
      move    x:2,y0
      mpy     x0,y0,a #>0.5,y0
      move    a,x0
      mac     x0,y0,b
      move    b,y:(r0)+
      move    r0,y:12
      move    m1,m0

; code from star mcodreverb.Comb2 (class CG56Comb)
      move    x:1,b
      move    y:14,r0
      move    #>10-1,m0
      move    y:(r0),a
; go ahead and output the result
      move    a,x:8
; oldest data sample is now in a. filter it with the LPF
      move    y:13,y0
      move    #>0.5,x0
      mac     x0,y0,a
; store the state
      move    a,y:13  a,x0
; multiply it by the feedback constant, which is time*(1-pole)
      move    x:3,y0
      mpy     x0,y0,a #>0.5,y0
      move    a,x0
      mac     x0,y0,b
      move    b,y:(r0)+
      move    r0,y:14
      move    m1,m0

; code from star mcodreverb.Comb3 (class CG56Comb)
      move    x:1,b
      move    y:26,r0
      move    #>10-1,m0
      move    y:(r0),a
; go ahead and output the result
      move    a,x:7
; oldest data sample is now in a. filter it with the LPF
      move    y:15,y0
      move    #>0.5,x0
```

```
        mac     x0,y0,a
; store the state
        move    a,y:15  a,x0
; multiply it by the feedback constant, which is time*(1-pole)
        move    x:4,y0
        mpy     x0,y0,a #>0.5,y0
        move    a,x0
        mac     x0,y0,b
        move    b,y:(r0)+
        move    r0,y:26
        move    m1,m0

; code from star mcodreverb.Comb4 (class CG56Comb)
        move    x:1,b
        move    y:28,r0
        move    #>10-1,m0
        move    y:(r0),a
; go ahead and output the result
        move    a,x:6
; oldest data sample is now in a. filter it with the LPF
        move    y:27,y0
        move    #>0.5,x0
        mac     x0,y0,a
; store the state
        move    a,y:27  a,x0
; multiply it by the feedback constant, which is time*(1-pole)
        move    x:5,y0
        mpy     x0,y0,a #>0.5,y0
        move    a,x0
        mac     x0,y0,b
        move    b,y:(r0)+
        move    r0,y:28
        move    m1,m0

; code from star mcodreverb.Add1 (class CG56Add)
     move  x:6,x0       ; 1st input -> x0
     move  x:7,a        ; 2nd input -> a
     add   x0,a  x:8,x0
     add   x0,a  x:9,x0
     add   x0,a
     move  a,x:10             ; this move saturates
; code from star mcodreverb.Allpass1 (class CG56Allpass)
        move    x:10,b
        move    y:29,r0
        move    #>10-1,m0
        move    x:11,x0
        move    y:(r0),y0
        mac     -x0,y0,b        y0,a
        move    b,x1    b,y:(r0)+
        mac     x0,x1,a
        move    a,x:12
        move    r0,y:29
; code from star mcodreverb.Allpass2 (class CG56Allpass)
        move    x:12,b
        move    y:30,r0
        move    #>10-1,m0
        move    x:13,x0
        move    y:(r0),y0
        mac     -x0,y0,b        y0,a
        move    b,x1    b,y:(r0)+
        mac     x0,x1,a
```

```
        move    a,x:14
        move    r0,y:30
; code from star mcodreverb.Add.input=21 (class CG56Add)
      move  x:14,x0      ; 1st input -> x0
      move  x:1,a        ; 2nd input -> a
      add   x0,a
      move  a,x:0        ; this move saturates
; code from star mcodreverb.BlackHole1 (class AnyAsmBlackHole)
      nop
; prevent two endloops in a row
LOOP_0
;  --------------------- Symmetric memory map:
;  --------------------- x memory map:
;  Loc 0, length 1, port mcodreverb.BlackHole1(input), type ANYTYPE (circular)
;   Loc  1,  length  1,  port  mcodreverb.Fork.output=21(input),  type  ANYTYPE
(circular)
;  Loc 2, length 1, port mcodreverb.Comb1(time), type FIX
;  Loc 3, length 1, port mcodreverb.Comb2(time), type FIX
;  Loc 4, length 1, port mcodreverb.Comb3(time), type FIX
;  Loc 5, length 1, port mcodreverb.Comb4(time), type FIX
;  Loc 6, length 1, port mcodreverb.Add1(input
;  Loc 7, length 1, port mcodreverb.Add1(input
;  Loc 8, length 1, port mcodreverb.Add1(input
;  Loc 9, length 1, port mcodreverb.Add1(input
;  Loc 10, length 1, port mcodreverb.Allpass1(input), type FIX
;  Loc 11, length 1, port mcodreverb.Allpass1(polezero), type FIX
;  Loc 12, length 1, port mcodreverb.Allpass2(input), type FIX
;  Loc 13, length 1, port mcodreverb.Allpass2(polezero), type FIX
;  Loc 14, length 1, port mcodreverb.Add.input=21(input
;  --------------------- y memory map:
;    Loc  0,  length  10,  state  mcodreverb.Comb1(delayBuf),  type  FIXARRAY
(circular)
;  Loc 10, length 1, state mcodreverb.Impulse1(pulse), type FIX
;  Loc 11, length 1, state mcodreverb.Comb1(state), type FIX
;  Loc 12, length 1, state mcodreverb.Comb1(delayBufStart), type INT
;  Loc 13, length 1, state mcodreverb.Comb2(state), type FIX
;  Loc 14, length 1, state mcodreverb.Comb2(delayBufStart), type INT
;  Loc 15, length 1, state mcodreverb.Comb3(state), type FIX
;    Loc  16,  length  10,  state  mcodreverb.Comb2(delayBuf),  type  FIXARRAY
(circular)
;  Loc 26, length 1, state mcodreverb.Comb3(delayBufStart), type INT
;  Loc 27, length 1, state mcodreverb.Comb4(state), type FIX
;  Loc 28, length 1, state mcodreverb.Comb4(delayBufStart), type INT
;  Loc 29, length 1, state mcodreverb.Allpass1(delayBufStart), type INT
;  Loc 30, length 1, state mcodreverb.Allpass2(delayBufStart), type INT
;    Loc  32,  length  10,  state  mcodreverb.Comb3(delayBuf),  type  FIXARRAY
(circular)
;    Loc  48,  length  10,  state  mcodreverb.Comb4(delayBuf),  type  FIXARRAY
(circular)
;   Loc  64,  length  10,  state  mcodreverb.Allpass1(delayBuf),  type  FIXARRAY
(circular)
;   Loc  80,  length  10,  state  mcodreverb.Allpass2(delayBuf),  type  FIXARRAY
(circular)
```

Glosario.

A/D Conversión Analógica-Digital. Proceso mediante el cual se convierte una señal analógica en digital. Implica la discretización y cuantización.

Aliasing Fenómeno de "enmascaramiento" que se presenta cuando la frecuencia de muestreo es menor al doble de la frecuencia más alta de la señal muestreada.

ASIC Circuito Integrado de Aplicación Específica (*Application Specific Integrated Circuit*).

CAD Diseño Asistido por Computadora (*Computer Aided Design*).

Chorus Efecto de audio que consiste en simular la presencia de una o más fuentes de sonido a partir de la fuente o señal original.

D/A Conversión Digital-Analógica. Proceso mediante el cual se convierte una secuencia de datos numéricos (señal digital) en una señal continua o analógica.

Deadlock Concepto de los sistemas operativos que se refiere al bloqueo circular entre procesos que solicitan recursos que están siendo ocupados por otros procesos. (Ej.: el proceso 1 está siendo ejecutado, se encuentra utilizando el recurso A y para continuar necesita del recurso B, que a su vez está siendo utilizado por el proceso 2, el cual para continuar requiere del recurso A)

DFT Transformada Discreta de Fourier (*Discrete Fourier Transform*, v. Cap. 1)

DSP Procesamiento Digital de Señales (*Digital Signal Processing*)

Feedback Retroalimentación o realimentación. Concepto de análisis de sistemas que implica la adición de la salida del sistema a la excitación o entrada del mismo.

FFT Transformada Rápida de Fourier (*Fast Fourier Transform*).

FIR Filtro digital con respuesta al impulso finita (*Finite Impulse Response*)

Flanger Efecto de audio basado en el coro (v. Chorus), que utiliza tiempos de retraso menores, además de realimentación. La percepción auditiva del efecto es similar al sonido de un motor de retropopulsión (jet).

IIR Filtro digital con respuesta al impulso infinita (*Infinite Impulse Response*).

Khoros Ambiente de análisis y simulación de sistemas desarrollado por la Universidad de Nuevo México.

Latencia Es el retraso de tiempo inherente entre la entrada y la salida de datos de un sistema de procesamiento de información.

LIT Siglas de Lineal e Invariante con el Tiempo.

Loop Ciclo o bucle. Estructura de un algoritmo en el que el control del procedimiento llega a un punto de éste y regresa al inicio.

Mathematica

Sistema de Algebra por Computadora que permite resolver problemas de ciencias básicas e ingeniería en forma numérica y simbólica.

OOA Análisis Orientado a Objetos (*Object Oriented Analysis*). Terminología usada para definir la fase de análisis de la metodología orientada a objetos (v. OOP), donde se definen las clases y los objetos básicos de la metodología.

OOD Diseño Orientado a Objetos (*Object Oriented Design*). Terminología usada para definir la fase de diseño de la metodologñia orientada a objetos (v. OOP), que refina la fase de OOA con los detalles adicionales de diseño e implantación.

OOP Programación Orientada a Objetos (*Object Oriented Programming*). Metodología de desarrollo de software que se basa en la definición de elementos primarios llamados objetos y métodos y propiedades hereditarias asociadas a éstos.

Overhead Sobrecarga. Es el tiempo y los recursos utilizados por un sistema operativo para administrarse a sí mismo.

Pitch Es la frecuencia fundamental de una onda de sonido. Se conoce en español como tono.

Prototyping Es una metodología de desarrollo de sistemas de procesamiento que se basa en la construcción rápida de un prototipo de solución a partir de especificaciones iniciales y el cual se irá refinando hasta llegar a la versión final.

Ptolemy Es un ambiente de diseño que soporta simultáneamente una mezcla de diferentes modelos de cálculo. Ha sido desarrollado en la Universidad de California en Berkeley.

Pyramid Es un ambiente interactivo de diseño automatizado de circuitos integrados para aplicaciones DSP. Consta de tres partes o subambientes: el Ambiente de Síntesis, el Ambiente de Configuración y el Ambiente de Generación de Módulos.

Reverb Reverberación. Efecto que simula el fenómeno que se presenta cuando en un lugar cerrado se producen múltiples reflexiones de una onda sonora y que provocan en el receptor la sensación de espacio.

SNR Razón de Señal a Ruido (*Signal to Noise Ratio*, v. Cap 5).

SOTR Sistemas Operativos en Tiempo Real.

SPL Nivel de Presión de Sonido (*Sound Pressure Level* , v. Cap. 5)

Trémolo Efecto que produce una variación cíclica en la amplitud de una señal de audio.

VSC VSC es un conjunto de herramientas de análisis y simulación de arquitecturas dedicadas.

VHDL Es un lenguaje estandarizado de descripción de hardware para especificar diseños con múltiples niveles de abstracción.

Referencias Bibliograficas y Hemerográficas.

PROCESAMIENTO DIGITAL DE SEÑALES.

[ALC88] Alcántara, Rogelio. *Apuntes de Procesamiento Digital de Señales*. DEPFI-UNAM, México, 1988.

[CAD85] Cadzow, James A. y Van Landingham, Hugh. *Signals, Systems and Transforms*. Prentice-Hall, Estados Unidos, 1985.

[DAT91] Datta, Jayant. "Digital Signal Processing: Theory" en *Advanced Digital Audio*, Ken C. Pohlman editor. Sams, Estados Unidos, 1991. pp. 293-346.

[DIE94] División de Ingenieria Eléctrica, Facultad de Ingenieria, UNAM. *Apuntes de Control Digital*. FI-UNAM, México, 1994.

[OPP90] Oppenheim, Alan V. *Introduction to Signals and Systems*. Prentice-Hall, Estados Unidos, 1990.

[PSE95] Psenicka, Bohumil. *Apuntes de Procesamiento Digital de Señales*. FI-UNAM, México, 1995.

[PRO88] Proakis, John G. y Manolakis, Dimitris G. *Digital Signal Processing; Principles, Algorithms, and Applications*. MacMillan, Estados Unidos, 1988.

[SMI85] Smith, Julius O. "An Introduction to Digital Filter Theory" en *Digital Audio Signal Processing; an Anthology*, John Strawn editor. William Kaufman, Estados Unidos, 1985. pp. 69-136.

HERRAMIENTAS DE SOFTWARE.

[EVA90] Evans, Brian, et. al. "Symbolic Z Transforms Using DSP Knowledge Bases", en *Proceedings ICASSP-90*. IEEE, Estados Unidos, 1990. pp. 1175-1179.

[EVA93] Evans, Brian, et. al. "Learning Signals and Systems with Mathematica". IEEE Transactions on Education. Vol. 36. No. 1. IEEE, Estados Unidos, 1993.

[FRA88] Frank, Geoffrey A. y McLin, David M. "Systems to Silicon Design and Assessment", en *VLSI Signal Processing III*, IEEE Press, Estados Unidos, 1988.

[HUI88] Huisken, J.A. et. al. "Design of DSP Systems on Silicon Using the Pyramid Library and Design Tools", en *VLSI Signal Processing III*, IEEE Press, Estados Unidos, 1988.

[MAT96] *Matlab*; Product Catalog. Mathworks, Estados Unidos, 1996.

[MIL94] Milne, George. *Formal Specification and Verification of Digital Systems*. McGraw-Hill, Reino Unido, 1994.

[ZEP95] Zepeda, Claudia. Tesis de licenciatura: *Sistema de Clasificación de Señales EEG Utilizando el Ambiente de Programación Visual Khoros*. Facultad de Ciencias Físico Matemáticas, BUAP, México, 1995.

[WOL88] Wolfram, S. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley, Estados Unidos, 1988.

PTOLEMY.

[BUC94] Buck, Joseph et. al. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". Department of Electrical Engineering and Computer Science, University of California, Berkeley, Estados Unidos, 1994.

[PIN95] Pino, Jose Luis et. al. "Software Synthesis for DSP using Ptolemy". *Journal of VLSI Signal Processing*, vol. 9, pp. 7-21. Kluwer Academic Publishers, Estados Unidos, 1995.

[PUM95] *Ptolemy User's Manual*. College of Engineering; Department of Electrical Engineering and Computer Science, University of California at Berkeley, Estados Unidos, 1995.

INGENIERÍA DE SOFTWARE.

[GUZ89] Guzman Robles, Luz Maria y Villagran Araiza, Manuel Agustin. Tesis de Licenciatura: "El uso del prototyping como metodología en el desarrollo de sistemas y su aplicación sobre lenguajes de cuarta generación". UNAM, F.I., México. 1989.

[HAR90] Harel, D. "STATEMATE: A Working Enviroment for the Development of Complex Reactive Systems", IEEE Transactions Software Engineering, vol. 16, no. 3, April 1990.

[MCC85] McCabe, T. J. et.al. "Structured Real-Time Analysis and Design", COMPSAC-85, IEEE, Octubre de 1985.

[PRE93] Pressman, Roger S. *Ingeniería del Software: un enfoque Práctico*, 3ª ed. McGraw-Hill, México, 1993.

[SOM88] Sommerville, Ian. *Ingeniería de software.* 2a. ed. Addison-Wesley, México, 1988.

[YOU93] Yourdon, Edward. *Análisis Estructurado Moderno.* Prentice-Hall Hispanoamérica, México, 1993.

METODOLOGÍAS DE DESARROLLO DE HARDWARE Y DE PROYECTOS TECNOLÓGICOS.

[ALC92] Alcántara, Rogelio. "Metodología de diseño de sistemas para el procesamiento digital de señales". DEPFI-UNAM, México, 1992.

[DEF88] DeFatta, David J. et.al. *Digital Signal Processing. A system design approach.* Wiley, Estados Unidos, 1988.

[GON92] Gonzalez Villela, Victor Javier. Tesis de maestría: *Metodología de diseño electrónico en un proyecto de desarrollo tecnológico.* Facultad de Ingeniería-UNAM, México, 1995.

[KAR92] Karjalainen, Matti. "Object -Oriented Programming of DSP Processors: A Case Study of Quick C30", *Proceedings ICASSP-92*, vol. 5. IEEE, Estados Unidos, 1992.

[KRI89] Krick, Edward V. *Introducción a la ingeniería y al diseño en la ingeniería.* 2a. ed. Limusa, México, 1989.

EL AUDIO DIGITAL Y SUS APLICACIONES.

[BLE78] Blesser, Barry y Kates, James M. "Digital Processing in Audio Signals" en *Applications of Digital Signal Processing*. Alan V. Oppenheim, Editor. Prentice-Hall, Estados Unidos, 1978.

[POH89] Pohlmann, Ken C. *Principles of Digital Audio*. SAMS, Estados Unidos, 1989.

[TAL94] Talbot, Smith Michael. *Audio Recording and Audio Reproduction; Practical Measures for Audio Enthusiasts*. Newnes, Gran Bretaña, 1994.

INSTRUMENTOS MUSICALES.

[AND84] Anderton, Craig. *Guitar Gadgets*. Amsco, Estados Unidos, 1984.

[HER81] Hermosa Donate, Antonio. *El moderno órgano electrónico; circuitería y sistemas*. Marcombo, España, 1981.