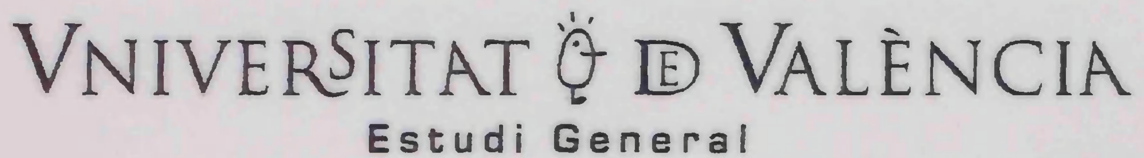
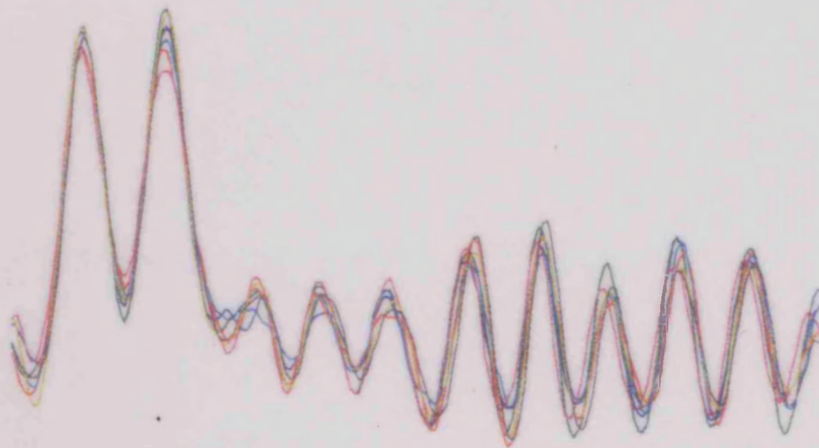


Facultad de C.C. Físicas
Departamento de Ingeniería Electrónica



cies
ncia

Sistema de Pesado Dinámico Basado en DSP y Bus CAN con Núcleo de Sistema Operativo de Tiempo Real

UNIVERSITAT DE VALÈNCIA
Biblioteca
80001792447

TESIS DOCTORAL

Jose V. Francés Villora

Valencia, 2000

UMI Number: U607758

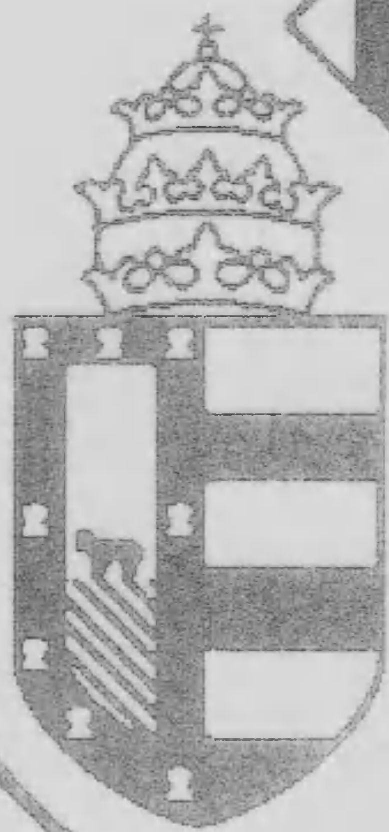
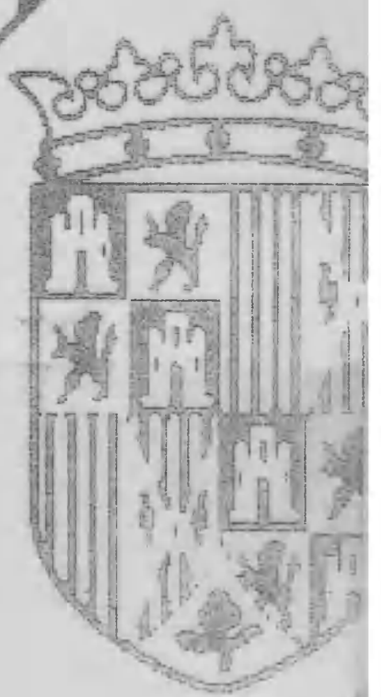
All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U607758
Published by ProQuest LLC 2014. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346



DEI GRA REX ARAGONVM · ALEXA
FERDINANDVS

TESIS DOCTORAL n° 391

20/7/2000

Físicas

T.D

391

VNIVERSITAT DE VALÈNCIA
Estudi General
Facultad de C.C. Físicas
Departamento de Ingeniería Electrónica

Físicas
T.D
391

Sistema de Pesado Dinámico Basado en DSP y bus CAN Con Núcleo de Sistema Operativo de Tiempo Real

Directores:
Dr. Javier Calpe Maravilla
Dr. Juan Francisco Guerrero Martínez

UNIVERSITAT DE VALÈNCIA
REGISTRE GENERAL
ENTRADA
29 MAYO 2000
N.º 71.153
HORA
OFICINA AUXILIAR NÚM. 16

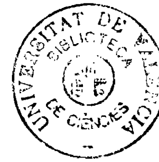
TESIS DOCTORAL
José Vicente Francés Villora
Valencia, 2000

UNIVERSITAT DE VALÈNCIA
BIBLIOTECA CIÈNCIES

Nº Registre 15.011
DATA 13.9.2000
SIGNATURA T.D.391(FISICAS)

Nº LIBIS: i 20182843

24 cm.



Dpto. Ingeniería Electrónica
Facultad de Física

D. JAVIER CALPE MARAVILLA, Doctor en Física, Profesor Titular de Universidad del Departamento de Ingeniería Electrónica de la Facultad de Física de la Universidad de Valencia y JUAN FRANCISCO GUERRERO MARTÍNEZ, Doctor en Física, Profesor Titular de Universidad del Departamento de Ingeniería Electrónica de la Facultad de Física de la Universidad de Valencia

HACEMOS CONSTAR:

QUE el Licenciado en Física D. José Vte. Francés Víllora ha realizado bajo nuestra dirección, en los laboratorios del Departamento de Ingeniería Electrónica, el trabajo titulado "SISTEMA DE PESADO DINÁMICO BASADO EN DSP Y BUS CAN CON NÚCLEO DE SISTEMA OPERATIVO EN TIEMPO REAL", que se presenta en esta memoria para optar al grado de Doctor en Ingeniería Electrónica.

Y para que así conste a los efectos oportunos, firmamos el presente certificado, en Valencia, a 23 de Abril de 2000.

Juan Fco. Guerrero Martínez   Javier Calpe Maravilla

Agradecimientos

Escribo buscando el final de esta página. Cuando lo encuentre, finalizará un largo e importante periodo de mi vida en el que he invertido gran cantidad de tiempo, ilusión y esfuerzo.

Han sido muchas las personas que han contribuido en la consecución de este trabajo bien en forma de colaboración, consejos y sugerencias, o con un sencillo, e inestimable, apoyo. A todas ellas quiero expresar mi reconocimiento. En primer lugar a los directores de Tesis, a J. Calpe quiero agradecer su dedicación y apoyo, encontrando para ello un tiempo que, a modo de correcaminos, no tiene; a J. Guerrero, el gran apoyo recibido desde el momento en que entré a formar parte del grupo. Por otro lado, a E. Soria su ayuda y colaboración referente a los sistemas adaptativos; a A. Serrano su visión objetiva del trabajo y las sugerencias y consejos planteados; a Gus por sus consejos sobre la edición en LATEX.

En general, quiero expresar mi reconocimiento a todos los miembros del Grupo de Procesado Digital de Señales, al cual tengo la satisfacción de pertenecer, por el ambiente de colaboración y compañerismo, que hace que hasta el más duro trabajo se convierta en algo más grato y llevadero.

A I. Llorens por su apoyo y amistad. A "Sastre" y Ana por su amistad y su "savoir-faire" en la cocina.

A Joana quiero agradecer su apoyo incondicional, sus palabras de ánimo, su actitud siempre positiva, su sonrisa, ... que aún en la distancia me hacen seguir adelante.

Finalmente, quiero agradecer a mis familiares más allegados su cariño, comprensión y apoyo. Especialmente a mis padres, pilares fundamentales sin cuyo esfuerzo y sacrificio esto no hubiese sido más que un sueño.

A mis padres
A Joana

Índice General

Lista de Figuras. vii
Lista de Tablas. xiii

1 Introducción **1**

2 Transductores **5**
2.1 Células de carga 5
2.1.1 Galgas extensométricas 6
2.1.2 El puente de Wheatstone 11
2.2 Sensores de aceleración 13
2.2.1 Introducción a las micromáquinas de superficie 14
2.2.2 Principios del acelerómetro 15
2.2.3 Usos actuales de los sensores de aceleración 19

3 El sistema de adquisición **21**
3.1 Conexionado de los transductores 21
3.2 Caracterización de la adquisición 24
3.3 Adquisición 24
3.3.1 DAQCard-AI-16XE-50 24
3.3.2 Equipamiento opcional 25
3.3.3 Descripción del hardware 25
3.3.4 DAQCard-AI-16XE-50 versus AT-MIO-16E-10 29
3.4 Programación del sistema de adquisición 30
3.4.1 NI-DAQ 30
3.4.2 El entorno de desarrollo 31
3.4.3 Técnica de adquisición continua de datos (Doble Buffer) 31
3.4.4 Interfaz de usuario 35
3.4.5 Programación de la aplicación 36
3.4.6 Análisis de la aplicación 38

3.5 Funcionamiento de la aplicación 41

4 Caracterización y algoritmos **45**

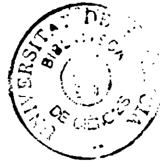
4.1 Caracterización de la señal 45
4.2 Modelización 47
4.2.1 Test a diferentes velocidades 49
4.2.2 Test de la naturaleza de las oscilaciones en las *zonas peso* . . . 51
4.2.3 Hipótesis de la generación de oscilaciones 52
4.2.4 Modelización 55
4.2.5 Justificación del ruido de la célula de test 59
4.3 Acondicionamiento para el estudio estadístico 60
4.3.1 Marcado 61
4.3.2 Troceado 62
4.3.3 Solapado 62
4.4 Preprocesado de la señal 63
4.4.1 Filtrado 63
4.4.2 Deconvolución mediante el modelo ARMA 64
4.4.3 Sistemas adaptativos 64

5 Sistema de calibración de alta velocidad **71**

5.1 Introducción 71
5.2 Sistemas calibradores 72
5.3 Objetivos 75
5.4 Introducción al sistema MAXSORTER de clasificación de productos hortofrutícolas 76
5.4.1 Elementos del sistema 76
5.4.2 El entorno de usuario 76
5.4.3 El módulo de clasificación 83
5.4.4 El módulo de pesado 83
5.4.5 El sistema de visión 84
5.4.6 El módulo de encoder 84
5.4.7 El módulo de electroimanes 86

6 El módulo de pesado **87**

6.1 Introducción 87
6.2 Comunicaciones CAN 88
6.2.1 Modos de funcionamiento 90
6.3 El procesador digital TMS320C26 93
6.3.1 Características básicas del TMS320C26 94



6.4 Lógica programable 95
6.5 Organización de memoria 98
6.5.1 Memoria interna 98
6.5.2 Memoria externa y memoria global 100
6.5.3 Mapa de memoria del sistema 103
6.6 Conversores ADC 104
6.7 Interfaz DSP-Conversores 106
6.7.1 Implementación de la comunicación serie TMS320C26-AD7730 107
6.8 Interfaz DSP-Placa de comunicaciones 113
6.9 Interfaz de comunicación RS232 a PC *host* 116

7 El sistema operativo EMMOS 119

7.1 Introducción 119
7.1.1 Sistemas operativos serie y multiprogramados 120
7.1.2 Sistemas operativos de multiprogramación 123
7.1.3 Requerimientos funcionales 125
7.2 Estructuras de datos 126
7.2.1 EPROM: Las tablas de inicialización 126
7.2.2 NVRAM: Los datos no volátiles 127
7.2.3 SRAM externa y memoria interna de datos 128
7.3 Gestión de E/S 135
7.3.1 Estrcutura de los mensajes recibidos 135
7.3.2 Estructura de los mensajes transmitidos 137
7.3.3 Gestión de la recepción de mensajes 138
7.3.4 Gestión de la transmisión de mensajes 140
7.3.5 Gestión de mensajes 142
7.4 Procesos 143
7.4.1 El concepto de proceso 143
7.4.2 El proceso desde el punto de vista del programador de sistemas 144
7.4.3 El proceso desde el punto de vista del sistema operativo 150
7.4.4 Los procesos EMMOS 153
7.5 El planificador 168
7.5.1 Tipos de planificadores 168
7.5.2 Criterios de planificación y rendimiento 170
7.5.3 Tipos de planificadores 173
7.5.4 La planificación EMMOS 178
7.5.5 Otras consideraciones 184
7.6 Gestión de memoria 186

7.7 Sincronización 187
7.8 Resumen 188

8 La Aplicación **189**
8.1 Introducción 189
8.2 Estructura 191
8.2.1 Prioridad 192
8.3 Modos de funcionamiento 193
8.3.1 Modo configuración 193
8.3.2 Tarado 195
8.3.3 Calibración 199
8.3.4 Ajuste de cero 201
8.3.5 Verificación 204
8.3.6 Osciloscopio 205
8.3.7 Clasificación 207
8.3.8 Proceso de los sincronismos 209
8.3.9 Petición y entrega de parámetros 212
8.3.10 Mensajes asíncronos 214
8.3.11 Mantenimiento 215
8.3.12 Sin modo 216

9 Resultados **217**
9.1 Selección del método de preprocesado 217
9.1.1 Criterio 1: Presencia de oscilaciones en la zona peso 218
9.1.2 Criterio 2: Dispersión respecto de un patrón promedio 221
9.1.3 Criterio 3: Convergencia del comportamiento tras la transición 223
9.2 Algoritmo para la estimación del peso 225
9.3 Uso de los acelerómetros 227
9.3.1 Generación inteligente de sincronismos 228
9.4 Resultados 230
9.4.1 Evaluación de las prestaciones de los algoritmos propuestos . . 231
9.4.2 Algoritmos implementados en tiempo real 239

10 Conclusiones y Proyección Futura **241**
10.1 Conclusiones 241
10.2 Proyecciones Futuras 246

A El bus CAN **249**

A.1 Sistema de Control basado en módulos de comunicación 250

A.2 Protocolo de comunicaciones en sistemas de control 251

A.2.1 Especificaciones del protocolo de comunicaciones 253

B El puerto serie del TMS320C26 **257**

B.1 Operaciones de transmisión y recepción. 259

C El puerto serie del AD7730 **263**

C.1 Operación de escritura 264

C.2 Operación de lectura. 264

C.3 Interconexión 265

D El fichero de Comandos **267**

E La Declaración de las Estructuras de Datos **273**

F La Inicialización del Sistema **281**

F.1 Carga y arranque del DSP 281

F.2 El proceso de carga primaria 284

F.3 El proceso de carga secundaria 285

F.4 La inicialización del sistema operativo 287

F.4.1 Inicialización de los punteros a pila 288

F.4.2 Inicialización de las variables globales 288

F.4.3 Configuración de memoria, inicialización de variables y estructuras de datos 289

F.4.4 Habilitación de las interrupciones y programación del *timer* . . 290

F.4.5 Inicialización de las comunicaciones CAN 291

F.4.6 Test y programación de los conversores 291

F.4.7 Paso del control del programa al proceso nulo 291

G Registros, contextos de interrupciones y marcos de función **293**

G.1 Registros del 'C26 293

G.2 Las interrupciones en el TMS320C26 296

G.3 El contexto en las interrupciones 297

G.4 Marcos de función 298

H La tabla de edición de procesos **303**

I Tiempos Carcaterísticos de EMMOS 307
I.1 Medida de tiempos de funciones críticas 307
I.2 Medida de tiempos de servicios de interrupción 309
I.3 Referencias temporales 310

J Mensajes, parámetros y errores 311

K Pines del bloque de conectores 321

L Diagramas de clases 323

Índice de Figuras

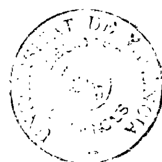
2.1 a) Elemento de esfuerzo sin tensión aplicada, y b) con tensión aplicada. . 5
2.2 Relación entre esfuerzos y deformaciones (escala de la zona elástica muy ampliada). 7
2.3 Galga impresa. Los hilos más finos son la parte activa de galga. 9
2.4 Montaje de una galga impresa. 1 Sustrato donde se monta; 2 adhesivo; 3 galga; 4 terminales para soldar; 5 soldadura; 5 hilos de conexión; 7 aislamiento protector. 10
2.5 Diversos tipos de galgas metálicas y semiconductoras (Group, 1999). . . . 11
2.6 Puente de Wheatstone. 12
2.7 Comparativa de varias tecnologías. 13
2.8 Tren de ruedas dentadas de 135μm de largo. 14
2.9 Pasos para la formación del engranaje de la figura 2.8. 15
2.10 Sistema de masa móvil usado en el interior de un sensor de aceleración. . 16
2.11 Una capacidad simple. 17
2.12 Capacidad dual usada para medir el desplazamiento de la masa del acelerómetro. 17
2.13 Diagrama de bloques del acelerómetro ADXL150 de Analog devices. . . . 18

3.1 a) Esquema de la fijación de los acelerómetros. b) Ilustración del PCB de acondicionamiento. 22
3.2 Esquema del acondicionamiento de la señal del acelerómetro. 23
3.3 Esquema del acondicionamiento de la señal de las células de pesado. . . . 23
3.4 Conexionado del sistema. 26
3.5 Diagrama de bloques de la DAQCard-AI-16XE-50. 26
3.6 Tipos de conexiones analógicas con entrada flotante. 27
3.7 Adquisición con doble-buffer y transferencias de datos secuenciales. . . . 33
3.8 Adquisición con doble-buffer: Sobreescritura antes de la copia. 34
3.9 Adquisición con doble-buffer: Sobreescritura. 34

3.10 Formato de la interfaz con el usuario. 36
3.11 Elementos de los diagramas del modelo de objetos. 38
3.12 Inicialización y finalización de la adquisición. 40
3.13 Adquisición doble-buffer y refresco de pantalla. 40
3.14 Interfaz de usuario del sistema de adquisición. 41
3.15 Relación entre la barra de tabs y los canales seleccionados. 44

4.1 Esquema del sistema dinámico de pesado. 46
4.2 Señal ideal: Tramos *peso* y *no-peso* de tazas vacías y no vacías con diferentes pesos. 47
4.3 Señales adquiridas para velocidades de a) 6 frutos/segundo y b) 20 frutos/segundo. 48
4.4 Célula testigo a 0 (azul), 6 (rojo) y 20 frutos/segundo (verde). 50
4.5 Tramo de 0.5 segundos del registro tomado por la célula testigo a 6 frutos/segundo. 50
4.6 Transformada de Fourier del registro obtenido a máxima velocidad. 51
4.7 Solapamiento de a) 16 tramos correspondientes a experimentos consecutivos con taza llena con 149gr a 16 frutas/segundo y las tazas siguientes, b) 8 tramos correspondientes a distintos experimentos con taza llena con 182gr a 20 frutas/segundo. 53
4.8 Distintos casos, a 6 frutos/segundo, de oscilaciones de las zonas peso. . . 54
4.9 Segmento correspondiente a 50 ms, 10 ms después de que la última taza tras la referencia deje la plataforma. 56
4.10 Respuesta en frecuencia de la función de transferencia del modelo. 58
4.11 Diagrama de polos y ceros del modelo. 59
4.12 Salida del sistema modelado ante una entrada de ruido gaussiano. Proceso aleatorio de banda estrecha. 60
4.13 Ilustración del marcado y troceado. 61
4.14 Varios tramos correspondientes a distintos casos del mismo experimento. 62
4.15 Respuesta en frecuencia del promediado de orden 20. 63
4.16 Retardo de grupo de la función de transferencia inversa del modelo. . . . 65
4.17 Preprocesado completo mediante deconvolución del modelo ARMA y filtro promediador. 65
4.18 Esquema de un cancelador de ruido adaptativo. 66

5.1 Esquema de una máquina calibradora. 74
5.2 Organización del sistema desde el punto de vista del software y las comunicaciones. 77



5.3 Formato del interfaz para el mantenimiento. 78
5.4 Formato del interfaz para el mantenimiento de la tarjeta de pesado. . . . 79
5.5 Formato del interfaz para la tarjeta de electroimanes. 79
5.6 Formato del interfaz para el control de los variadores. 80
5.7 Formato del interfaz para la gestión de partidas y la representación gráfica de estadísticas. 80
5.8 Formato del interfaz para la gestión de programas. 81
5.9 Formato del interfaz para los mapas de color. 81
5.10 Formato del interfaz para los grupos de color. 82
5.11 Formato del interfaz para los grupos de peso, el formato para los grupos de tamaño es el mismo. 82
5.12 Formato del interfaz para la asignación de salidas. 83
5.13 Fotografía del módulo de pesado. 84
5.14 Fotografía del módulo de encoder. 85
5.15 Conexión del encoder. 85
5.16 Fotografía del módulo de electroimanes. 86

6.1 Fotografía del módulo de pesado. 87
6.2 Fotografía de la placa de comunicaciones. 89
6.3 Diagrama simplificado de bloques del TMS320C26. 94
6.4 Diagrama de bloques del TMS320C26. 96
6.5 Arquitectura de la XC95108. 97
6.6 Mapa de memoria interna después de la configuración CONF1. 100
6.7 Interfaz del DSP con a) la memoria EPROM, b) la memoria NVRAM. . . 101
6.8 Mecanismo de generación de un único estado de espera. 102
6.9 Mapa de memoria. 105
6.10 Diagrama funcional de bloques del AD7730. 105
6.11 Conexión simplificada para un puente de excitación en DC. 106
6.12 Escritura en el AD7730. 110
6.13 Lectura del AD7730. 111
6.14 Esquema del mecanismo de buffer intermedio implementado por la CPLD. 114
6.15 Comunicación RS2323 con el DSP. 117
6.16 Detalle de la comunicación RS2323. 117
6.17 Detalle del 'C26 y la CPLD. 118
6.18 Detalle de la zona de los conversores AD7730. 118

7.1 La aplicación hace uso del SO para acceder a los recursos del sistema. . . 120
7.2 a) Fases de ejecución de varios programas. b) Ejecución serie de estos. . . 121

7.3 a) Ejecución concurrente. b) Diagrama *prioridadtiempo proceso* 122
7.4 Buffers circulares para el manejo de muestras o resultados de filtrado. . . 129
7.5 Representación lineal de los buffers circulares de muestras y resultados intermedios. 129
7.6 Representación lineal de los buffers circulares de sincronismos y tiempos. . 130
7.7 Esquemático de la cola de recepción, molde de recepción y variables asociadas. 131
7.8 Esquemático de la cola de transmisión, molde de transmisión y variables asociadas. 132
7.9 Esquemático de las distintas colas de mensajes priorizadas. 133
7.10 Ejecución basada en prioridades. 146
7.11 Ejecución basada en prioridades con derecho preferente. 149
7.12 Forma general del diagrama de transición de estados. 151
7.13 Diagrama de transición de estados en EMMOS. 154
7.14 Flujo de desarrollo de la aplicación. 156
7.15 Entorno de depuración de la aplicación. 159
7.16 Esquema de la jerarquía de los diferentes niveles de prioridad. 163
7.17 Tiempo de respuesta. 172
7.18 a) Planificación SRTN. b) Diagrama *prioridad versus tiempo de proceso.* . 175
7.19 a) Planificación Round Robin para T=0.125 unidades. b) Diagrama *prioridad versus tiempo de proceso.* 176
7.20 Traza de la ejecución del ejemplo de planificación. 180
7.21 Traza de la recepción de un mensaje de 6 bytes. 181
7.22 Diagrama de flujo del servicio del *timer.* 185

8.1 Esquema del paso de una taza sobre la célula de carga. 189
8.2 Árbol de modos de funcionamiento. 191
8.3 Topología de los 16 bits de cada tara. 195
8.4 Campos delmensaje OSCILOSCOPIO. 206
8.5 Momento en el que la taza empieza a salir sobre la célula de carga. . . . 209

9.1 Comparación del Fletcher-Reeves (negro), la variante del momento (rojo), y la deconvolución con el modelo ARMA (azul). 219
9.2 Taza llena con 50gr. Comparativa de las oscilaciones en la zona peso de los algoritmos de preproceso comparados (ARMA, en azul; promediado de orden 20, en rojo; ALMS en verde; MLMS en negro). 220

9.3 Taza llena con 200gr. Comparativa de las oscilaciones en la zona peso de los algoritmos de preproceso comparados (ARMA, en azul; promediado de orden 20, en rojo; ALMS en verde; MLMS en negro). 221
9.4 Diagrama de magnitudes de oscilación de pico a pico en la zona peso relativas a la máxima oscilación de pico a pico. 222
9.5 Dispersión promedio para cada algoritmo respecto del patrón promedio. . 222
9.6 Comportamiento de los patrones promedio antes de iniciar la transición. . 224
9.7 Comportamiento de los patrones promedio durante y tras la transición. . 224
9.8 Aplicación del algoritmo a un conjunto de casos del mismo experimento con una pesa de 197gr a 15 frutas/segundo. 227
9.9 Esquema del filtrado adaptativo. 228
9.10 Vibración del vástago de la célula de carga. 229
9.11 Vibración del cuerpo de la célula de carga. 229
9.12 Señal de error procedente del filtro adaptativo: Sincronismos. 230
9.13 a) Intervalo transformado encajado sobre el intervalo de precisión b) Encajado ideal. 238
9.14 Histograma 250 pruebas a 15 frutas/segundo con la pesa de 182gr. . . . 239

A.1 Formato de los mensajes CAN 249
A.2 Proceso de sincronización de las tarjetas. 253
A.3 Estructura de un mensaje CAN. 254
A.4 Formato del identificador del mensaje CAN. 255

B.1 Diagrama de bloques del puerto serie. 259
B.2 Cronograma de transmisión del puerto serie del 'C26. 260
B.3 Cronograma de recepción del puerto serie del 'C26. 260

C.1 Cronograma de escritura de datos al AD7730. 264
C.2 Cronograma de lectura de datos del AD7730. 265

D.1 Representación gráfica de las especificaciones de la directiva MEMORY. . 269
D.2 Ordenación de las secciones durante la construcción. 270

F.1 Comunicación serie. 282
F.2 Proceso de carga del programa a partir de la EPROM. 283
F.3 Inicialización modelo ROM. 288

G.1 Fichero de registros auxiliares. 294
G.2 Unidad Central Aritmético-Lógica (CALU). 295
G.3 Registro máscara de interupciones. 296

G.4 Cambios en pila para la generación de un marco de función. 300

H.1 Jerarquia de prioridades software. 305

K.1 Asignación de pines del conector 68-AI. 321

Índice de Tablas

2.1 Características de galgas extensométricas metálicas y semiconductoras. 12

3.1 Descripción de las configuraciones de entrada. 27
3.2 Precisiones de la medida con diferentes rangos y ganancias. 28

4.1 Resultados de estimación de diferentes criterios. 58

6.1 Conexión del módulo al bus CAN 89
6.2 Mensajes del display en cada modo de funcionamiento. 90
6.3 Información sobre las comunicaciones. 91
6.4 Tests realizados. 93

7.1 Estructura de los mensajes CAN recibidos por el DSP. 136
7.2 Estructura de los mensajes internos recibidos por el DSP. 136
7.3 Estructura de los mensajes CAN transmitidos por el DSP. 137
7.4 Estructura de los mensajes internos transmitidos por el DSP. 138

9.1 Resultados experimentales obtenidos para 20 frutas/segundo. 233
9.2 Resultados experimentales obtenidos para 15 frutas/segundo. 234
9.3 Resultados experimentales obtenidos para 12 frutas/segundo. 234
9.4 Resultados experimentales obtenidos para 6 frutas/segundo. 234
9.5 Transformación a gramos, de orden 2, de los intervalos [$\bar{x}-\sigma$,$\bar{x}-\sigma$]. 237
9.6 Intervalos [$\bar{x}-\sigma$,$\bar{x}-\sigma$] relativos al valor del peso. 237
9.7 Intervalos relativos al valor del peso para transformación de orden 3. 238

1.1 Velocidad de transmisión del bus CAN. 251

B.1 Tabla de bits del puerto serie, pines y registros del 'C26. 257

G.1 Tabla de registros mapeados en memoria. 293

G.2 Vectores de interrupción y prioridades.................................... 297

Capítulo 1

Introducción

En 1997 se planteó un ambicioso proyecto en un sector tan importante para la Comunidad Valenciana como es la maquinaria industrial hortofrutícola. El objetivo era la realización de un sistema de calibración de fruta, de precisión y alta velocidad. Estos dispositivos poseen por una parte un dispositivo mecánico de transporte de los productos y por otra una electrónica que gobierna la extracción de las características de los productos, la clasificación y el control del proceso.

En el mercado de productos perecederos, la velocidad de proceso es un factor muy importante para la consecución de un producto competitivo, dado que un retraso en el procesado de los productos supone una devaluación del valor del mismo. Así, se parte de la necesidad de obtener un sistema electrónico de proceso que iguale la velocidad sostenida de transporte del sistema mecánico.

La velocidad que puede alcanzar una mecánica avanzada en el mercado (denominada mecánica de cadena inteligente) es de hasta 15 frutas por segundo y línea. La electrónica propuesta en este proyecto, o más bien macroproyecto, parte de las siguientes especificaciones:

- Velocidad de hasta 20 frutos por segundo.
- Máximo de 10 líneas de transporte de fruta.
- Máximo de 64 salidas de frutas.
- Máximo de 1350 tazas por línea.

En conclusión, la electrónica debe poder procesar, tanto en pesado como en calibración de tamaño o color, a un ritmo de hasta 200 frutas por segundo (20 frutas/segundo con 10 líneas), pudiendo a la vez realizar el proceso de grandes cantidades de fruta, caracterizada individualmente, en programas de peso, color, etc.

A este macroproyecto se encuentran adscritos diferentes equipos de desarrollo de la empresa privada y la universidad. Actualmente, ya concluido, el sistema calibrador de alta velocidad es fabricado por la empresa MAXFRUT S.L. de Alzira (Valencia), encargada del desarrollo de la parte mecánica. La electrónica ha sido desarrollada conjuntamente la empresa de Algemesí DISMUNTEL S.A.L. y la Universidad de Valencia a través del proyecto "Contrato de asesoramiento y asistencia técnica en el área de control distribuido, comunicaciones CAN y desarrollo de estrategias de pesada", dirigido por Dr. D. J. Calpe Maravilla, y del proyecto cofinanciado con Fondos FEDER "Desarrollo de subsistemas inteligentes para el tratamiento de señales procedentes de sensores de peso. Aplicación al control distribuido de calidad en frutos", también dirigido por el Dr. D. J. Calpe. El desarrollo del sistema de análisis visual de la fruta y su entorno ha sido desarrollado por el Departamento de Informática de la Universidad Jaume I de Castellón.

El propósito del proyecto consiste en superar las tecnologías actuales en el mercado de calibradores de alta velocidad, mayoritariamente de origen extranjero, por una arquitectura que presente, además de mayores prestaciones, mejores características de flexibilidad y fácil escalabilidad, frente a la rigidez de los calibradores actuales. Al mismo tiempo, se busca facilitar la integración de estos sistemas en la gestión integral de los almacenes con accesos a herramientas externas de tratamiento de bases de datos y dejar abiertas posibilidades de accesos remotos al sistema para realizar funciones de mantenimiento.

Para un calibrador de estas características, nos hemos decantado por una electrónica de control distribuido en diferentes módulos inteligentes conectados por dos tipos de bus digitales de comunicación.

El trabajo presentado describe parte del macroproyecto. El autor ha sido el encargado de la realización a nivel de diseño, test y programación del módulo inteligente de pesado y del procesado digital que este realiza. Este módulo, que es descrito en el capítulo 6, consta de una conexión a bus CAN, está basado en un procesador digital de señal DSP TI-TMS320C26, posee una CPLD grabable en placa, un sistema autónomo de arranque y basado en host para depuración, 10 líneas de adquisición y un sistema operativo, microkernel, embebido de tiempo real diseñado también por el autor, el cual se adapta a las necesidades y características específicas del sistema.

El módulo se describe profundamente sin llegar proporcionar documentación suficiente para la reproducción del sistema, por el compromiso de privacidad que, obviamente, se debe a la empresa que comercializa el producto. Por ello no se ofrecerán, en general, piezas completas de código, esquemáticos o PCBs.

Este módulo utiliza eventos temporales externos, denominados *sincronismos*. En la calibradora comercial se utiliza un algoritmo de inteligente de posicionamiento

de sincronismo que proporciona una precisión de ±1gr a una velocidad de 15 frutas/segundo. Nótese que aunque la electrónica está preparada para mayores velocidades, esta primera calibradora comercializada ofrece menores prestaciones en cuanto a velocidad por cuestiones que nacen de la mecánica de arrastre.

Dando un paso más allá, el autor introduce un estudio para realizar una mejora de los algoritmos de preprocesado y estimación del peso. Con ello, se ha obtenido una mejora sustancial en la tolerancia del peso, con tolerancias de ±1gr. para velocidades de hasta 20 frutas/segundo. Estos podrían incorporarse a la máquina en lo sucesivo. No obstante, se verá que este método necesita una calibración muy rigurosa, aspecto que, a menudo, el cliente final pretende evitar a costa de asumir una perdida en la precisión.

En lo sucesivo, los capítulos 2 y 3 describen los transductores usados: células de carga y sensores de aceleración, que forman parte de un sistema de adquisición, capítulo 3, realizado con el objeto de obtener los registros con los que realizar el estudio de la caracterización y los algoritmos, capítulo 4. Esta es la base del analisis realizado para obtener un incremento de la tolerancia de la estimación del peso. El capítulo 9 describe las partes del método propuesto y los resultados obtenidos relativos a este. El trabajo realizado viene respaldado por su publicación en importantes congresos de procesado digital como son el ICASSP (Frances, 2000) y el ICSPAT (Calpe, 2000).

Por otra parte, el módulo de pesado, se describe en los capítulos 5 a 8. El capítulo 5 describe el sistema calibrador de forma integral. El capítulo 6 versa sobre el hardware del módulo de pesado y el 7 sobre el microkernel de tiempo real creado por el autor para el procesador TMS320C26. El capítulo 8 describe la aplicación que corre sobre el microkernel.

Capítulo 2

Transductores

Este capítulo describe los fundamentos de los transductores utilizados. El transductor más importante es la célula de carga. Sin embargo, también han sido usados, y por ello también se describe a continuación, sensores de aceleración.

2.1 Células de carga

Las células de carga son utilizadas en la práctica totalidad de los sistemas de pesado. Una célula de carga es clasificada como un transductor de fuerza. Este dispositivo convierte fuerza peso en una señal eléctrica.

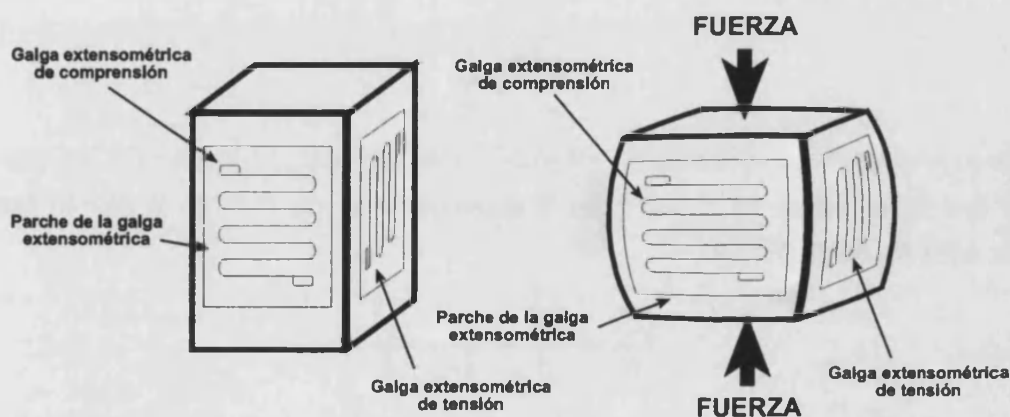


Figura 2.1: a) Elemento de esfuerzo sin tensión aplicada, y b) con tensión aplicada.

La galga extensométrica es el corazón de la célula de carga. Una galga extensométrica es un dispositivo cuya resistencia cambia cuando es sometida a tensiones.

Las galgas son realizadas a partir de una hoja ultrafina de metal tratada térmicamente y químicamente rodeada por una fina capa de dieléctrico. Después se montan en forma de parche en el elemento de esfuerzo, con adhesivos especialmente formulados, figura 2.1. El preciso posicionamiento de la galga, el procedimiento de montado, y los materiales usados, tienen, todos ellos, un efecto medible sobre el conjunto de prestaciones de la célula de carga.

Cada parche consiste en uno o más hilos finos, fijados a la superficie de una barra, anillo o columna (el elemento de esfuerzo) dentro de una célula de carga. Conforme se tensa la superficie a la que la galga está fijada, los hilos se estiran o comprimen cambiando su resistencia de forma proporcional a la carga aplicada. En la construcción de una célula de carga son usadas una o más galgas.

Se pueden conectar múltiples galgas extensométricas para crear los cuatro brazos de una configuración en puente de Wheatstone. Cuando se aplica una tensión de entrada al puente, la salida genera una tensión proporcional a la fuerza sobre la célula de carga. Esta salida puede ser amplificada y procesada por instrumentación convencional.

2.1.1 Galgas extensométricas

Fundamento: Efecto piezorresistivo

Las galgas extensométricas se basan en la variación de la resistencia de un conductor o un semiconductor cuando se les somete a un esfuerzo mecánico (Pallás, 1993; Pallás, 1994). Este efecto fue descubierto por Lord Kelvin en 1856. Si se considera un hilo metálico de longitud l, sección A y resistividad ρ, su resistencia eléctrica R es

$$R = \rho \frac{l}{A} \tag{2.1}$$

Si se le somete a un esfuerzo en dirección longitudinal, cada una de las tres magnitudes que intervienen en el valor de R experimentan un cambio y, por lo tanto, R también cambia de la forma

$$\frac{dR}{R} = \frac{d\rho}{\rho} + \frac{dl}{l} - \frac{dA}{A} \tag{2.2}$$

El cambio de longitud que resulta de aplicar una fuerza F a una pieza unidimensional, siempre y cuando no se entre en la zona de fluencia (figura 2.2), viene dado por la ley de Hooke,

$$\sigma = \frac{F}{A} = E\varepsilon = E\frac{dl}{l} \tag{2.3}$$

donde E es una constante del material, denominada módulo de Young, σ es la tensión mecánica y ε es la deformación unitaria. ε es adimensional, pero para mayor claridad se suele dar en "microdeformaciones" (1 microdeformación $= 1\mu\varepsilon = 10^{-6}m/m$).

Si se considera ahora una pieza que además de la longitud l tenga una dimensión transversal t, resulta que como consecuencia de aplicar un esfuerzo longitudinal, no sólo cambia l sino que también lo hace t. La relación entre ambos cambios viene dada por la ley de Poisson, de la forma

$$\mu = -\frac{dt/t}{dl/l} \tag{2.4}$$

donde μ es el denominado coeficiente de Poisson. Su valor está entre 0 y 0,5, siendo, por ejemplo, de 0,17 para la fundición maleable, de 0,303 para el acero y de 0,33 para el aluminio y el cobre. Obsérvese que para que se conserve constante el volumen debe cumplirse $\mu = 0{,}5$.

Figura 2.2: Relación entre esfuerzos y deformaciones (escala de la zona elástica muy ampliada).

Para el hilo conductor considerado anteriormente, si se supone una sección cilíndrica de diámetro D, se tendrá

$$A = \pi D^2/4 \tag{2.5}$$

$$dA/A = 2dD/D = -2\mu dl/l \tag{2.6}$$

La variación que experimenta la resistividad como resultado de un esfuerzo mecánico es lo que se conoce como *efecto piezorresistivo.* Estos cambios se deben a la variación de la amplitud de las oscilaciones de los nudos de la red cristalina del metal. Si éste

se tensa, la amplitud aumenta, mientras que si se comprime, la amplitud disminuye. Si la amplitud de las oscilaciones de los nudos aumenta, la velocidad de los electrones disminuye, y ρ aumenta. Si dicha amplitud disminuye, ρ también disminuye. Para el caso de los metales, resulta que los cambios porcentuales de resistividad y de volumen son proporcionales

$$\frac{d\rho}{\rho} = C\frac{dV}{V} \tag{2.7}$$

donde C es la denominada constante de Bridgman, cuyo valor es de 1,13 a 1,15 para las aleaciones empleadas comúnmente en galgas, y de 4,4 para el platino. Aplicando la expresión 2.6, el cambio de volumen se puede expresar como

$$V = \pi l D^2/4 \tag{2.8}$$

$$\frac{dV}{V} = \frac{dl}{l} + 2\frac{dD}{D} = \frac{dl}{l}(1-2\mu) \tag{2.9}$$

y, por lo tanto, si el material es isótropo y no se rebasa su límite elástico, la expresión 2.2 se transforma finalmente en

$$\frac{dR}{R} = \frac{dl}{l}\left[1 + 2\mu + C(1-2\mu)\right] = K\frac{dl}{l} \tag{2.10}$$

donde K es el denominado factor de sensibilidad de la galga, definido directamente como el factor dentro del corchete en la expresión anterior. A partir de los valores dados se ve que K es del orden de 2, salvo para el platino en cuyo caso es del orden de 6.

Así pues, para pequeñas variaciones la resistencia del hilo metálico deformado puede ponerse de la forma

$$R = R_0\,(1+x) \tag{2.11}$$

donde R_0 es la resistencia en reposo y $x = K\varepsilon$. El cambio de resistencia no excede del 2%.

En el caso de un semiconductor, al someterlo a un esfuerzo predomina el efecto piezorresistivo. Las expresiones de la relación resistencia/deformación son para un caso concreto (Pallás, 1994):

- Para un material tipo p

$$\frac{dR}{R_0} = 119,5\varepsilon + 4\varepsilon^2 \tag{2.12}$$

- Para un material tipo n

$$\frac{dR}{R_0} = -110\varepsilon + 10\varepsilon^2 \tag{2.13}$$

donde R_0 es la resistencia en reposo a 25°C, y se supone una alimentación a corriente constante.

Vemos, pues, que existe una relación entre el cambio de resistencia de un material y la deformación que experimente éste. Si se conoce la relación entre esta deformación y el esfuerzo que la provoca (Pallás, 1994), a partir de la medida de los cambios de resistencia se podrán conocer los esfuerzos aplicados y, en su caso, las magnitudes que provocan dichos esfuerzos en un sensor apropiado. Una resistencia dispuesta de forma que sea sensible a la deformación constituye una galga extensométrica.

Las limitaciones que cabe considerar en la aplicación de este principio de medida son numerosas y conviene conocerlas con detalle, pues de lo contrario es difícil obtener información útil con este método.

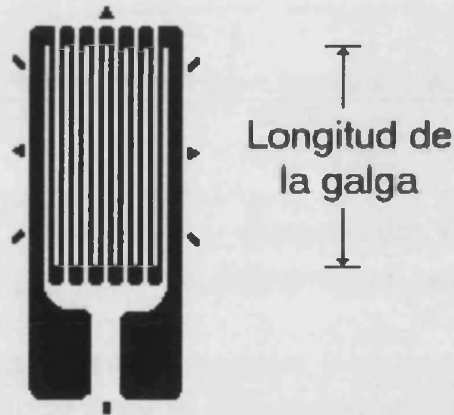


Figura 2.3: Galga impresa. Los hilos más finos son la parte activa de galga.

En primer lugar, el esfuerzo aplicado no debe llevar a la galga fuera del margen elástico de deformaciones. Éste no excede del 4% de la longitud de la galga y va desde unas 3000 $\mu\varepsilon$ para las semiconductoras a unas 40.000 $\mu\varepsilon$ para las metálicas.

En segundo lugar, la medida de un esfuerzo sólo será correcta si es transmitido totalmente a la galga. Ello se logra pegando ésta cuidadosamente mediante un adhesivo elástico que sea suficientemente estable con el tiempo y la temperatura. A la vez, la galga debe estar aislada eléctricamente del objeto donde se mide y protegida del ambiente.

Se supone también que se está en un estado plano de deformaciones, es decir, que no hay esfuerzos en la dirección perpendicular a la superficie de la galga. Para que la resistencia eléctrica de ésta sea apreciable se disponen varios tramos longitudinales y en el diseño se procura que los tramos transversales tengan mayor sección (figura 2.3), pues así se reduce la sensibilidad transversal a un valor de sólo el 1 o el 2% de la longitudinal. En la figura 2.4 se muestra la forma convencional de montar una galga.

La temperatura es una fuente de interferencias por varias razones. Afecta a la resistividad del material, a sus dimensiones y a las dimensiones del soporte. Como resultado de todo ello, una vez la galga está dispuesta en la superficie de medida, si hay un cambio de temperatura, antes de aplicar ningún esfuerzo se tendrá ya un cambio de resistencia. En galgas metálicas este cambio puede ser de hasta 50 $\mu\varepsilon/^{o}C$.

Esta interferencia se compensa con el método de la entrada "opuesta". Consiste en este caso en el empleo de las denominadas galgas "pasivas", que son galgas iguales a la de medida dispuestas junto a ésta, de forma que experimentan el mismo cambio de temperatura, pero que no están sometidas a esfuerzos mecánicos.

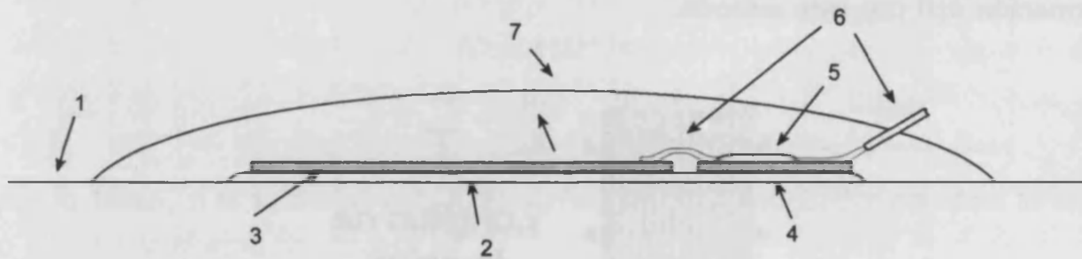


Figura 2.4: Montaje de una galga impresa. 1 Sustrato donde se monta; 2 adhesivo; 3 galga; 4 terminales para soldar; 5 soldadura; 5 hilos de conexión; 7 aislamiento protector.

Un factor que puede provocar el calentamiento de la galga es la propia potencia que disipe cuando, al medir su resistencia, se haga circular por ella una corriente eléctrica.

Idealmente, las galgas deberían ser puntuales para poder medir los esfuerzos en un punto concreto. En la práctica sus dimensiones son apreciables, y se supone que el "punto" de medida es el centro geométrico de la galga. Si se van a medir vibraciones, la longitud de onda de éstas debe ser mucho mayor que la longitud de la galga. Si, por ejemplo, ésta es de 5 mm y se mide en acero, donde la velocidad del sonido es de unos 5900 m/s, la máxima frecuencia medible es del orden de 100 kHz (1 MHz/10) que es ciertamente muy alta.

Sin embargo, pese a todas estas posibles limitaciones, por su pequeño tamaño, gran linealidad y baja impedancia, las galgas extensométricas son uno de los sensores de mayor aplicación.

Tipos y aplicaciones

Los materiales empleados para la fabricación de galgas extensométricas son diversos conductores metálicos, como las aleaciones *constantan*, *advance*, *karma*, y también semiconductores como el silicio y el germanio. Las aleaciones metálicas escogi-

das tienen siempre un bajo coeficiente de temperatura. Las galgas pueden tener o no soporte propio, eligiéndose en su caso en función de la temperatura a la que se va a medir. Para aplicaciones de sensores táctiles en robots, se emplean también elastómeros conductores. Para la medida de grandes deformaciones en estructuras biológicas, se emplean galgas elásticas que consisten en un tubo elástico lleno de mercurio u otro líquido conductor (Group, 1999).

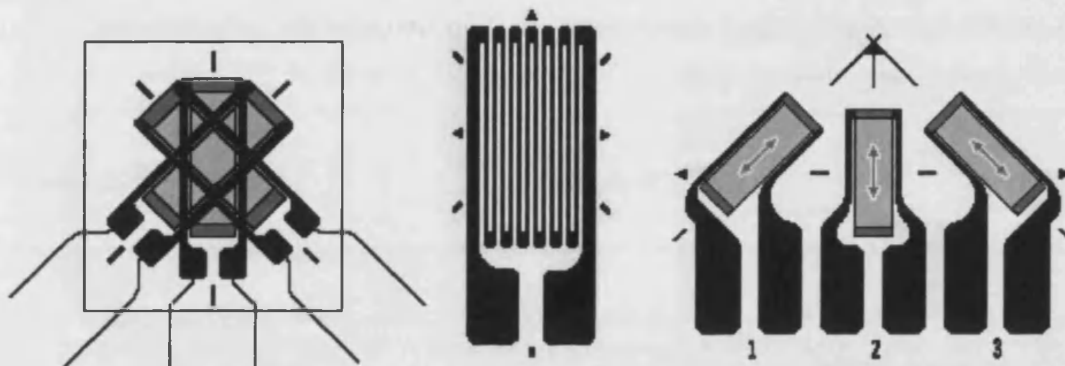


Figura 2.5: Diversos tipos de galgas metálicas y semiconductoras (Group, 1999).

En la figura 2.10 se muestran diversos tipos de galgas extensométricas. Las galgas metálicas con soporte pueden ser de hilo bobinado o plegado con soporte de papel, o impresas por fotograbado. En este caso pueden observarse algunas de las diferentes configuraciones posibles, adaptadas cada una de ellas a diversos tipos de esfuerzos. Hay modelos para diafragma, para medir torsiones, para determinar esfuerzos máximos y mínimos y sus direcciones (rosetas múltiples), etc.

En la tabla 2.1 se presentan algunas de las características habituales de las galgas metálicas y semiconductoras. El factor de sensibilidad se determina por muestreo, pues una vez utilizada la galga es irrecuperable.

Las galgas extensométricas se pueden aplicar a la medida de cualquier variable que pueda convertirse, con el sensor apropiado, en una fuerza capaz de provocar deformaciones del orden de 10 μm e incluso inferiores.

2.1.2 El puente de Wheatstone

La forma habitual de proporcionar la señal eléctrica de la salida de una célula de carga es una medida empleando un puente de Wheatsone mediante deflexión (Pallás, 1994). En éste, se mide la diferencia de tensión entre ambas ramas. Como ya se ha hecho referencia, una de las resistencias de los brazos del puente es proporcionada por la galga extensométrica. La única limitación es que la temperatura será una causa de interferencia. Esto se soluciona utilizando una compensación por el método de la

entrada “opuesta”. Este método consiste en la utilización como otras resistencias del puente, de una o varias galgas denominadas “galgas pasivas”, iguales a la utilizada para medida, y dispuestas junto a ésta de forma que experimenten el mismo cambio de temperatura. La única diferencia es que no se verán sometidas a esfuerzos mecánicos.

Cuando una tensión de entrada es aplicada al puente, la salida proporciona una tensión “proporcional” (realmente la salida se aparta un poco de la linealidad, a menos que se realice una linealización analógica) a la fuerza sobre la célula de carga. Esta salida puede ser amplificada y procesada por instrumentación convencional.

Parámetro	*Metálicas*	*Semiconductoras*
Margen de medida, $\mu\varepsilon$	0,1 a 40.000	0,001 a 3000
Factor de sensibilidad	1,8 a 2,35	50 a 200
Resistencia, Ω	120,350,600...5000	1000 a 5000
Tolerancia en la resistencia,%	0,1 a 0,2	1 a 2
Tamaño, *mm*	0,4 a 150 estándar: 3 a 6	1 a 5

Tabla 2.1 Características típicas de las galgas extensométricas metálicas y semiconductoras.

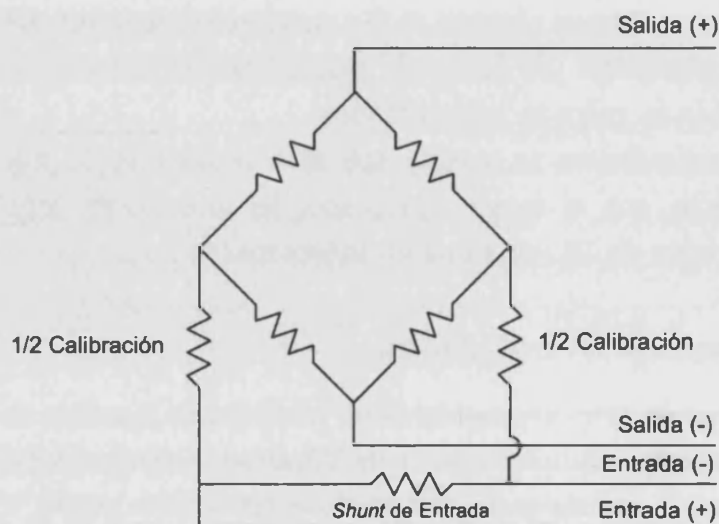


Figura 2.6: Puente de Wheatstone.

2.2 Sensores de aceleración

La aceleración es una magnitud que es manejada en numerosas aplicaciones, como sistemas inerciales, sensado de golpes, monitorización de la salud de la maquinaria móvil, niveles de vibración, etc. Para ello, y dependiendo de los requerimientos, se dispone de una amplia gama de transductores. Algunas de las tecnologías usadas en el sensado de la aceleración son:

- Piezo-film.
- Servo electromecánico.
- Piezoeléctrico.
- Sensores líquidos de inclinación.
- Micromáquina piezo-resistiva.
- Micromáquina capacitiva.
- Micromáquina capacitiva de superficie.

Todas éstas tienen diferentes prestaciones en cuanto a respuesta en frecuencia, rango dinámico, ruido, precisión, coste, fragilidad y pérdidas. Sin embargo, actualmente la mejor relación calidad/precio la proporcionan los sensores del tipo micromáquina capacitiva de superficie, que se describirán posteriormente, y que por ello ha sido la tecnología seleccionada. La figura 2.7 proporciona una comparativa de las diferentes tecnologías de sensado en términos de prestaciones/coste (Doscher, 1998).

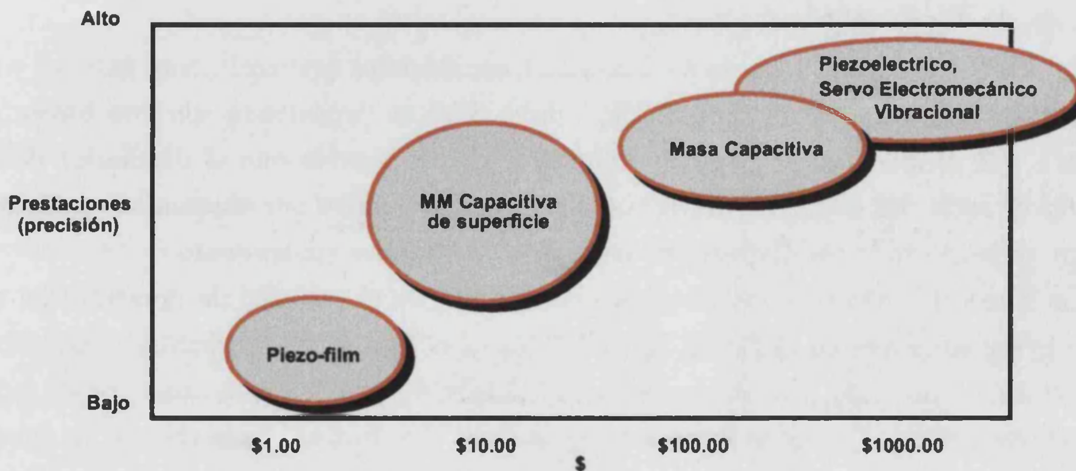


Figura 2.7: Comparativa de varias tecnologías.

2.2.1 Introducción a las micromáquinas de superficie

Los actuales avances en materia de miniaturización están llegando a todos los ámbitos, incluso la electrónica y las tecnologías de sensado. En estos campos, se manufacturan actualmente micromáquinas en el interior de chips, denominadas "micromáquinas superficiales de silicio".

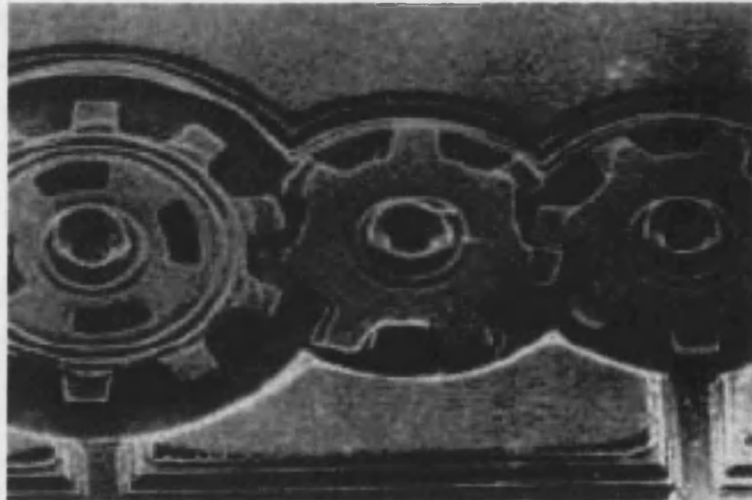


Figura 2.8: Tren de ruedas dentadas de 135μm de largo.

La confección de las micromáquinas superficiales de silicio aprovecha los procesos y equipamiento de la industria electrónica de semiconductores. Estas técnicas han evolucionado muy rápidamente y hoy muchas compañías ofrecen la integración de micromáquinas superficiales de silicio y electrónica CMOS en el mismo chip. La figura 2.8, muestra, a modo de ejemplo, unos engranajes que conjuntamente miden 135μm de largo, lo cual ilustra el alcance de las posibilidades de miniaturización y creación de micromáquinas.

Esta técnica deposita capas de materiales sacrificiales (generalmente óxidos) y estructurales sobre un sustrato de Silicio. Cada capa es depositada con una forma, de manera que el material se deposite sólo en aquellos lugares que el diseñador desea. Posteriormente, las capas de materiales sacrificiales pueden ser eliminadas completamente, dejando todo un dispositivo mecánico totalmente ensamblado.

La figura 2.9 muestra los pasos seguidos durante el proceso de creación del engranaje representado en la figura 2.8. El óxido es el material sacrificial y el polisilicio el material estructural. Como puede verse, inicialmente se deposita una capa de óxido termal (en gris claro), que es la base del polisilicio que formará cada una de las ruedas dentadas, y sobre la cual también se deposita una capa de óxido CDV. En el paso siguiente, podemos ver cómo se crea, también en polisilicio, el eje del engranaje. Fi-

nalmente se eliminan los óxidos quedando el engranaje libre para moverse alrededor del eje.

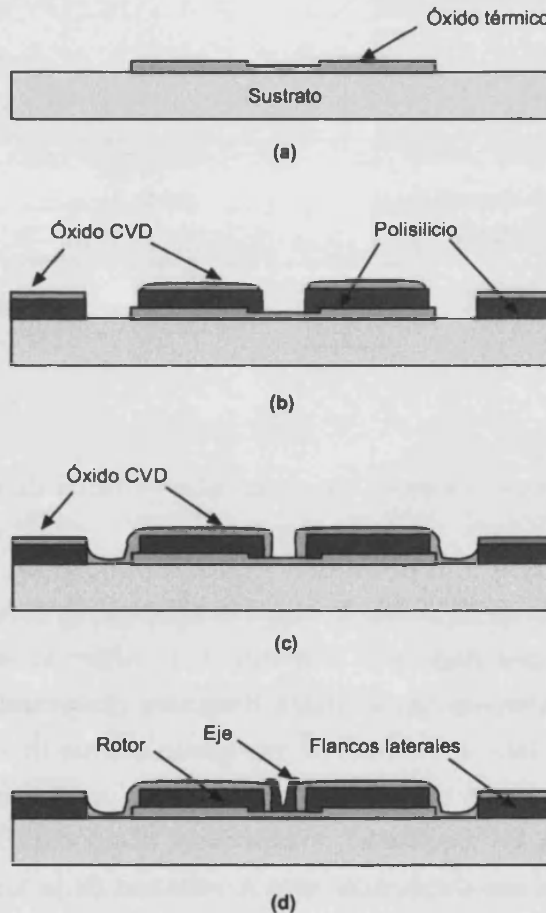


Figura 2.9: Pasos para la formación del engranaje de la figura 2.8.

Mediante estas técnicas se fabrican muchas otras máquinas como motores electrostáticos, etc. Una importante aplicación de éstas son los sensores de aceleración, usados, por ejemplo, para disparar los *air-bag* de los automóviles.

2.2.2 Principios del acelerómetro

El principio básico del acelerómetro es un sistema de masas móviles (Doscher, 1998). Los movimientos (dentro de su región lineal) están gobernados por la ley de Hook. Según ésta, el movimiento de la masa inercial establecerá una fuerza de recuperación proporcional a la cantidad de su estiramiento o compresión. Más concretamente $F = Kx$, donde K es la constante de proporcionalidad entre el desplazamiento x y la fuerza F. El otro principio físico importante es la segunda ley de Newton del

movimiento, la cual establece que una fuerza actuando sobre una masa exhibe un movimiento de aceleración dado por $F = ma$.

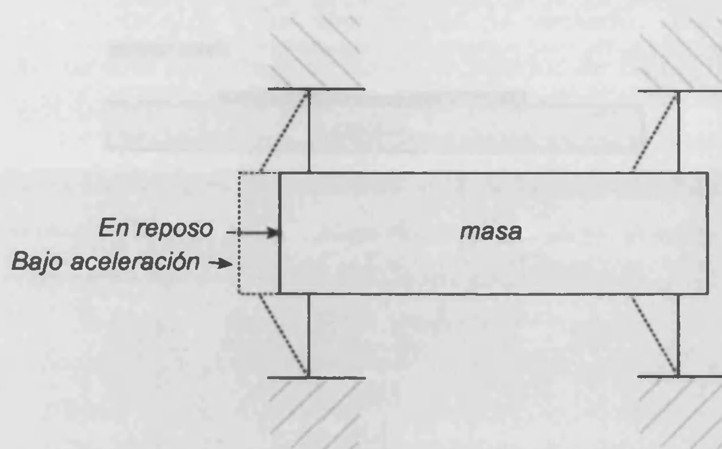


Figura 2.10: Sistema de masa móvil usado en el interior de un sensor de aceleración.

Esta fuerza causará que una masa móvil (como podría ser la de la figura 2.10) se mueva bajo la restricción de $F = Kx = ma$. De aquí que la aceleración a provoque un desplazamiento de la masa dado por $x = ma/K$ o, alternativamente, si observamos un desplazamiento x, sabemos que la masa lleva una aceleración $a = Kx/m$.

De esta manera, podemos traducir el problema de medir la aceleración en la de medir distancias de la masa anclada como muestra la figura 2.10 (suponiendo un comportamiento bajo la ley de Hook), y viceversa. Esto constituirá un acelerómetro de eje simple, y necesita ser duplicado para el sensado de la aceleración en otros ejes requeridos.

Un acelerómetro, como el de la figura 2.10, consiste en una barra de Silicio, que forma una masa móvil, unida mediante unos anclajes a la base de Silicio, de forma que puede moverse libremente dentro de un grado de libertad, respondiendo a la aceleración que ocurre en la línea de la masa. Esta masa se asociará a un sistema para medir su desplazamiento y el circuito adecuado de acondicionamiento de señal, como se verá posteriormente.

Cuando una aceleración acontece, la masa se mueve respecto a los anclajes que la soportan. Así, la cantidad de aceleración será proporcional a la cantidad de desplazamiento de la masa. Esto no es verdad en caso de que el sistema no sea una "masa móvil ideal", de la cual se habló en el punto anterior. De hecho, el sistema es compensado por algún, más o menos complicado, circuito de acondicionamiento de señal, que puede estar incluso en el interior del mismo chip que la micromáquina.

El siguiente problema que necesita ser solucionado es la medida del desplazamiento de la barra. Éste se basa en la capacitancia. Un condensador simple está formado

por dos placas de metal, cada una en paralelo con la otra. La capacitancia de este dispositivo viene dada por $C = K/x_0$, donde K es la propiedad del material entre las placas.

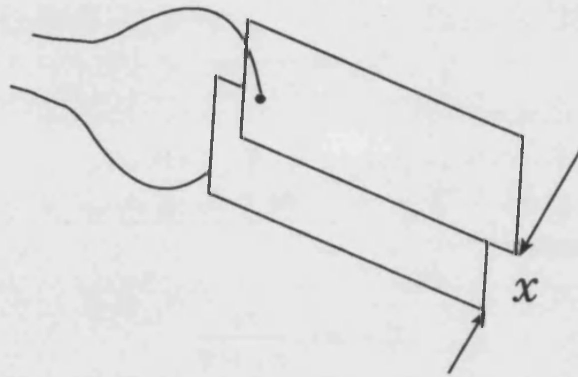


Figura 2.11: Una capacidad simple.

Usando esta expresión, y asumiendo K conocido, se puede medir x a partir de la capacitancia, figura 2.11, es decir, el espacio entre las placas.

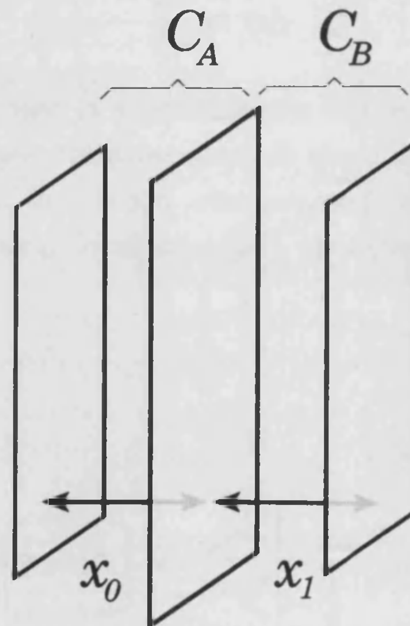


Figura 2.12: Capacidad dual usada para medir el desplazamiento de la masa del acelerómetro.

Los acelerómetros empleados en esta tesis, llevan esta técnica un paso más adelante al utilizar dos capacidades configuradas como se indica en la figura 2.12. Si el

dispositivo está en reposo, y el espacio entre cada una de las placas es x_0, entonces cada una de las capacidades exhibe una capacitancia de $C = K/x_0$. Si la placa central se mueve una distancia x, esto redunda en que los resultados para las capacidades cambian a:

$$C_A = \frac{k}{x_0 + x} \tag{2.14}$$

$$C_B = \frac{k}{x_0 - x} \tag{2.15}$$

que puede ser escrito como:

$$C_A = C\frac{x_0}{x_0 + x} \tag{2.16}$$

$$C_B = C\frac{x_0}{x_0 - x} \tag{2.17}$$

Para valores pequeños de la capacitancia es proporcional a x, y entonces la expresión anterior se reduce a:

$$\Delta C \approx \frac{-2}{x_0^2} \tag{2.18}$$

La diferencia en capacitancia es proporcional a x, pero sólo para pequeños valores de desplazamiento. Se usa un bucle de realimentación negativa para asegurar que el movimiento de la masa se mantiene pequeño, por el cual la expresión 2.18 permanece correcta. La figura 2.13 muestra un diagrama de bloques del acelerómetro (ADXL, 1998).

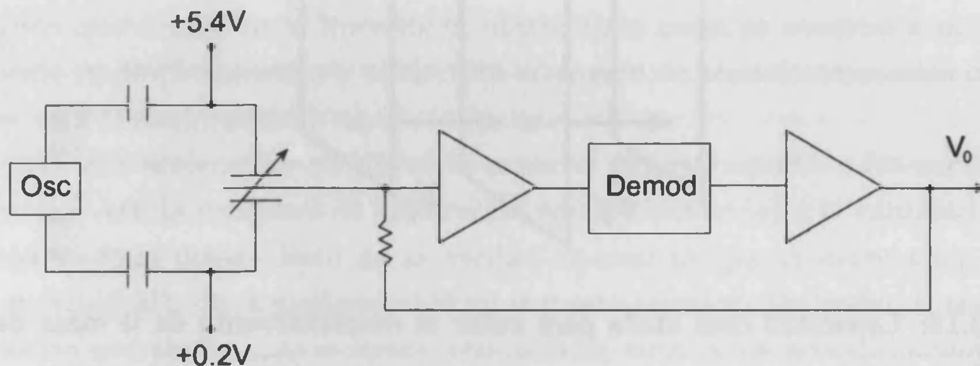
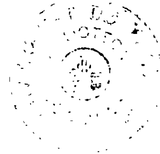


Figura 2.13: Diagrama de bloques del acelerómetro ADXL150 de Analog devices.



2.2.3 Usos actuales de los sensores de aceleración

Actualmente, los sensores de aceleración (o acelerómetros) se utilizan para realizar medidas inerciales de posicionamiento y velocidad, donde la velocidad es la integración simple de la aceleración y el posicionamiento, la segunda integral de ésta. Estos sensores son un método *sin contacto*, para medir posición, velocidad y aceleración. Otra aplicación ampliamente extendida en la industria automovilística es la detección de golpes. Así, puede estudiarse la firma de un golpe para saber su intensidad y circunstancias para distinguir un accidente y su intensidad. Aplicaciones son: cinturones de seguridad activos y, en general, cualquier sistema de seguridad activo (Doscher, 1998).

Aplicaciones nacientes son la medida de las vibraciones para la detección de la salud de maquinaria industrial, y la detección de posicionamiento en el bi- y tridimensional.

Capítulo 3

El sistema de adquisición

Para el estudio de mejoras de la precisión en algoritmos de pesada a altas velocidad, se utilizan señales reales obtenidas mediante un sistema de adquisición multicanal de realización propia, cuya descripción es el tema de este capítulo.

Para estudiar el comportamiento a altas velocidad, debe tenerse en cuenta la vibración, por lo que se usarán dos acelerómetros para recoger las vibraciones verticales del vástago y cuerpo de la célula de carga de línea. Además de estas señales, el sistema ha de adquirir la señal de salida de la célula de carga de línea y otra célula de carga fijada al chasis de la calibradora, denominada célula de carga testigo, utilizada como planteamiento alternativo para el registro de las vibraciones.

De esta forma, se plantea el estudio de la influencia de las vibraciones que se producen a altas velocidades de funcionamiento de la cinta de arrastre, véase sección 5.2. Los registros adquiridos y utilizados en los capítulos posteriores han sido obtenidos tomando estas cuatro señales a 1KHz y con 16 bits de resolución.

En este capítulo se tratará en profundidad la creación del sistema de adquisición desde la disposición de los transductores en la medida, módulos diseñados para el acondicionamiento, equipamiento utilizado, programación de la aplicación, hasta el funcionamiento de ésta.

3.1 Conexionado de los transductores

Las células de carga utilizadas son células de acero de 10Lbs de Artech Industries Inc. (Riverside, CA) con una deformación de 2.096mV/V @ 10Lbs. Ésta describe una vista de perfil con la fijación de los sensores de aceleración a dos partes de la célula de carga de línea. Como puede verse, para realizar la fijación de los sensores

de aceleración ADXL150 a las partes indicadas y recoger la vibración tanto del chasis de la calibradora, en el sensor fijado al cuerpo de la célula, como de la zona de pesado, en el sensor fijado al vástago, se ha diseñado un pequeño PCB que realiza un primer acondicionamiento de la señal y al que se encuentra soldado el acelerómetro, como ilustra la figura 3.1b. Dicho PCB posee dos orificios. En la figura 3.1b puede apreciarse como ambos orificios son usados por tornillos que son fijados fuertemente a la célula de carga mediante pegamento de contacto para superficies metálicas. De esta forma, el par de tornillos actúan como guías para la introducción del pequeño PCB, que posteriormente es fijado a la superficie por la presión que las tuercas ejercen. De esta forma se evita una fijación permanentemente del sensor a la célula de carga, permitiendo la movilidad del módulo con sólo desenroscar las tuercas que, por otra parte, realizan una fijación muy eficiente.

Nótese que el sensor de aceleración tiene un eje de aceleración y un sentido que deben ser colocados, figura 3.1a, ambos en dirección vertical con sentido descendente.

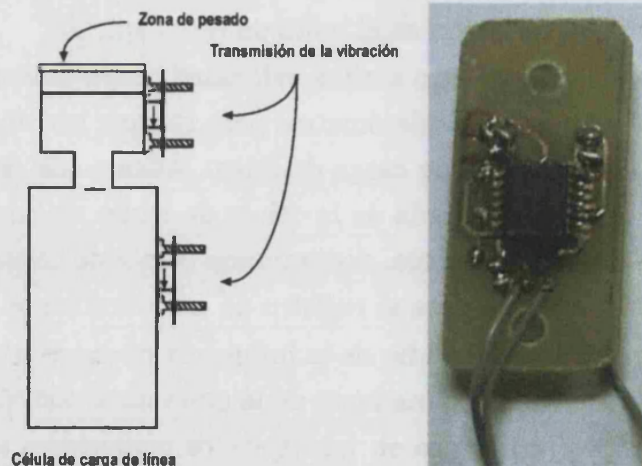


Figura 3.1: a) Esquema de la fijación de los acelerómetros. b) Ilustración del PCB de acondicionamiento.

El esquemático de la figura 3.2 corresponde al pequeño PCB del acelerómetro descrito. Como vemos, este circuito proporciona una baja impedancia de salida, un nivel de continua para que dicha señal pueda ser adquirida con rango unipolar y para ajustar mejor el rango de la señal al rango de entrada del conversor.

Las células de carga también poseen un acondicionamiento. Para ello se utiliza el integrado LTC1100 (LT, 1994), que es un amplificador de instrumentación de alta precisión que usa técnicas de estabilización que mejoran las prestaciones DC. Posee una baja corriente de bias de 50pA y una deriva del offset de continua típica de 10nV/°C. Obtiene una ganancia diferencial de 100 sin necesidad de resistencias ex-

ternas de ganancia. La ganancia de la linealidad es 8ppm y la deriva de la ganancia es 4ppm/ºC.

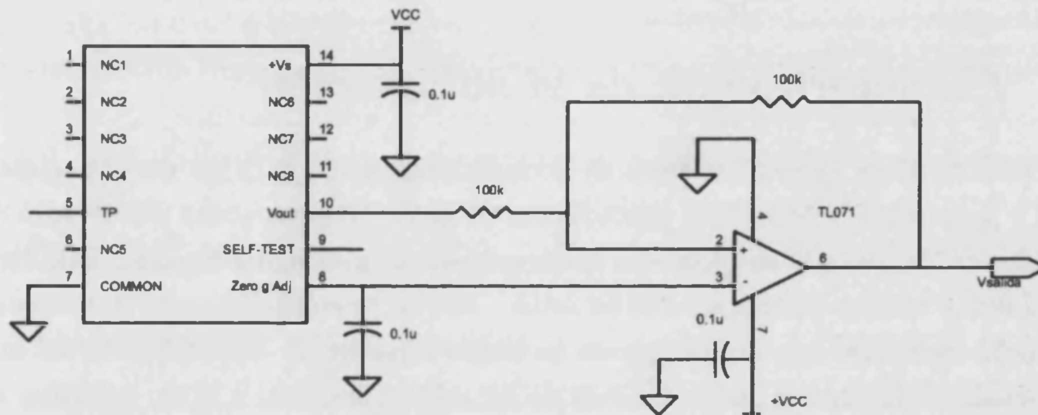


Figura 3.2: Esquema del acondicionamiento de la señal del acelerómetro.

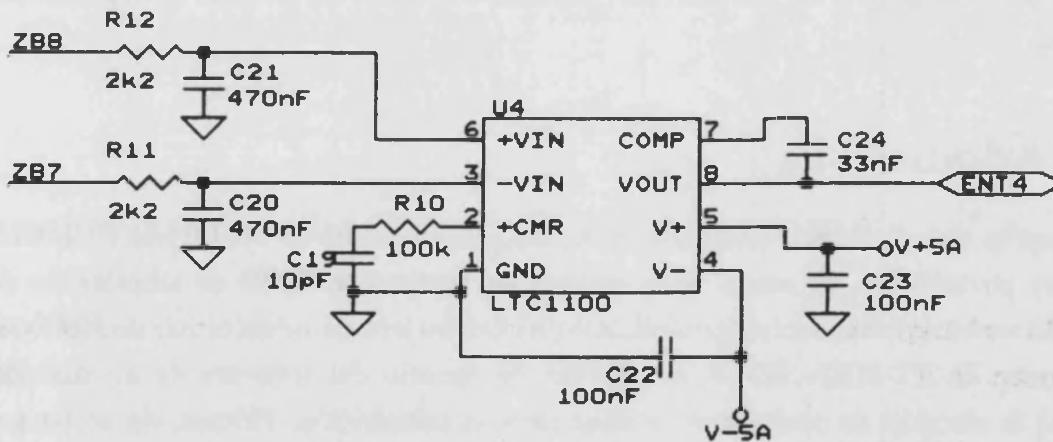


Figura 3.3: Esquema del acondicionamiento de la señal de las células de pesado.

La figura 3.3 muestra el esquemático del módulo de acondicionamiento. La alimentación del LTC1100 utiliza su propia alimentación para asegurar una mayor estabilización de las alimentaciones respecto a la que podría obtener del ordenador (Dedé, 1983). La salida de la adquisición, es conectada directamente a la caja de conectores de la tarjeta de adquisición, apéndice K .

3.2 Caracterización de la adquisición

El sistema debe poseer módulos de acondicionamiento para un par de células de carga, y para hasta seis señales procedentes de acelerómetros (caso del estudio de la vibración en los tres ejes en cada una de las partes que muestra la figura 3.1a), aunque normalmente sólo se utilizarán dos de éstos. Así, se prevé un sistema multicanal de hasta ocho entradas, que impondremos en modo diferencial. Dependiendo del origen de la señal, se dispondrá de un módulo de acondicionamiento u otro, descritos en la sección anterior.

Se realizará una adquisición con rango unipolar de la señal de entrada, frecuencia de muestreo f=1KHz, y una resolución de 16 bits. Para ello se utilizará una tarjeta de adquisición de National Instruments con equipamiento adicional descrito en la sección siguiente, necesario para conectar los módulos de acondicionamiento a dicha tarjeta que, conectada a un PC portátil, permitirá obtener un sistema de adquisición portable.

3.3 Adquisición

La tarjeta de adquisición utilizada es la DAQ-Card-AI-16XE-50, tarjeta PCMCIA ideal para portátiles y, en suma, para realizar un sistema portable de adquisición de datos. Sin embargo, dada la disponibilidad de otra tarjeta de adquisición de National Instruments, la AT-MIO-16E-10, el interfaz de usuario del software de adquisición permitirá la elección de cualquiera de ellas para la adquisición. Nótese, sin embargo, que las similitudes entre ambas son amplias, por lo que en lugar de describir las dos, se describirá únicamente la primera, para finalmente analizar las diferencias.

Sea cual sea la tarjeta elegida para la adquisición, el bloque de conectores de tornillo de los módulos de acondicionamiento y los cables utilizados son compatibles con ambas (NI-DAQCard, 1999; NI-AT, 1996; NI-Cat, 1999).

3.3.1 DAQCard-AI-16XE-50

Ésta es una tarjeta de adquisición de National Instruments que, al igual que todas

las pertenecientes a la serie E, es multifunción analógica, digital y de temporización de E/S para computadores equipados con slots PCMCIA tipo II. Posee un ADC de 16 bits con ocho líneas digitales de E/S y dos contadores/temporizadores de 24 bits.

Esta tarjeta usa el sistema controlador de temporización DAQ-STC para funciones temporales. El DAQ-STC consta de tres bloques que controlan entrada analógica, salida analógica y funciones de contador/temporizador de propósito general. Estos grupos incluyen un total de siete contadores de 24 bits y tres contadores de 16 bits y un máximo de resolución temporal de 50ns.

La AI-16XE-50 puede realizar interfaz a sistemas SCXI, pudiendo llegar a adquirir hasta 3000 señales analógicas de termopares, RTDs, galgas extensométrica, fuentes de tensión y/o fuentes de corriente. También pueden adquirirse o generarse señales digitales para comunicaciones y control (NI-DAQCard, 1999).

3.3.2 Equipamiento opcional

National Instruments ofrece una amplia variedad de productos para ser usados con las tarjetas DAQCard de la serie E, incluyendo cables apantallados y no apantallados, ensamblajes de cable, bloques de conectores, módulos SCXI y accesorios para aislamiento, amplificación, excitación y señales de multiplexado para relés y salidas analógicas.

El equipamiento del que se dispone para la conexión de las señales procedentes de los módulos de acondicionamiento a la tarjeta son:

- Obviamente la tarjeta DAQCard-AI-16XE-50.
- Cable PSHR68-68M. Un cable ribbon apantallado de 68 posiciones, con conectores macho a macho.
- Cable SH6868 apantallado para conexión al PSHR68-68M y al bloque de conectores.
- Bloque de conectores de 68 tornillos.

Con todo esto, figura 3.4, se realiza la conexión del bloque de conectores, al que van conectados los módulos de acondicionamiento de los transductores, al computador.

3.3.3 Descripción del hardware

La figura 3.5, muestra el diagrama de bloques de la tarjeta DAQCard-AI-16XE-50.

La sección de entrada analógica es configurable por software. Dispone de tres modos de entrada: entrada *single-ended* no referenciada (NRSE), entrada *single ended*

referenciada (RSE), y entrada diferencial (DIFF). Mediante configuraciones *single-ended* se dispone de hasta 16 canales analógicos de entrada. La configuración DIFF permite hasta 8 canales. Los modos de entrada son programados multimodo sobre una base por canal. Por ejemplo, se puede configurar el sistema para escanear 12 canales; cuatro configurados como diferenciales y 8 como *single-ended.* La tabla 3.1 describe las tres configuraciones de entrada.

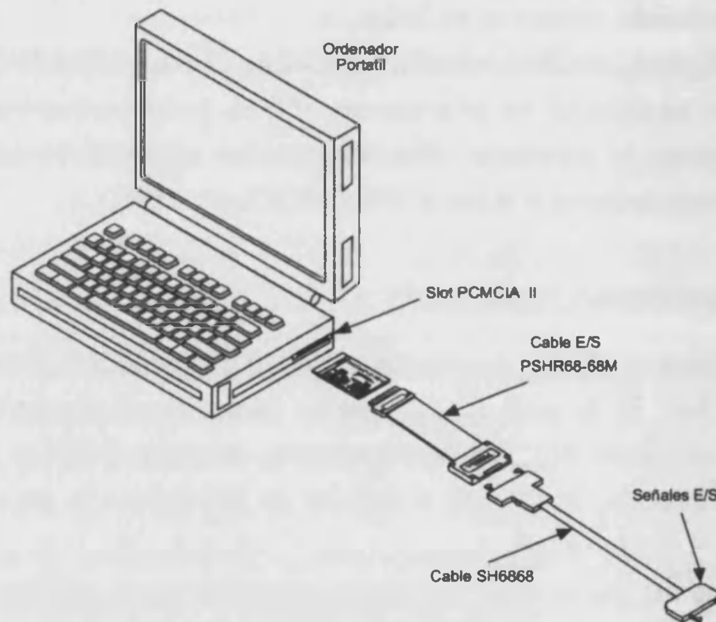


Figura 3.4: Conexionado del sistema.

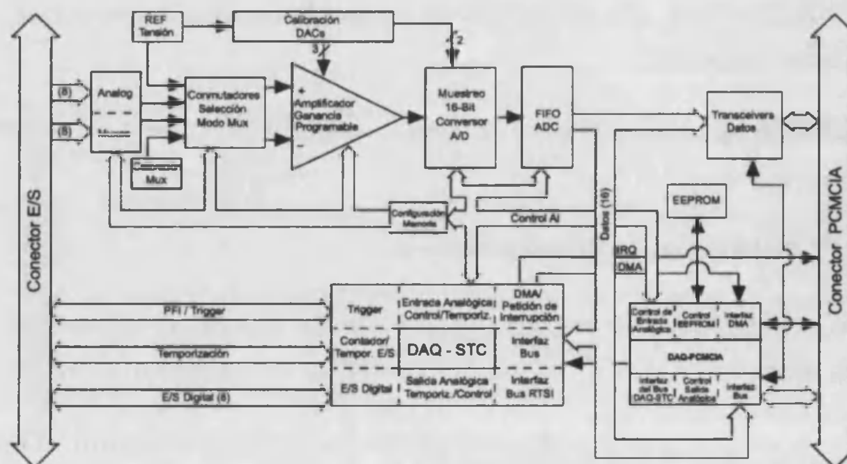


Figura 3.5: Diagrama de bloques de la DAQCard-AI-16XE-50.

La DAQCard-AI-16XE-50 tiene un rango unipolar de entrada de 10V (0 a 10V) y

un rango bipolar de entrada de 20V (±10V). La polaridad también puede ser programada por canal para que cada canal de entrada pueda configurarse de forma separada.

Configuración	Descripción
DIFF	Un canal configurado en modo DIFF usa dos líneas de entrada de canales analógicos. Una línea conecta a la entrada positiva del amplificador de instrumentación de ganancia programable (PGIA) del DAQCard, y el otro conecta a la entrada negativa del PGIA.
RSE	Un canal configurado en modo RSE usa una línea de entrada de canal analógico, que conecta a la entrada positiva del PGIA. La entrada negativa del PGIA es internamente llevada a la entrada de tierra analógica (AIGND).
NRSE	Un canal configurado en modo NRSE usa una línea de entrada de canal analógico, que conecta a la entrada positiva del PGIA. La entrada negativa del PGIA conecta a la entrada analógica AISENSE.

Tabla 3.1: Descripción de las configuraciones de entrada.

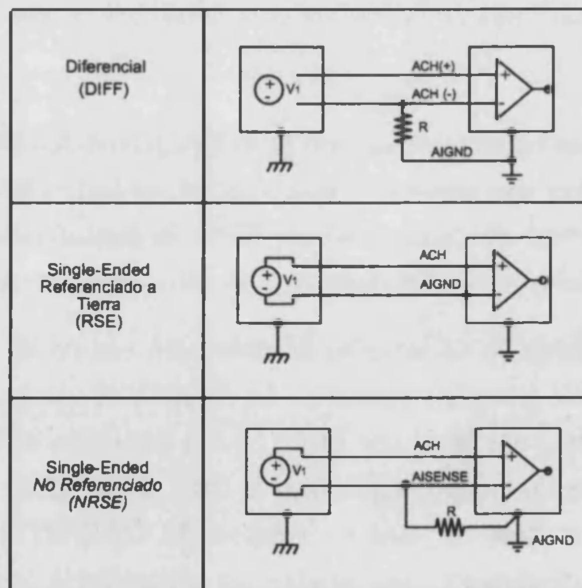


Figura 3.6: Tipos de conexiones analógicas con entrada flotante.

La ganancia programable por software en estas tarjetas incrementa la flexibilidad. Se utiliza para hacer coincidir los rangos de la señal de entrada a aquéllos que el ADC puede acomodar. La DAQCard-AI-16XE-50 está ideada para una amplia variedad de niveles de señal, tiene ganancias de 1,2,10 y 100. Con una configuración de ganancia correcta, se puede usar el ADC a completa resolución para adquirir la señal de entrada.

La tabla 3.2 muestra la resolución obtenida de acuerdo al rango de configuración y ganancia.

Configuración Rango	Ganancia	Rango Actual Entrada	**Resolución**
0 to +10 V	1.0	0 to +10 V	2.44 mV
	2.0	0 to +5 V	1.22 mV
	5.0	0 to +2 V	488.28 μV
	10.0	0 to +1 V	244.14 μV
	20.0	0 to +500 mV	122.07 μV
	50.0	0 to +200 mV	48.83 μV
	100.0	0 to +100 mV	24.41 μV
–5 to +5 V	0.5	–10 to +10 V	4.88 mV
	1.0	–5 to +5 V	2.44 mV
	2.0	–2.5 to +2.5 V	1.22 mV
	5.0	–1 to +1 V	488.28 μV
	10.0	–500 to +500 mV	244.14 μV
	20.0	–250 to +250 mV	122.07 μV
	50.0	–100 to +100 mV	48.83 μV
	100.0	–50 to +50 mV	24.41 μV

Tabla 3.2: Precisiones de la medida con diferentes rangos y ganancias.

Al igual que todas las tarjetas de la serie E, la DAQCard-AI-16XE-50 contiene ocho líneas de E/S digital para uso general. Éstas pueden ser individualmente configuradas a través de software como entrada o salida. Tras la inicialización del sistema o un reset, los puertos digitales E/S se encuentran en alta impedancia.

El DAQ-STC proporciona un interfaz flexible para conexión de señales de temporización a otras tarjetas o circuitos externos. La DAQCard usa los pines PFI (entradas de función programables) del conector de I/O para conexión al circuito externo. Estas conexiones están diseñadas para habilitar la DAQCard tanto para controlar como para ser controlada por otras tarjetas y circuitos. El DAQ-STC tiene un total de 13 señales internas de temporización que pueden ser controladas por una fuente externa. Estas señales de temporización pueden también ser controladas por señales generadas internamente por el DAQ-STC, y son enteramente configurables por software.

Algunas funciones realizadas por las tarjetas de la serie E requieren un base temporal para generar las señales de temporización, necesarias para controlar conversiones A/D, actualizaciones DAC, o señales de propósito general en el conector I/O. Estas señales pueden ser bien externas o bien una base temporal de 20MHz generada internamente.

3.3.4 DAQCard-AI-16XE-50 versus AT-MIO-16E-10

A continuación se describen las características de ambas tarjetas:

DAQCard-AI-16XE-50 (NI-DAQCard, 1999)

- Bus tipo PCMCIA II
- 16 entradas *single-ended*, 8 diferenciales.
- 200Kmuestras/s un canal o 20Kmuestras/s multicanal.
- 16 bits de resolución.
- Polaridades: unipolar 0-10V, bipolar ±10V.
- Ganancia 1,2,10,100.
- 8 líneas E/S digital.
- 2 contadores / temporizadores.
- Trigger digital.

AT-MIO-16E-10 (NI-AT, 1996)

- Bus tipo ISA
- 16 entradas *single-ended*, 8 diferenciales.
- 2 líneas analógicas de salida.
- 100Kmuestras/s multicanal.
- 12 bits de resolución.
- Polaridades: unipolar 0-10V, bipolar ±10V, bipolar ±5V.
- Ganancia 1,2,5,10,20,50,100.
- 8 líneas E/S digital.
- 2 contadores / temporizadores.
- Trigger digital.

Ambas tarjetas son muy similares: idéntico número de líneas de entrada analógicas, las mismas líneas digitales, contadores/temporizadores, etc. Las diferencias residen básicamente en la velocidad, la resolución y el conjunto de ganancias permisibles en cada una de ellas. Éstos son los únicos aspectos que debe tener en cuenta el usuario cuando realiza la selección de una u otra tarjeta para la adquisición.

3.4 Programación del sistema de adquisición

Se plantea la programación de un entorno de adquisición multicanal que, básicamente, pueda realizar una adquisición continuada y almacenarla progresivamente en fichero los datos. La adquisición sólo terminará cuando se termine manualmente el proceso, cerrando entonces el fichero. El tiempo de adquisición máximo sólo dependerá de la capacidad de almacenamiento secundario disponible. El almacenamiento se realizará mediante la multiplexación de canales en un formato que pueda ser recuperado por Matlab para procesar off-line la señal adquirida.

La aplicación permitirá seleccionar el tipo de tarjeta con la que se desea realizar la adquisición, de entre las dos disponibles. La aplicación utilizará los drivers facilitados por el fabricante para la programación de la aplicación en un entorno de programación Visual C++. En esta sección se describen el entorno de desarrollo, el método de adquisición elegido, denominado doble-buffer y un análisis de los objetos de la aplicación.

3.4.1 NI-DAQ

NI-DAQ es un conjunto de funciones que controlan todos los dispositivos DAQ de National Instruments para: E/S analógica, E/S digital, temporización, acondicionamiento de señales SCXI y sincronización multitarjeta RTSI (NI-DAQ, 1998).

NI-DAQ tiene tanto funciones E/S *de alto nivel* para un uso sencillo, como complicadas funciones E/S de *bajo nivel* para conseguir un máximo de flexibilidad y prestaciones. Ejemplo de funciones de alto nivel es la adquisición a fichero de un determinado número de muestras. Ejemplo de las funciones de bajo nivel son la escritura directa a los registros del dispositivo DAQ o la calibración de las entradas analógicas.

NI-DAQ incluye un *buffer* y un *gestor de datos.* Este último utiliza sofisticadas técnicas para el manejo y gestión de de los buffers de adquisición, por lo que se puede adquirir y procesar datos simultáneamente. NI-DAQ puede transferir datos usando DMA, interrupciones, o polling por software. Se pueden usar transferencias DMA para transferir los datos a la memoria por encima de los 16MB aún sobre computadores con bus ISA.

Se utiliza un *gestor de recursos* para permitir el uso simultáneo de distintas funciones y diversos dispositivos. El gestor de recursos previene la contención multitarjeta sobre canales DMA, niveles de interrupción y canales RTSI.

NI-DAQ posee una extensa librería de funciones que pueden ser invocadas desde cualquiera que sea el entorno de programación usado. Estas funciones incluyen rutinas para entrada analógica (conversión A/D), buffer de adquisición de datos (conversión

A/D de alta velocidad), salida analógica (conversión D/A), generación de formas de onda (conversión temporizada D/A), E/S digital, operaciones de cuenta y temporización, SCXI, RTSI, calibración, mensajería, y adquisición de datos a la memoria extendida.

NI-DAQ puede enviar *mensajes conducidos por eventos* a aplicaciones Windows o Windows NT, cada vez que ocurre un evento especificado por el usuario. Así, se elimina el polling y se pueden, como en nuestro caso, desarrollar aplicaciones DAQ gestionadas por eventos. Estos eventos son programables. Algunos ejemplos de eventos del NI-DAQ son:

- Cuando un número especificado de muestras analógicas ha sido adquirido.
- Cuando el nivel analógico y la pendiente de una señal coinciden con los niveles especificados.
- Cuando la señal está dentro o fuera de una banda de tensión.
- Cuando coincide un patrón especificado de E/S digital.
- Cuando ocurre un flanco de subida o bajada en una línea de temporización.

3.4.2 El entorno de desarrollo

Las funciones del NI-DAQ permiten utilizar varios entornos de desarrollo integrado (IDE) de aplicaciones bajo Windows 95/NT, como son: Microsoft Visual C++, Visual Basic y Borland C++. En nuestro caso se ha utilizado el Microsoft Visual C++ 6.0 sobre NT 4.0, siendo la aplicación resultante compatible con los sistemas operativos Windows NT 4.0, Windows 95 y Windows 98 (NI-DAQ, 1998).

Bajo Windows, el NI-DAQ está constituido por bibliotecas de funciones en forma de DLLs, lo cual implica que las rutinas NI-DAQ no son enlazadas en ficheros ejecutables de aplicaciones. Para el desarrollo de la aplicación, las bibliotecas importadas contienen información acerca de las funciones exportadas por las DLLs. Éstas indican la presencia y localización de rutinas DLL.

3.4.3 Técnica de adquisición continua de datos (Doble Buffer)

Las técnicas convencionales de software para la adquisición de datos funcionan bien para la mayoría de las aplicaciones. Sin embargo, para aplicaciones más sofisticadas, que implican mayores cantidades de entrada o salida de datos con frecuencias de muestreo elevadas, se requieren técnicas más avanzadas. Una de tales técnicas es el *doble buffer*, que será descrita a continuación por ser la utilizada en la adquisición

de la aplicación. National Instruments implementa técnicas de doble buffer en sus drivers software para entrada o salida continua de grandes cantidades de datos.

Buffer Simple Vs Doble Buffer

Sin embargo, antes de proceder a la descripción de esta técnica vamos a establecer una comparación respecto a la técnica básica: El buffer simple. Éste es el método más común de adquisición de datos. En operaciones de entrada con un buffer sencillo, se adquiere un número de muestras a una frecuencia de muestreo especificada y los datos son transferidas a la memoria de la computadora. Cuando finaliza el almacenamiento de los datos, el computador puede analizar, visualizar o almacenar los datos en disco para posterior procesado.

Las operaciones con buffer simple son relativamente sencillas de implementar, y son muy usadas en algunas aplicaciones. La mayor desventaja de las operaciones de buffer sencillo es que la cantidad de datos que puede ser introducida en cualquier instante está limitada por la cantidad de memoria libre disponible.

En operaciones con doble buffer, el buffer de datos se configura como un buffer circular. Para operaciones de entrada, el dispositivo DAQ rellena el buffer circular con datos. Cuando se alcanza el final del buffer, el dispositivo retorna al principio del buffer y rellena éste de nuevo con datos. Este proceso continúa *ad infinitum* hasta que es interrumpido por un error hardware o parado por una función de llamada.

A diferencia de las operaciones con buffer sencillo, las operaciones con doble-buffer reutilizan el mismo buffer y son, por lo tanto, capaces de introducir un infinito número de muestras sin requerir infinita cantidad de memoria. Sin embargo, debe existir un mecanismo por el cual acceder a los datos para almacenamiento, actualización y procesado. Ésto se describe en la sección siguiente.

Adquisición con Doble Buffer

El buffer de datos para las operaciones doble-buffer de entrada está configurado como un buffer circular. En suma, NI-DAQ divide el buffer en dos mitades iguales. Con lo que se puede coordinar el acceso del usuario al buffer de datos con el dispositivo DAQ. El esquema de coordinación es simple, la aplicación debe proporcionar un buffer de transferencia que debe ser utilizado para ir vaciando el buffer circular.

La operación de entrada con doble-buffer comienza cuando el dispositivo DAQ comienza a escribir datos en la primera mitad de el buffer circular (figura 3.7a). Después de que el dispositivo comience a escribir la segunda parte del buffer circular, el NI-DAQ puede copiar los datos desde la primera mitad al buffer de transferencia (figura 3.7b). En este punto se deben almacenar los datos en el bloque de transfe-

rencia a disco o procesarlos de acuerdo a las necesidades de la aplicación. Cuando la entrada del dispositivo ha rellenado la segunda mitad del buffer circular, el dispositivo retorna a la primera mitad del buffer y comienza a reescribir sobre los datos viejos. La aplicación puede ahora copiar la segunda mitad del buffer circular al buffer de transferencia (figura 3.7c). Así, los datos del buffer de transferencia están de nuevo disponibles para ser usados por la aplicación. El proceso puede ser repetido sin fin para producir una corriente de datos continua. Nótese que la figura 3.7d es equivalente a 3.7b y el comienzo de un ciclo de dos pasos.

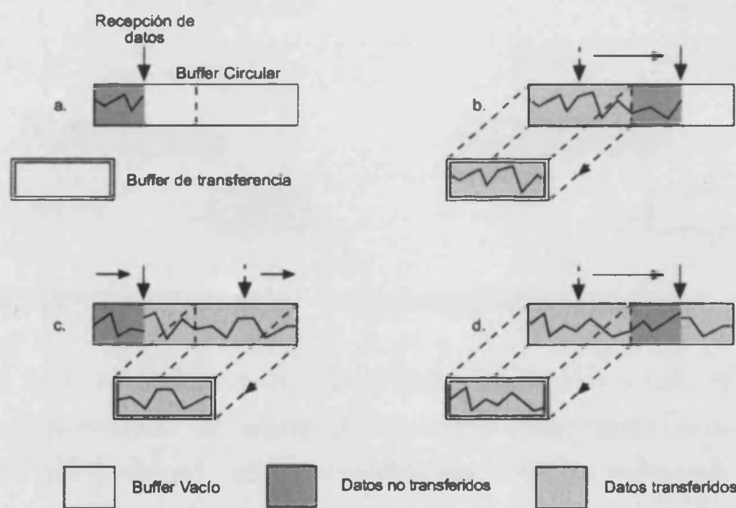


Figura 3.7: Adquisición con doble-buffer y transferencias de datos secuenciales.

Sin embargo, el mecanismo de doble-buffer no es un esquema de adquisición exento de problemas. Una aplicación puede experimentar dos posibles problemas. El primero es la posibilidad de que el dispositivo DAQ sobreescriba los datos antes de que el NI-DAQ los haya copiado en el buffer de transferencia. La situación se ilustra en la figura 3.8.

Nótese que en la figura 3.8b, el NI-DAQ deja perder la oportunidad de copiar los datos de la primera mitad del buffer circular mientras el dispositivo DAQ escribe los datos en la segunda mitad. Como resultado, el dispositivo DAQ empieza sobre-escribiendo el dato en la primera mitad del buffer circular antes de que el NI-DAQ haya copiado éste en el buffer de transferencia (figura 3.8c). Para garantizar que los datos adquiridos no se corrompen, se debe esperar a que el dispositivo termine de sobreescribir los datos en la primera mitad, antes de copiar los datos en el buffer de transferencia. Después de que el dispositivo haya empezado a escribir la segunda mitad, se pueden copiar los datos desde la primera mitad del buffer circular al buffer de transferencia (figura 3.8d). En esta situación, el NI-DAQ devuelve una

precaución por sobreescritura. Esta precaución indica que los datos en el buffer de transferencia son válidos, pero algunas entradas anteriores han sido perdidas. Las transferencias subsiguientes no devolverán más precauciones mientras se mantenga el ritmo de adquisición conforme a la figura 3.7.

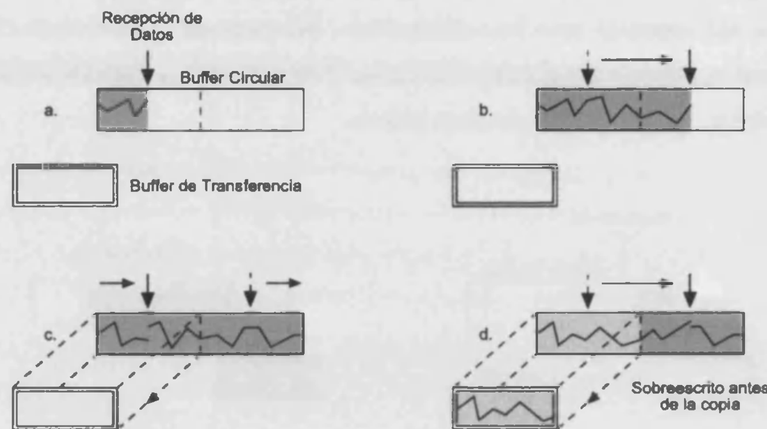
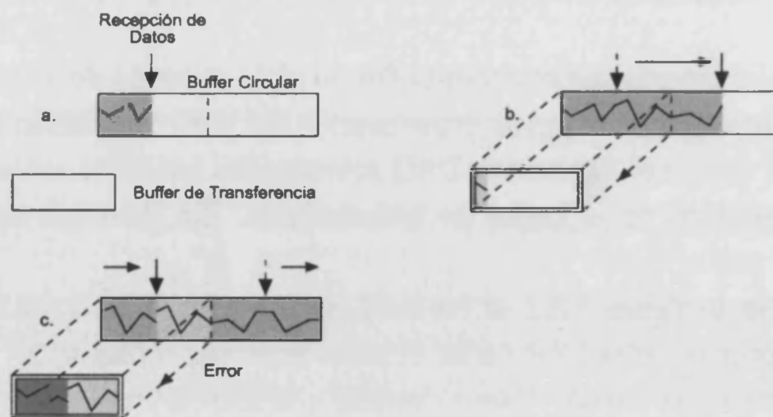


Figura 3.8: Adquisición con doble-buffer: Sobreescritura antes de la copia.

El segundo problema potencial ocurre cuando se sobreescriben los datos que la aplicación está simultáneamente copiando al buffer de transferencia. Cuando esto ocurre, NI-DAQ devuelve un error de sobreescritura. La situación es ilustrada en la figura 3.9.



Double-Buffered Input with an Overwrite

Figura 3.9: Adquisición con doble-buffer: Sobreescritura.

En la figura 3.9b, se ilustra como se han comenzado a copiar los datos desde la primera mitad del buffer circular al buffer de transferencia. Sin embargo, no es posible

la copia de la mitad entera antes de que el dispositivo DAQ comience a sobreescribir los datos de la primera mitad del buffer (Figura 5-3c). Consecuentemente, los datos copiados en el buffer de transferencia pueden estar corruptos; esto es, pueden contener datos nuevos y viejos. Las transferencias futuras se ejecutarán normalmente si este problema no vuelve a ocurrir, sin embargo, como ya sabemos, en esta transferencia se envía un error de sobreescritura.

3.4.4 Interfaz de usuario

La interfaz con el usuario aprovecha al máximo las facilidades proporcionadas por el sistema operativo Windows para simplificar las acciones del usuario, mostrar la información con claridad y facilitar el uso de la aplicación.

La interfaz se ha estructurado como muestra la figura 3.10, donde se planifican las siguientes regiones en la pantalla:

- **Barra de herramientas y de título:** Proporciona un acceso rápido a las funciones más habituales.
- **Parámetros generales de la adquisición:** Contiene información, visible en todo momento, sobre la adquisición en general:
  - Dispositivo origen de la adquisición.
  - Frecuencia de muestreo base.
  - Canales seleccionados para la adquisición.
  - Fichero destino de la adquisición.
  - ..
- **Barra de selección de canal:** Permite mediante una serie de tabs, que enumeran exactamente los canales seleccionados, seleccionar el canal activo (aquél cuya configuración se visualiza en el área correspondiente y, en adquisición, también se visualiza la forma de onda adquirida, a modo de osciloscopio).
- **Configuración del canal seleccionado:** Muestra la configuración propia para la adquisición y visualización del canal activo.
- **Visualización de la forma de onda:** Área reservada para la visualización, durante la adquisición, de la forma de onda adquirida, a modo de osciloscopio, del canal seleccionado.

3.4.5 Programación de la aplicación

A continuación se presenta una introducción a la programación orientada a objetos y a la metodología de desarrollo OMT (técnica de modelado de objetos). La descripción realizada se basa en el modelo de objetos y el modelo de los eventos más importantes de la aplicación (Langsdam, 1997; Deitel, 1994).

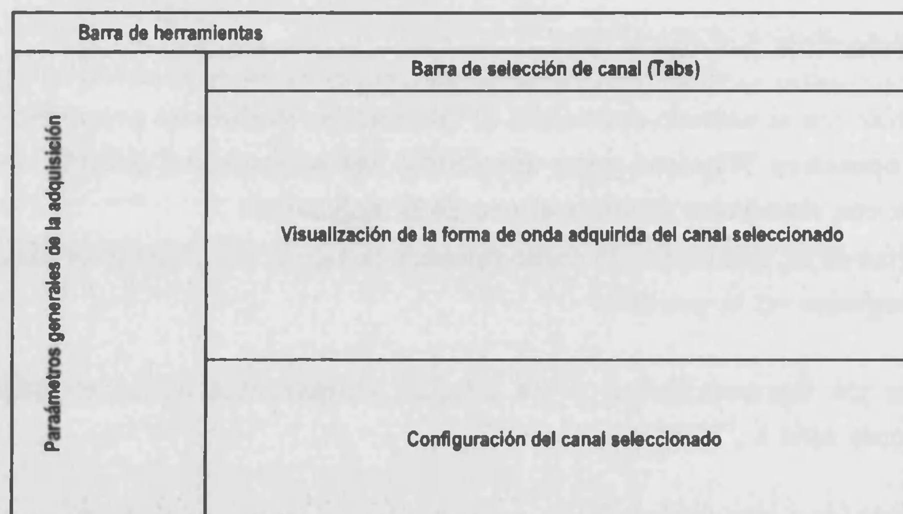


Figura 3.10: Formato de la interfaz con el usuario.

Programación orientada a objetos

La programación orientada a objetos representa una metodología de programación que se basa en las siguientes características:

- Los diseñadores definen nuevas clases (o tipos) de objetos.
- Los objetos poseen una serie de operaciones asociadas a ellos.
- Las operaciones tienden a ser genéricas, es decir, a operar sobre múltiples tipos de datos.
- Las clases de objetos comparten componentes comunes mediante el mecanismo de herencia.

Se entiende por *objeto* al elemento de programa que engloba una estructura de datos y un conjunto de procedimientos (o métodos) que operan sobre dicha estructura. Una *clase* es una colección de objetos que poseen características y operaciones

comunes. Las clases contienen toda la información necesaria para generar objetos con unos ciertos atributos y operaciones.

Un lenguaje de programación es considerado como orientado a objetos cuando posee las siguientes características: ***Encapsulación*** (técnica que permite ocultar los detalles de un objeto), *Herencia* (permite crear una clase a partir de otra u otras ya existentes que tengan propiedades que se desea que estén presentes en la nueva clase), y *Polimorfismo* (significa que más de un método tiene el mismo nombre, esto permite tratar con una misma rutina diferentes tipos de datos. En unión con la herencia, el polimorfismo posibilita la reutilización de código en las aplicaciones).

Metodología de desarrollo O.M.T

En la metodología O.M.T. (*Object Modelling Technique*), el principal constructor es el objeto, que combina datos estructurados y comportamiento en una sola entidad. Esta metodología abarca las fases principales del ciclo de vida de un programa: análisis, diseño e implementación. Para la descripción de la aplicación, nos centraremos en la fase de análisis. En esta se definen tres modelos:

- **Modelo de Objetos**: Refleja la estructura estática del sistema.
- **Modelo de Eventos**: Describe la estructura dinámica.
- **Modelo Funcional**: Explica cómo se transforman los datos.

Cada uno de estos modelos tiene asociados varios tipos de diagramas. En el modelo de objetos se utilizan diagramas de clases. En ellos las clases se ordenan en jerarquías de herencia y pueden estar asociadas o agregadas con otras clases. También pueden utilizarse diagramas que contengan objetos, llamados diagramas de instancias. Los objetos o clases se representan mediante recuadros, como muestra la figura 3.11. En estos recuadros se pueden enumerar los atributos del objeto o clase, y también las operaciones.

Dada la simplicidad de la aplicación que nos ocupa, únicamente se describirá el modelo de objetos y el modelo de eventos.

Los conceptos más importantes en el modelo dinámico son los eventos y los estados. Un evento es algo que ocurre en el entorno del sistema y que prácticamente no tiene duración comparado con su escala de tiempo, eventos pueden ser la llegada de un mensaje de red o la pulsación de un botón del ratón por parte del usuario. Un evento es un camino de información de un objeto a otro (Schildt, 1995).

Los eventos se representan en un diagrama de traza de eventos. Este diagrama se compone de los siguientes elementos:

- **Objetos**: Se muestran como una línea vertical.
- **Eventos**: Se representan como una flecha horizontal desde el objeto emisor al objeto receptor.

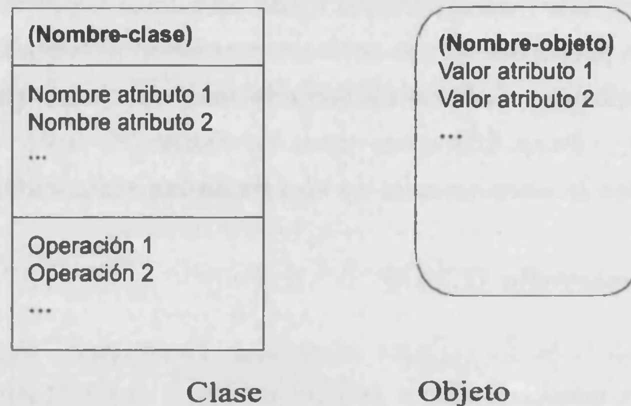


Figura 3.11: Elementos de los diagramas del modelo de objetos.

El tiempo crece desde la parte superior del diagrama hacia la parte inferior del mismo. Las figuras 3.12 y 3.13 son una muestra de la traza de eventos.

3.4.6 Análisis de la aplicación

El objetivo del análisis es construir un modelo que describa el comportamiento del sistema. En primer lugar, se describen las clases a través de los atributos y operaciones que pueden considerarse en cada una. Posteriormente, se muestran las trazas de eventos más importantes que se producen entre los objetos. Las trazas de eventos permiten especificar cómo se transmite información entre los objetos y en qué secuencia realizan éstos las suboperaciones para llevar a cabo una determinada operación.

Diccionario de clases

En la aplicación pueden identificarse las siguientes clases y estructuras de datos:
Clases:

CvibraApp Clase aplicación derivada de la clase *CwinApp*. Tiene asociadas tres instancias de objetos documento, frame y vista. Éstas son, respectivamente, las instancias de las clases *CvibraDoc*, *CmainFrame* y *CviewGeneral*.

CmainFrame El objeto generado de esta clase presenta y gestiona el frame principal, derivado de la clase *CFrameWnd*. Éste tendrá partida el área del cliente

en cuatro partes, asociadas a vistas diferentes, instancias de las cuatro clases derivadas de vistas que se citan a continuación.

CvibraDoc Derivada de *Cdocument*, es la encargada de cargar durante la inicialización de la aplicación, y salvar a su finalización los parámetros tanto generales como propios de cada uno de los canales.

CviewGeneral Derivada de *CformView*, gestiona el formulario de parámetros generales de la adquisición, figura 3.10.

CviewTabs Derivada de *CformView*, gestiona la barra de selección de canal, figura 3.10. En ésta se indican los canales seleccionados en la adquisición y el canal activo.

CviewCanal Derivada de *CformView*, gestiona el formulario de configuración de canal, figura 3.10. Los parámetros particulares para cada canal cambian conforme a la selección realizada sobre la barra de tabs.

CviewVibra Derivada de *CView*, gestiona la vista del osciloscopio de visualización de la forma de onda cuando la adquisición está en curso. Las propiedades de este objeto realizan el inicio y detención de la adquisición, así como la visualización de la forma de onda del canal seleccionado y la escritura a fichero.

Estructuras:

AnalogCanalStruct Atributo de los objetos generados de *CvibraDoc*, utilizada para almacenar los parámetros propios de cada canal.

GeneralStruct Estructura utilizada para almacenar los parámetros generales de la aplicación.

ScanData Estructura utilizada para contener parámetros que serán usados para la configuración de la adquisición cuando ésta sea iniciada.

ScreenBuffers Atributo de *CviewVibra*, utilizada para gestionar los buffers de visualización de cada canal.

VibraBuffers Atributo de *CviewVibra*, utilizada para gestionar los buffers de almacenamiento intermedio para almacenamiento en fichero.

El apéndice L describe el diagrama de clases de todas las clases y estructuras citadas.

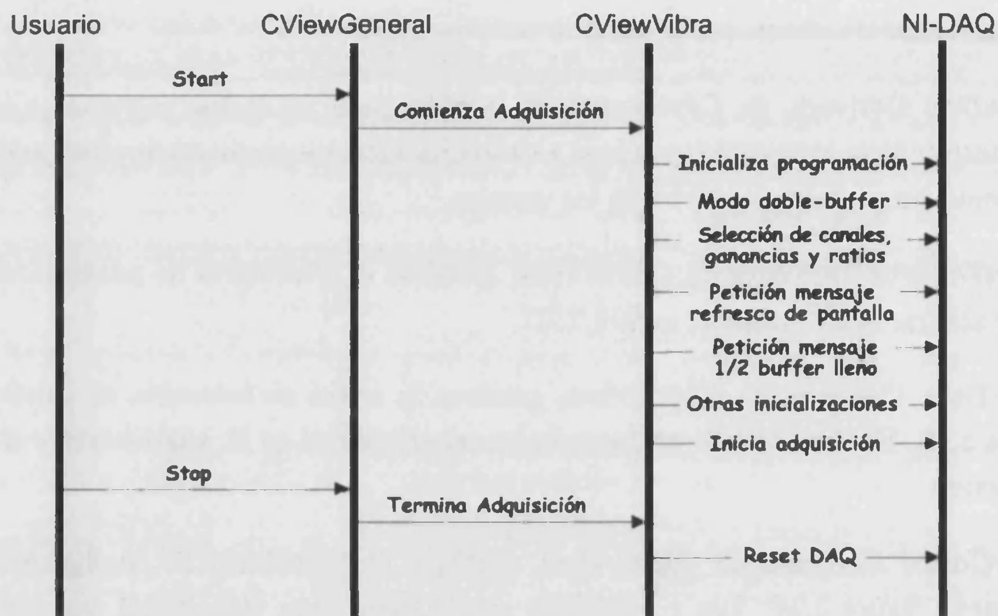


Figura 3.12: Inicialización y finalización de la adquisición.

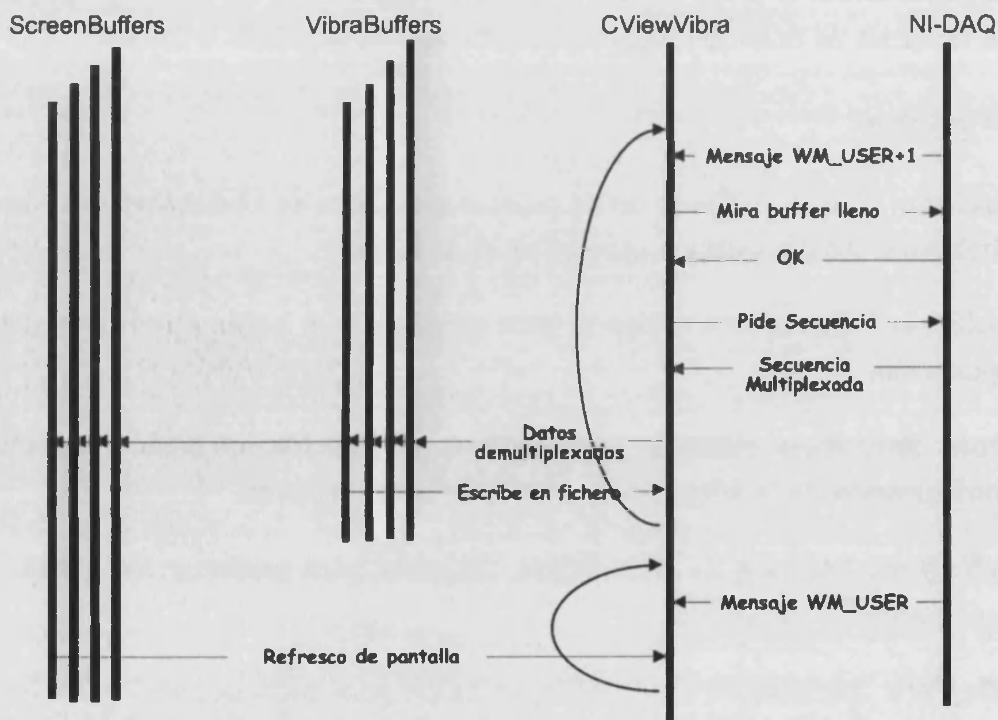


Figura 3.13: Adquisición doble-buffer y refresco de pantalla.

Trazas de eventos

Las figuras 3.12 y 3.13 representan las trazas de eventos más importantes de la aplicación. Por un lado, en 3.12 se representa la traza de la inicialización y finalización de la adquisición. Por otro lado, la figura 3.13 representa los eventos de adquisición y refresco de pantalla.

El código fuente se aporta en el apéndice M. La aplicación debe ejecutarse sobre un computador con una tarjeta DAQCard-AI-16XE-50 o AT-MIO-16E-10 de National Instruments previamente instalada y configurada.

3.5 Funcionamiento de la aplicación

La figura 3.14 muestra el interfaz de la aplicación de adquisición. Como puede observarse, éste sigue los requerimientos exigidos por las especificaciones de la sección 3.4.4, pudiéndose fácilmente identificar las partes allí citadas.

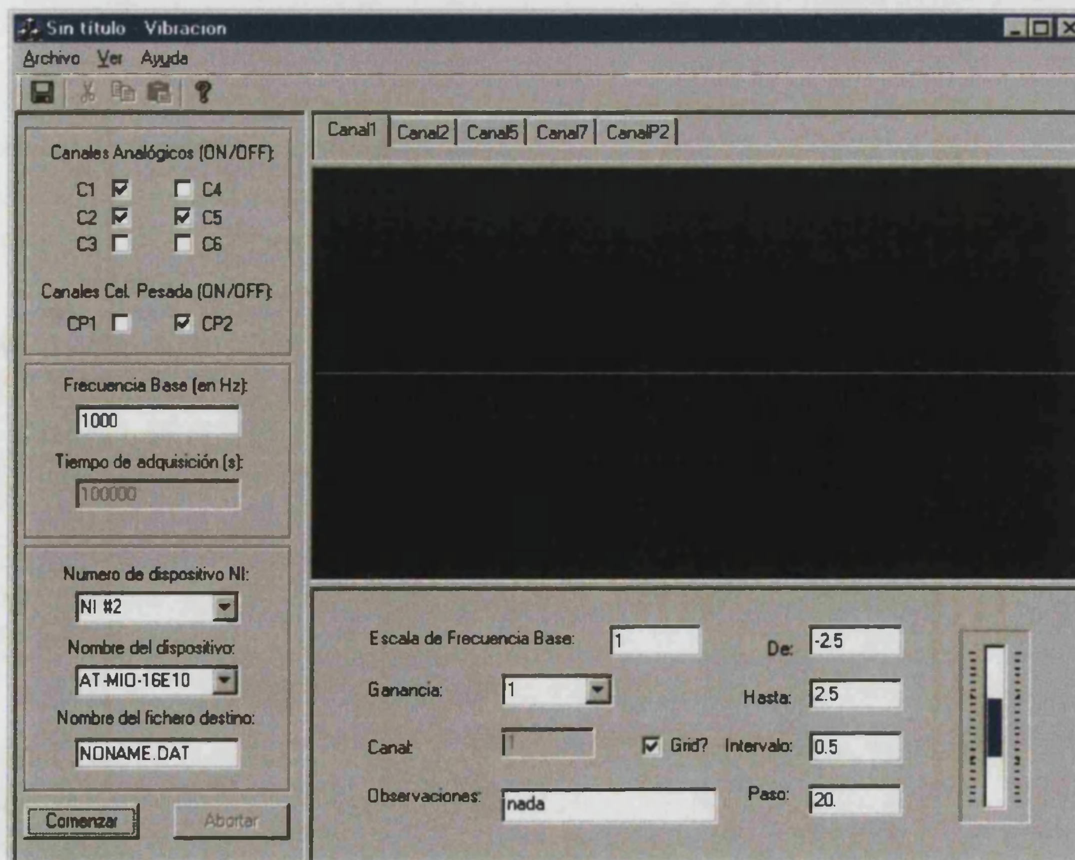


Figura 3.14: Interfaz de usuario del sistema de adquisición.

La primera área a la que se ha de acudir para realizar una adquisición es al formulario general, sito en la columna izquierda de la ventana. En él se incorporan datos generales de la configuración. Si examinamos su funcionamiento por campos, podemos ver los siguientes:

- **Canales analógicos ON/OFF**: Conjunto de cuadros de selección que representan hasta seis canales analógicos de adquisición, los canales diferenciales 0 a 5. Indican si el canal asociado participará en la próxima adquisición. De esta intuitiva manera se incorporan o eliminan de la adquisición unos canales u otros.

  El proceso de selección de canales sólo puede realizarse cuando el sistema no está adquiriendo. Una vez en marcha la adquisición, no podrán modificarse hasta su finalización.

- **Canales Células de Carga (ON/OFF)**: Son dos cuadros de selección equivalentes a los anteriormente descritos. Sin embargo, hacen referencia a las células de carga, conectadas únicamente a los canales diferenciales 6 y 7, correspondientes a las células de carga 1 y 2.

  La selección indica la participación en la adquisición de los canales específicamente utilizados, en nuestro diseño, para las células de carga.

- **Frecuencia Base (en Hz):** Este cuadro de edición permite introducir, en Hertzios, la frecuencia de muestreo que se utilizará como frecuencia base de la adquisición. En este punto cabe notar que puede realizarse la adquisición de los distintos canales con frecuencias de muestreo diferentes. Sin embargo, deben ser fracciones de la frecuencia base, en la forma $f=f_{base}/N$ con N entero.

  De esta forma, por ejemplo puede realizarse una adquisición de tres canales, el canal 1 a la frecuencia base de 100Hz ($f_1 = f_{base}/1$), el canal 2 a 50Hz ($f_2 = f_{base}/1$), y el canal 3 a 5hz ($f_3 = f_{base}/20$). De aquí la importancia de la frecuencia base para establecer las frecuencias de muestreo de los diferentes canales, aunque sean diferentes.

- **Número de dispositivo NI:** Cada una de las tarjetas de NI posee un número de dispositivo, establecido en su configuración, que, básicamente, ayuda a los drivers del NI-DAQ a discernir entre ellas.

- **Nombre del dispositivo:** El sistema de adquisición realizado no está basado solamente en un tipo de tarjeta de National Instruments. Éste puede seleccionar, dada la disponibilidad de dos de ellas, entre dos la DAQCard-AI-16XE-50 y la AT-MIO-16E-10. De cualquier forma, la tarjeta seleccionada debe estar configurada con el número de dispositivo indicado anteriormente.

La indicación del tipo de dispositivo es necesaria para realizar algunos ajustes en la adquisición, como por ejemplo las ganacias posibles en cada una de ellas, necesarios para su correcto funcionamiento.

- **'Comenzar' y 'Abortar':** Estos botones corresponden al comienzo y finalización de la adquisición. En cualquier momento sólo uno de ellos estará activo, informando así del estado en el que se encuentra el sistema.

Como vemos, este formulario corresponde a parámetros generales de la aplicación. Sin embargo, cada uno de los canales seleccionados para la adquisición también debe configurarse individualmente en cuanto a aquellos parámetros que puedan variar de uno a otro. Para ello se dispone de un formulario de configuración de canal. Éste permite actuar tanto sobre la configuración individual de la adquisición para este canal, como en la configuración de su visualización.

En el formulario de configuración de canal, se presentan campos relativos a parámetros propios de la adquisición como:

- **Escala de la frecuencia Base:** Anteriormente ya se ha descrito cómo la frecuencia de muestreo de un canal individual puede diferir de la frecuencia de muestreo base según una fracción $f = f_{base}/N$, con N entero. En este caso, el entero N será la escala respecto de la frecuencia base, mediante la cual se halla la frecuencia de muestreo del canal seleccionado.

- **Ganancia:** Los dispositivos DAQ proporcionan una serie de ganacias que dependen del tipo de dispositivo. En este combo indicado se listan las posibles ganacias dependiendo del tipo de dispositivo DAQ seleccionado.

Los campos de este formulario, relativos a visualización son:

- **Grid:** Indica si se pretende una visualización con o sin *grid*.

- **De y hasta:** Acotan la visualización de la señal adquirida para dicho canal (en Voltios con gancia a 1) en el eje de las ordenadas.

- **Intervalo:** Es el intervalo del *grid* en el eje de ordenadas.

- **Paso:** Es relativo a la visualización. Se pretende que la señal visualizada se desplace suavemente conforme progresa la adquisición, por ello este campo indica el número de muestras entre los que se enviará un mensaje de refresco de la visualización.

Además de estos, existen otros campos como:

- **Número de canal:** Indica cuál es el canal del cual se presenta la información.
- **Observaciones:** Corresponde a un String en el cual pueden realizarse anotaciones y comentarios sobre la adquisición, el contexto en el que produce, etc.

Nótese que el formulario de configuración de canal sólo hace referencia a la información de un canal. Así, debe establecerse un método por el cual la información que aparezca en este formulario pueda pertenecer a uno u otro canal. Este mecanismo es realizado de forma muy intuitiva por la barra de selección de canal, véase figura 3.14. Ésta está formada por un conjunto de tabs que se corresponden unívocamente con aquellos cuados de selección seleccionados en el formulario general. Nótese que, caso de existir al menos un tab, siempre está activo uno y sólo uno de ellos. La figura 3.15 ilustra como en el ejemplo de la figura 3.14, aparecen 5 tabs que se corresponden a la selección en el form general de los canales 1, 2, 5 y 7, y al canal de la célula de pesada 2. En este caso el canal activo es el canal 1.

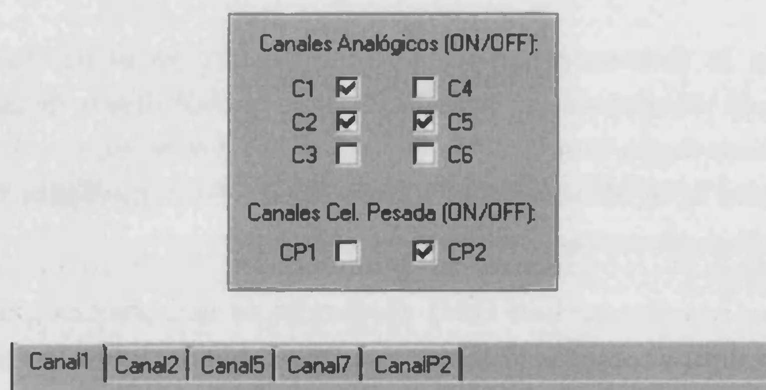
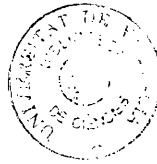


Figura 3.15: Relación entre la barra de tabs y los canales seleccionados.

De esta forma, la adquisición se realiza contínuamente, escribiendose progresivamente el resultado en disco. Ésta se escribe entrelazada, mediante un formato de bloques legible por Matlab para procesar off-line estas señales.



Capítulo 4

Caracterización y algoritmos

Tradicionalmente, a partir de la señal adquirida de una célula de carga, se obtiene el valor del peso mediante un simple promediado. Esta aproximación es válida para bajas velocidades de la cinta de arrastre, hasta 6 frutas/segundo, en las que este simple método es suficiente para obtener una estimación del valor del peso con una precisión de un gramo siempre que la plataforma de pesado aisle las vibraciones de la máquina de forma eficiente.

Nótese que, a partir de este momento, se utilizan 'frutas/segundo' como unidades de velocidad de la cinta de arrastre. Esta unidad define la velocidad de la cinta de arrastre, que también puede determinarse a través de la frecuencia, en hertzios, del variador, fijando indirectamente el número de tazas por segundo que pasan sobre la célula de carga en una determinada línea.

Este capítulo presenta la caracterización de la señal de trabajo y el modelo del sistema. Posteriormente se realiza una prospección de nuevos algoritmos de preprocesado de la señal. Cabe destacar la importancia del preprocesado para la estimación de un valor preciso de peso, manteniendo o mejorando la precisión típica de $\pm 1g$ de error, pues éstos realizan el acondicionamiento inicial que limpia la señal y posibilitan el incremento de la velocidad de la cinta de arrastre en clasificación.

4.1 Caracterización de la señal

Una taza es similar a los dedos de una mano en forma de copa que transportan el fruto a velocidad constante a través de las secciones de pesado y visión (la presencia del sistema de visión es opcional). Finalmente, éstas son levantadas individualmente ante una de las cintas de la sección de salida para expulsar el fruto que contienen.

Las tazas son transportadas por la cinta de arrastre rotativa a la que van unidas, que es la responsable de hacerlas pasar sobre la célula de carga a velocidad constante. La figura 4.1 ilustra el mecanismo de pesado dinámico, en el que se distingue una zona denominada *patín* formada por un soporte de aluminio fijado al chasis de la calibradora, y cubierto de Teflon, un material de bajo rozamiento, que además minimiza el impacto de las tazas al entrar al tramo aislado de patín situado sobre la célula de carga.

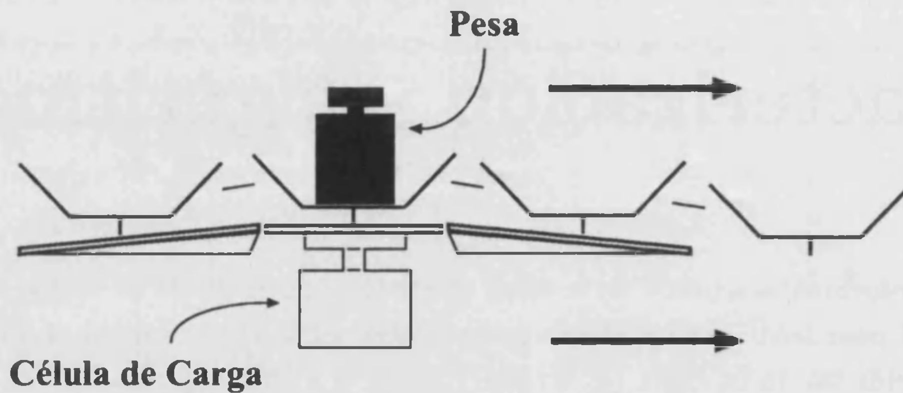


Figura 4.1: Esquema del sistema dinámico de pesado.

El patín posee inicialmente una parte ascendente, donde las tazas son elevadas paulatinamente hasta una altura en la que su peso descansa únicamente sobre el patín y no sobre la cadena de transporte. El mecanismo de fijación de la taza a la cadena proporciona un grado de libertad en este sentido, lo cual posibilita que la cinta de arrastre sólo genere la componente horizontal de movimiento, que se supone constante. Tras la elevación de la taza, sigue una parte plana en cuyo centro existe un tramo de patín aislado, fijado a la célula de carga, que es donde se realiza el proceso de pesado. Este tramo también es denominado *zona de pesado.* Como puede verse, la base de las tazas intenta proporcionar un apoyo lo más puntual posible en el sentido de la dirección del movimiento de la cadena. Esta aproximación posibilita que la transición del patín a la zona de pesado sea lo más abrupta posible (de la misma forma que la transición de salida). Así, la señal que idealmente se debería adquirir es la representada en la figura 4.2, donde puede verse que las transiciones de no haber taza a haberla sobre la zona de pesado, son abruptas. Finalmente, tras el paso sobre la zona de pesado, la taza sale de nuevo a otro tramo fijo de patín que hace descender las tazas paulatinamente, permitiendo que la cinta de arrastre recupere el soporte de la taza y su contenido.

Como sabemos, la señal que idealmente esperamos de la adquisición debería ser aproximadamente la ilustrada en la figura 4.2. En ésta, podemos distinguir dos zonas,

denominadas *zonas-peso* y *zonas-no-peso* según correspondan a las zonas en las que hay o no hay taza sobre la zona de pesado de la célula de carga. Nótese que la longitud de estos tramos es proporcional a la velocidad de la cinta de arrastre. Por otra parte, las *zonas-peso* pueden clasificarse en *llenas* o *vacías* según correspondan al paso de una taza llena o vacía sobre la célula de carga. En este caso, el nivel de la correspondiente *zona-peso* indica el peso de la taza en las zonas de taza vacía, o el peso de la taza más su contenido en las zonas de taza llena. Obviamente, la diferencia entre el nivel de esta taza cuando está vacía y el alcanzado cuando hay fruta será función del peso en gramos de la pieza.

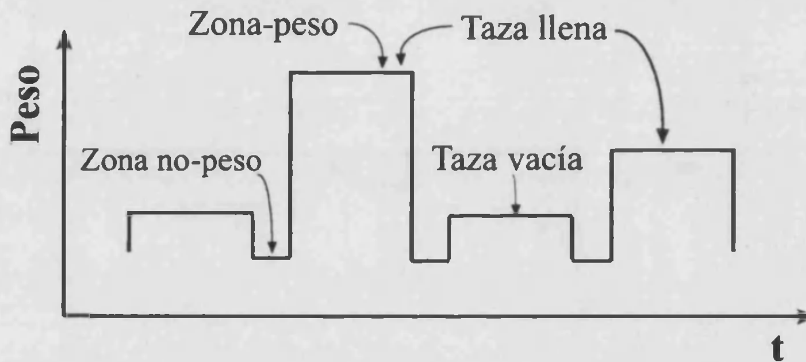


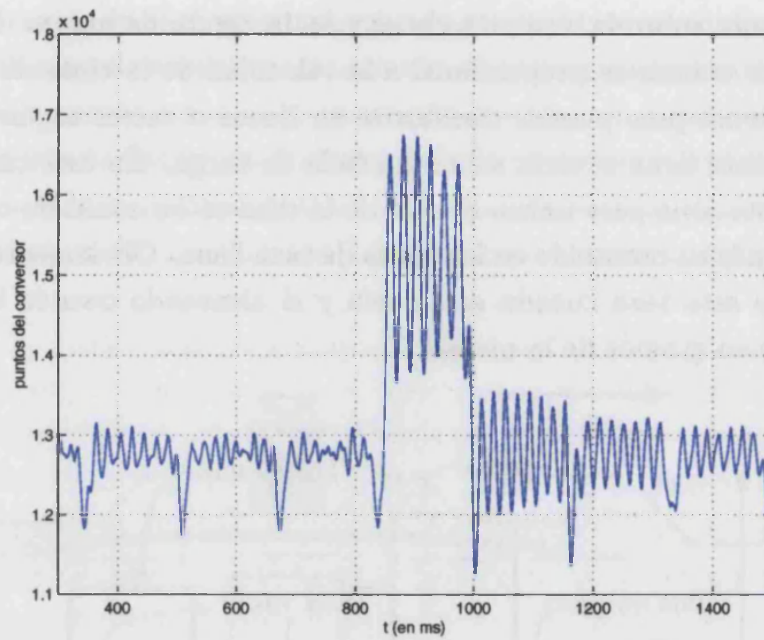
Figura 4.2: Señal ideal: Tramos *peso* y *no-peso* de tazas vacías y no vacías con diferentes pesos.

Sin embargo, el aspecto de la señal adquirida varía notablemente respecto del comportamiento ideal y depende, además, de la velocidad. La figura 4.3 muestra las señales reales obtenidas de la adquisición a 6 frutos/segundo y 20 frutos/segundo. En dichos tramos todas las tazas pasan vacías a excepción de una que pasa conteniendo, en ambos casos, un peso de 182 gr.

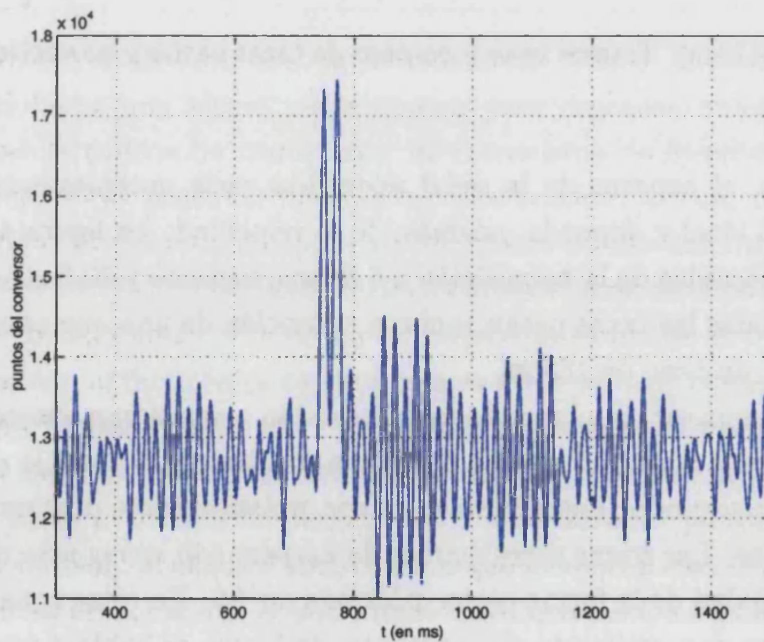
Como cabría esperar, en el mundo real no existen transiciones abruptas. Las zonas *no-peso* vienen señaladas por un pico hacia abajo entre zonas peso, lo cual se observa mejor a 6 frutas/segundo. Estas marcas no son prácticamente distinguibles para velocidades elevadas. Las zonas entre marcas de *no-peso* son *zonas peso* que también se alejan de la idealidad de la forma plana mostrada en 4.2. En estas zonas puede observarse típicamente una oscilación no constante, de la que se hablará posteriormente.

4.2 Modelización

En una primera aproximación se pensó que las oscilaciones que se producen en las *zonas peso* estarían afectadas de ruido de red, sin embargo un análisis más profundo



a)



b)

Figura 4.3: Señales adquiridas para velocidades de a) 6 frutos/segundo y b) 20 frutos/segundo.

de las señales comprobó que ésta no era la principal fuente de ruido. En esta sección vamos a centrarnos en observar la naturaleza de dichas perturbaciones.

4.2.1 Test a diferentes velocidades

El estudio espectral de la señal adquirida, demuestra que existe gran cantidad de energía en la zona de los 50Hz, más concretamente de 51 a 52Hz dependiendo de la velocidad de la cinta (Oppenheim, 1999; Stearns, 1988). Esta frecuencia es la correspondiente a las oscilaciones que aparecen en las zonas peso. Ante este dato, una primera e inevitable idea es que dicha oscilación pudiera producirse por una contaminación de ruido de red de 50Hz, deficientemente eliminado, sobre todo para un ambiente tan ruidoso como es el industrial.

Para estudiar esta posibilidad vamos a utilizar una célula de carga testigo, que es fijada a la calibradora de la misma forma que las células de carga de cada línea, pero a diferencia de éstas, la célula de carga testigo no soportará la carga de ningún peso a lo largo del test. Esta célula de carga testigo se utiliza sencillamente para captar el comportamiento en vacío cuando la calibradora funciona en clasificación a distintas velocidades.

La figura 4.4 ilustra el comportamiento captado por el sistema de adquisición. Se representan dos segundos de registros obtenidos por la célula de test con la máquina parada (en azul), la máquina funcionando a 6 frutos/segundo (en rojo), y a 20 frutos/segundo (en verde). El registro obtenido con la máquina parada es constante, basado en 50Hz y armónicos en 100Hz y 150Hz. Éste es indudablemente originado por ruido de red. Sin embargo, nótese cómo posee baja amplitud comparado con los niveles de la oscilación que afectan a las zonas peso.

Como vemos, se registra una vibración tanto mayor cuanto mayor es la velocidad de arrastre de la cinta. Para 6 frutas/segundo (figura 4.5) la calibradora está en marcha y la célula testigo capta una vibración de espectro complejo con una mayor componente en 51Hz. Para la mayor velocidad, figura 4.6, la oscilación captada por la célula testigo se incrementa, manteniéndose las características.

Nótese que existe una pequeña componente de ruido inducido, pero que puede descartarse el hecho de que sea éste el responsable del comportaminto observado en los registros tomados con la cinta en movimiento. ***Éste se debe atribuir en su mayoría a la vibración de la calibradora, ya que el nivel de ruido se hace mayor conforme aumenta la velocidad de la calibradora, factor que incrementa la vibración de la máquina.***

Los registros para la cinta en marcha parecen acercarnos a un esquema de modulación, por supuesto aleatorio, con una componente frecuencial definida. En definitiva, ***recuerdan a la salida de un proceso aleatorio de banda estrecha*** (Newland, 1997).

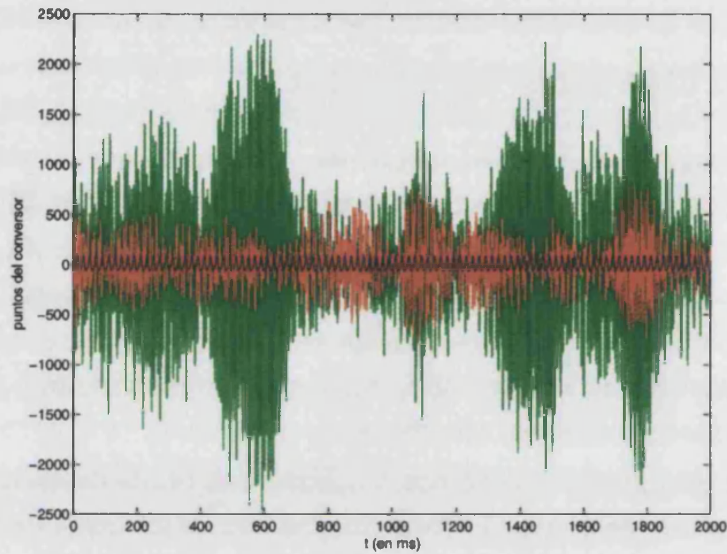


Figura 4.4: Célula testigo a 0 (azul), 6 (rojo) y 20 frutos/segundo (verde).

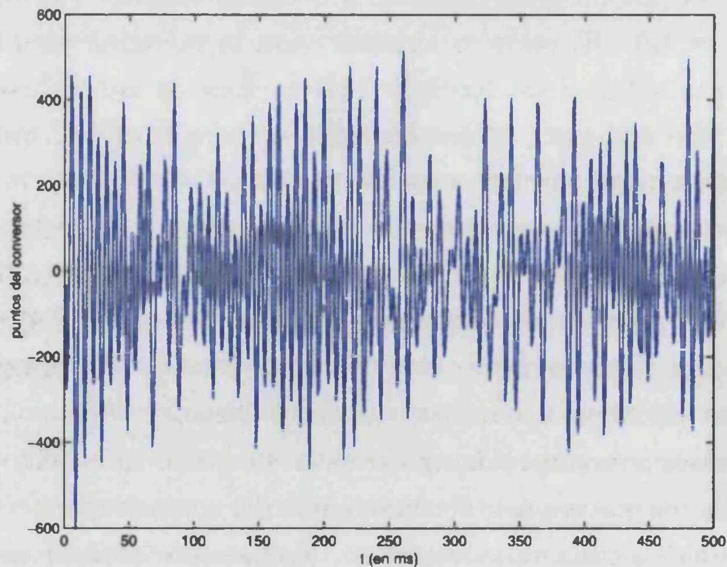


Figura 4.5: Tramo de 0.5 segundos del registro tomado por la célula testigo a 6 frutos/segundo.

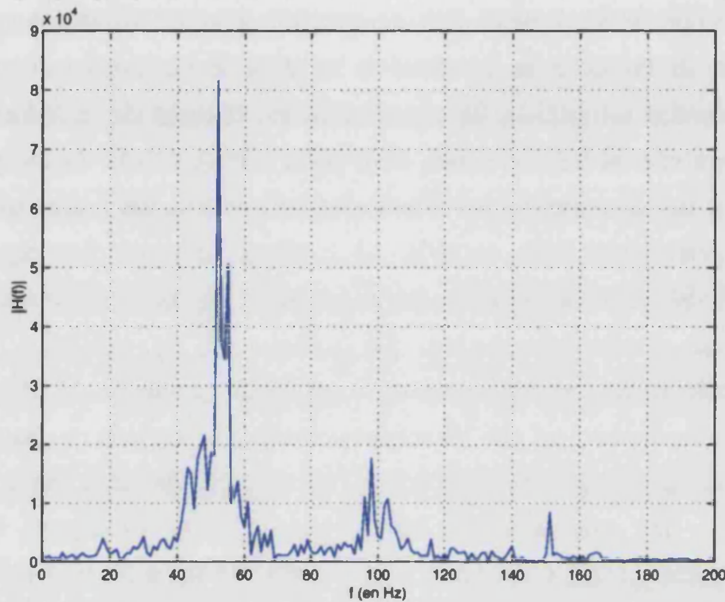


Figura 4.6: Transformada de Fourier del registro obtenido a máxima velocidad.

4.2.2 Test de la naturaleza de las oscilaciones en las *zonas peso*

Un paso más en la caracterización de las oscilaciones de las *zonas peso* es el estudio de la naturaleza de éstas, en otras palabras, debemos determinar si poseen naturaleza *aleatoria* o si siguen un proceso *determinista*, es decir, responden a ciertos patrones fijos y dependientes de las interacciones entre las tazas y la célula de carga.

Para ello se ha diseñado el siguiente experimento: Se pone en funcionamiento la calibradora para una velocidad que permanecerá constante hasta la finalización del experimento. Previamente ha sido marcada una taza, en la que será depositado vuelta tras vuelta siempre el mismo peso sobre la misma taza en la zona de entrada, que dista aproximadamente unos cinco metros de la zona de pesado, distancia suficiente para que la pesa se asiente sobre la taza. Se usa siempre la misma taza con el mismo peso para asegurar que el proceso de pesado se realiza en las mismas condiciones. El resto de las tazas permanecen vacías y existe una de ellas que se elimina de la cinta de transporte para proporcionar una referencia.

Durante el experimento se adquiere continuamente a 1KHz la señal de la célula de carga de línea, la célula de carga testigo y la de los acelerómetros fijados a la base y vástago de la célula de carga de línea.

Tanto la celula de carga testigo como los acelerómetros son mecanismos introducidos para la medición de la vibración. Sin embargo, el estudio de la señal procedente de los acelerómetros no ha sido determinante para la obtención de un modelo para

la eliminación de las oscilaciones. Posteriormente serán utilizados en el capítulo 9, donde se propone un método de generación interna de sincronismos en base a estos.

A partir de la señal adquirida, se estudiarán los tramos de la señal correspondientes a la misma taza con el mismo peso, a su paso por la célula de carga, en diferentes vueltas. Para ello, la selección de los tramos es procesada mediante un procedimiento que será descrito posteriormente, sección 4.3, realizando lo que llamaremos "troceado" y "solapamiento" de los tramos que nos interesan, los pertenecientes únicamente a la taza marcada en vueltas diferentes. El proceso de "troceado" y "solapamiento" posibilitará posteriormente la realización de estudios estadísticos del peso.

En la figura 4.7a podemos ver el solapamiento de tramos correspondientes a 16 vueltas distintas de la cinta transportadora, en el paso de taza llena con 149gr y las siguientes vacías, a una velocidad de 16 frutas/segundo. La figura 4.7b realiza una representación similar para 8 tramos con un peso de 182gr a 20 frutas/segundo. Como puede apreciarse, se siguen unos patrones similares de oscilación en la zona peso de diferentes vueltas.

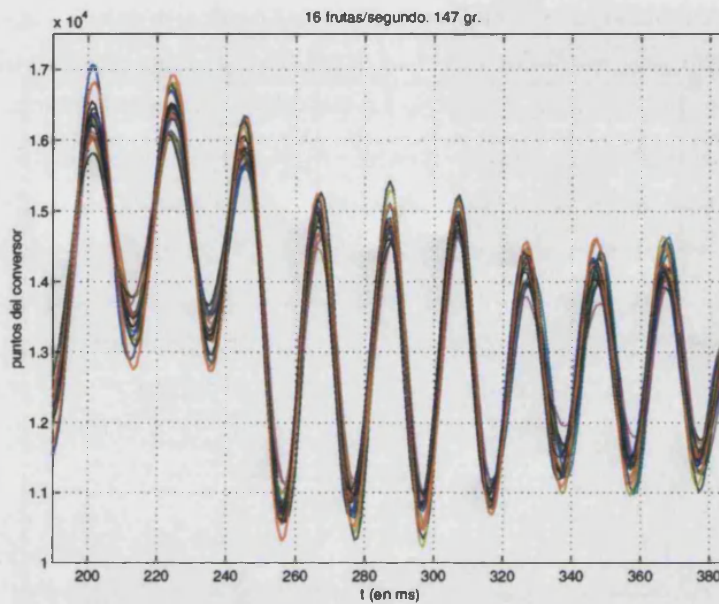
Los troceados y solapamientos de los tramos de diferentes vueltas se correlacionan, lo cual *rompe cualquier posible hipótesis sobre la aleatoriedad de las oscilaciones de las zonas peso.* No obstante, nótese que aunque se siga el mismo patrón existen *diferencias entre los tramos que, éstas sí, pueden ser asociadas a procesos aleatorios de vibración, rozamiento, etc.*

4.2.3 Hipótesis de la generación de oscilaciones

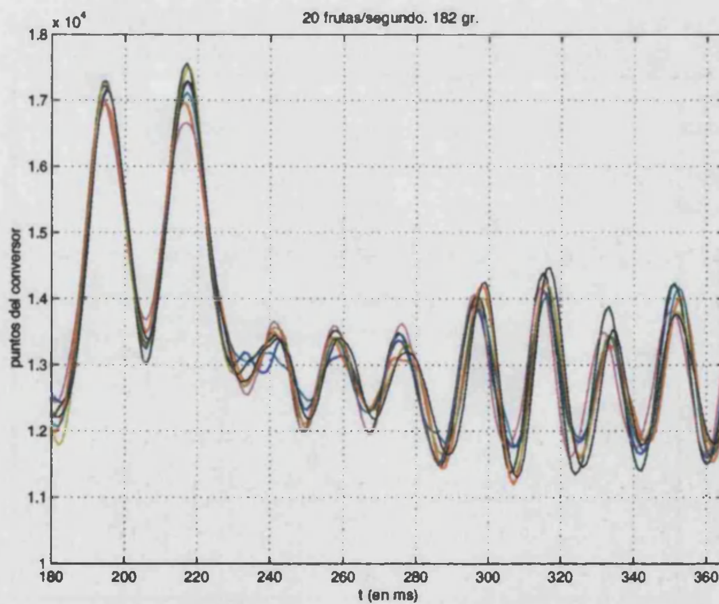
Una vez descartado el origen aleatorio de la oscilación captada, podemos decir que las oscilaciones en cuestión siguen un patrón de carácter determinista y se repiten de la misma forma en las mismas condiciones: este patrón depende del peso y de la velocidad de la cinta de arrastre. Partimos de la conjetura inicial por la que muchos tramos de la señal adquirida aparecen como oscilaciones amortiguadas que podrían asimilarse a la respuesta de un sistema de segundo, o mayor orden.

Esta idea no es gratuita, dado que la galga extensométrica impresa de la célula de carga se utiliza en un rango denominado *zona elástica*, figura 2.2. Obviamente, esto proporciona una característica propia de comportamiento de sistema de segundo orden con un rápido amortiguamiento. Cabe notar que las células de carga no suelen estar caracterizadas dinámicamente, ya que su uso dinámico, excepto en el caso que nos ocupa, no es muy común.

Nótese cómo en la figura 4.8a, las primeras oscilaciones sugieren una amortiguación de la oscilación hacia una estabilización de la salida, que finaliza con las oscilaciones volviendo a aumentar. Por otra parte, la figura 4.8b no refleja en absoluto un com-

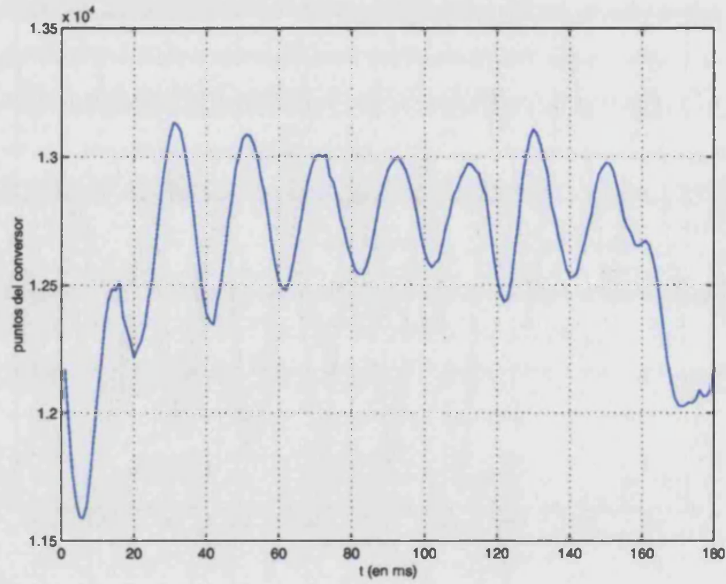


a)

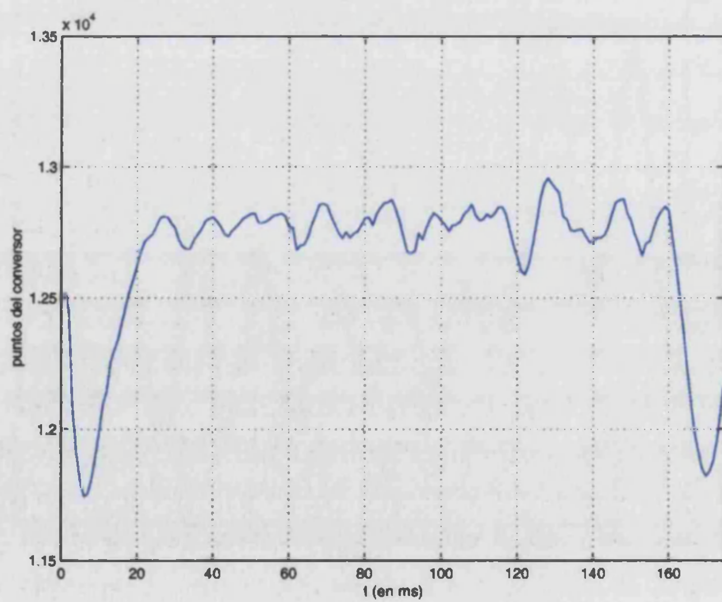


b)

Figura 4.7: Solapamiento de a) 16 tramos correspondientes a experimentos consecutivos con taza llena con 149gr a 16 frutas/segundo y las tazas siguientes, b) 8 tramos correspondientes a distintos experimentos con taza llena con 182gr a 20 frutas/segundo.



a)



b)

Figura 4.8: Distintos casos, a 6 frutos/segundo, de oscilaciones de las zonas peso.

portamiento de salida de segundo orden. Sin embargo, nótese que la señal de entrada no será la ideal sino que intervendrán además factores como:

- Vibración de la máquina.
- Rozamiento de la taza sobre el Teflon del tramo de patín aislado.
- Movimiento de la pesa sobre la taza.

Estos factores se suman a la entrada de un hipotético sistema de segundo orden, interviniendo en el mantenimiento, aumento y destrucción de las oscilaciones en 4.8b. En otras palabras, el comportamiento de segundo orden se puede observar en ciertos tramos, sin embargo la aparición de los factores de ruido listados modificaría el comportamiento ideal a formas como las ilustradas en las figuras 4.8a y 4.8b. Sin embargo, dado el determinismo del patrón seguido por las oscilaciones, los factores que más influencian deben ser los de recuperaciones de la taza sobre el patín y el rozamiento de la taza sobre el Teflon del tramo aislado, es decir, la dinámica de arrastre de uno sobre otro.

Por lo tanto partiremos de la hipótesis de que las oscilaciones se producen por un comportamiento dinámico amortiguado. Ante la entrada de una taza sobre el patín y el arrastre de ésta, más la vibración de la máquina.

4.2.4 Modelización

Para modelizar estas oscilaciones eliminando cualquier influencia de la cinta de arrastre sobre la célula de carga, podemos estudiar los tramos de referencia en los que se eliminó la taza. En la figura 4.9, vemos un segmento del comportamiento de uno de estos tramos, que es lo que podríamos denominar un 'segmento de amortiguamiento natural'. Nótese que todavía queda la influencia de la vibración por lo que no podemos observar un 'amortiguamiento natural' progresivo hasta la entrada de una nueva taza, pero si un mayor tiempo de 'amortiguación natural' en los tramos con taza eliminada.

Para modelizar el comportamiento dinámico del sistema de pesado, se ha elegido el tramo indicado en la figura 4.9, obtenido para una velocidad de 6 frutas/segundo. Este tramo posee características idénticas al de muchos otros sin taza. En adelante, asumiremos como hipótesis que dicho modelo es válido para todas las velocidades. Esta hipótesis se justificará posteriormente a partir de los resultados del capítulo 9 que validan la hipótesis formulada.

La modelización propuesta de la función de transferencia está basada en un modelo ARMA con una estructura (Ljung, 1996; Ljung, 1995):

$$A(q) \cdot y(n) = B(q) \cdot x(n-d) + C(q) \cdot e(n) \tag{4.1}$$

donde $y(n)$ es la salida del sistema, $x(n)$ el estímulo, $e(n)$ el error de predicción, $A(q)$ los coeficientes autorregresivos, $B(q)$ los coeficientes de promediado móvil, $C(q)$ la covarianza estimada y d el retraso del modelo.

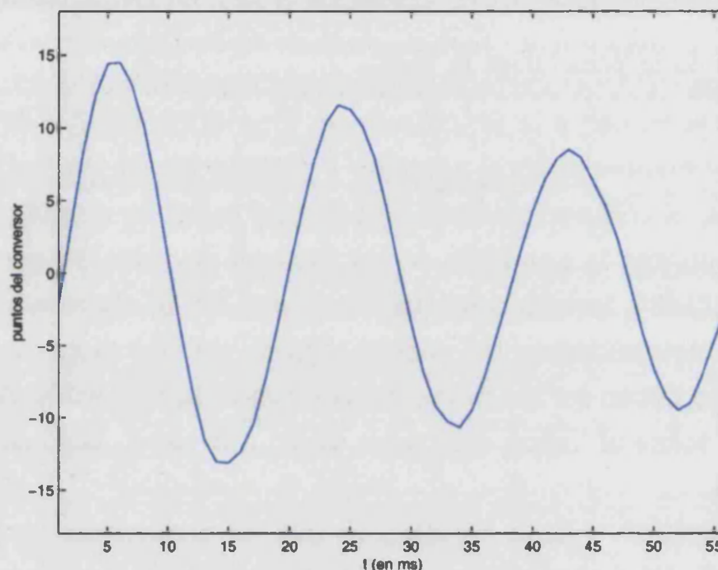


Figura 4.9: Segmento correspondiente a 50 ms, 10 ms después de que la última taza tras la referencia deje la plataforma.

El problema de la estimación a través de los modelos, sobre todo AR, está íntimamente ligado al de la predicción lineal (Makhoul, 1975) que pretende determinar una serie de coeficientes que permitan predecir las salidas futuras de un sistema a partir de las entradas (si las hubiera) y salidas anteriores.

Debido a que no se conoce, a priori, el orden del modelo, es necesario verificar el comportamiento del mismo para varios órdenes, y seleccionar algún parámetro o criterio de error que nos permita elegir el orden adecuado.

Una aproximación intuitiva sería construir modelos de orden creciente hasta detectar el mínimo de la varianza del error de predicción. Sin embargo, en la mayor parte de los procedimientos de estimación, este parámetro, ρ, se reduce al crecer el orden p. Otro análisis simple es realizar un estudio espectral del error de predicción o secuencia residual: los residuos deben ser aproximadamente blancos con lo que presentan un espectro plano (Kay, 1981; Marple, 1987).

Por tanto se propone definir una función de coste que tenga en cuenta el error producido en la estimación pero penalice el incremento del orden del modelo.

La mayor parte de los desarrollos encontrados en la bibliografía, se han realizado para modelos AR, aunque son extensibles a modelos ARMA disminuyendo ligeramente

el orden de la parte AR y seleccionando un orden similar para la MA. Los principales criterios para modelos AR son tres: el FPE, el AIC y el MDL.

Akaike propuso dos criterios, el primero (1969) es el "Final Prediction Error" (FPE) que se basa en minimizar la varianza del error medio de la predicción de orden 1, ya que considera que esta varianza es la suma de la potencia de la parte no predecible del proceso y una cantidad que representa las inexactitudes del cálculo de los parámetros AR, de forma que:

$$FPE\,(p) = \hat{\rho}_p \left[\frac{N + (p+1)}{N - (p+1)}\right] \tag{4.2}$$

donde N es el número de muestras, p el orden y $\hat{\rho}$ la estimación de la varianza del ruido blanco (aunque normalmente se emplee la varianza del error del predictor lineal). Este método resulta adecuado para procesos AR puros pero tiende a subestimar el orden en procesos reales.

Akaike sugirió en 1974 otro método basado en una aproximación de máxima similitud, el "Akaike Information Criterion" (AIC), que minimiza una función teórica de información:

$$AIC\,(p) = NLN\,(\hat{\rho}_p) + 2p \tag{4.3}$$

donde el término 2p representa una penalización por el empleo de coeficientes AR extra que no den lugar a una reducción sustancial de la varianza del error de predicción. Cuando N $\rightarrow \infty$, ambos criterios son asintóticamente equivalentes.

Tras demostrarse la inconsistencia del AIC como estimador, Rissanen, en 1983, desarrolló una variante denominada "Minimum Descriptor Length":

$$MDL\,(p) = NLN\,(\hat{\rho}_p) + pln\,(N) \tag{4.4}$$

que es estadísticamente consistente y suele generar los órdenes más bajos.

Hay muy pocos estudios estadísticos teóricos sobre la estimación espectral ARMA de registros reales finitos. Como prueba de ello, sólo existe un criterio de selección del orden del modelo bastante aceptado, el propuesto por Cadzow en 1982:

$$AIC\,(p,q) = ln\,\hat{\sigma}^2_{\omega pq} + \frac{2\,(p+q)}{N} \tag{4.5}$$

El análisis del orden del modelo realizado con 20 señales de prueba tomadas a 6 frutos/segundo, para los 4 algoritmos mencionados dieron como resultado la siguiente tabla.

Método	Numerador	Denominador
FPE	-	3
AIC (AR)	-	3
MDL	-	2
AIC (ARMA)	2	2

Tabla 4.1: Resultados de estimación de diferentes criterios.

Con lo cual se optó por considerar un orden $(2, 2)$, lo cual coincide con la hipótesis previa de que se trata de un fenómeno de amortiguamiento natural de un sistema elástico.

Los coeficientes del modelo ARMA se obtienen a partir del algoritmo iterativo de Gauss-Newton, que se usa para minimizar el criterio de predicción del error (Matlab, 1997; Matlab, 1993). La función de transferencia obtenida es:

$$H(z) = \frac{(1 + 0.0038 \cdot z^{-1} - 0.3784 \cdot z^{-2})}{(1 - 1.862 \cdot z^{-1} + 0.9763 \cdot z^{-2})} \tag{4.6}$$

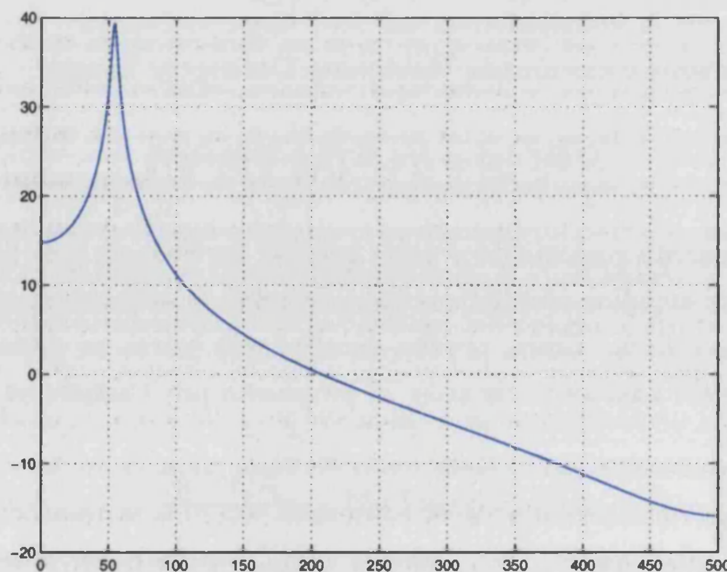


Figura 4.10: Respuesta en frecuencia de la función de transferencia del modelo.

La respuesta en frecuencia resultante, figura 4.10, muestra un sistema caracterizado por una resonancia centrada en 54Hz. La figura 4.11 ilustra el diagrama de polos y ceros de la función de transferencia 4.6.

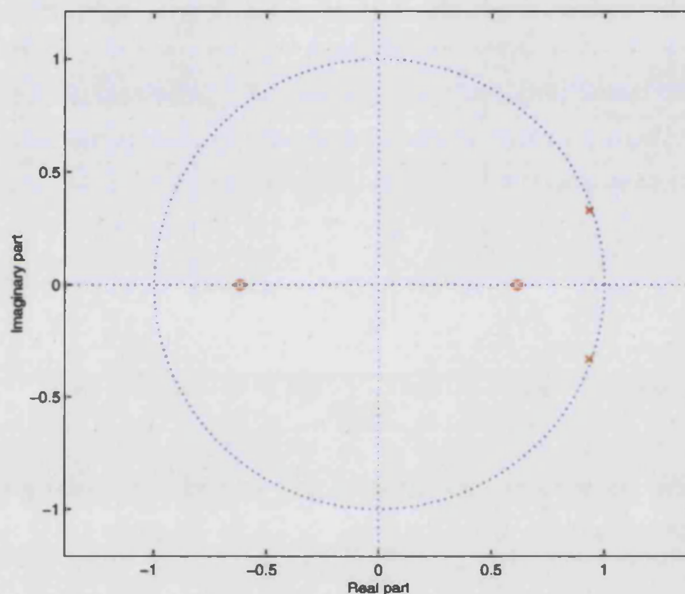


Figura 4.11: Diagrama de polos y ceros del modelo.

4.2.5 Justificación del ruido de la célula de test

Al modelizar, como hemos hecho, la respuesta dinámica del sistema podemos ver cómo este proceso tiene una respuesta en frecuencia característica de banda estrecha. Para comprobar, en una primera aproximación, la idoneidad de este modelo, vamos a intentar predecir y justificar los resultados obtenidos con la célula testigo, cuya salida proviene únicamente de una entrada: el peso del vástago de la célula de carga más la vibración asociada a éste.

Asumiendo la vibración como un proceso aleatorio gaussiano, la figura 4.12 presenta el resultado de una secuencia aleatoria gaussiana (Krauss, 1994). Como podemos ver, el resultado del modelo es una oscilación similar a la obtenida por la célula testigo, y ambas con una marcada base frecuencial de 50Hz.

La hipótesis de que la salida de la célula de la carga testigo es la salida de un proceso aleatorio se apoya en que tramos solapados de la célula testigo, paralelos a aquéllos de la célula de línea que presentan un patrón similar, no se correlacionan, confirmando la aleatoriedad del proceso debido a la vibración de la máquina.

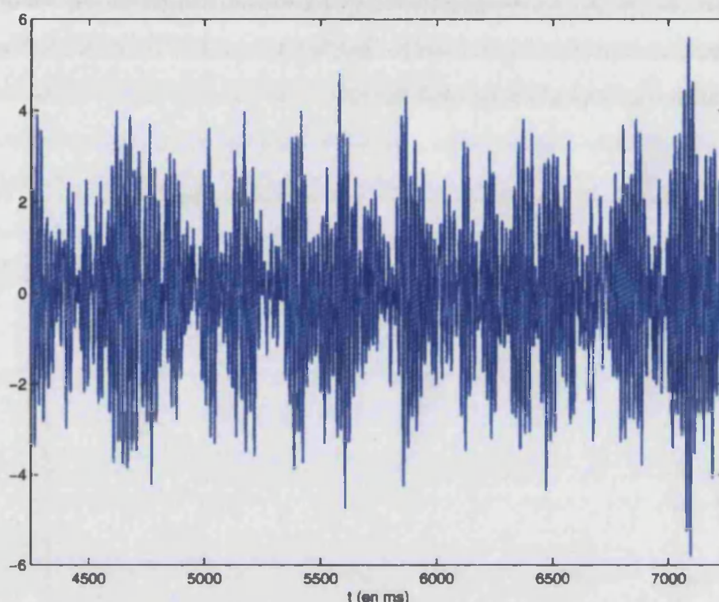


Figura 4.12: Salida del sistema modelado ante una entrada de ruido gaussiano. Proceso aleatorio de banda estrecha.

4.3 Acondicionamiento para el estudio estadístico

En el procedimiento de obtención del valor escalar del peso para cada una de las tazas a partir de la señal adquirida, existen dos pasos clave: por una parte el preprocesado de la señal y por otra el algoritmo de estimación del peso a partir de la señal preprocesada. Posteriormente, se discutirán cada uno de estos dos pasos, sin embargo para cualquier combinación de preprocesado y algoritmo de obtención del peso, es necesario realizar repetidos experimentos en las mismas condiciones para desarrollar un estudio estadístico de los resultados. Este necesario estudio estadístico permitirá la validación de los algoritmos utilizados y de la descripción de la precisión que con ellos se puede obtener.

Nótese que cada conjunto de experimentos queda registrado en uno o varios ficheros de adquisición. De esta forma, los experimentos quedan distribuidos en ciertos *tramos de interés* de la adquisición a lo largo de uno o varios registros (dado que un experimento suele utilizar siempre la misma taza para mantener controladas y similares las condiciones del experimento). Con ello, generamos una gran cantidad de información que necesitamos procesar para poder quedarnos, en cada momento, únicamente con la que nos interesa. Un ejemplo de ello sería la obtención de un vector de tramos de la taza número 170 (numerada a partir de la referencia) de cualesquiera de los registros que pertenecen al experimento, de esta forma podríamos extraer la información

y concentrarnos únicamente en los tramos del experimento para, por ejemplo, poder aplicar localmente los algoritmos de estimación del peso.

Este proceso, denominado a partir de ahora *troceado*, se divide en marcado, troceado y solapado de la señal preprocesada. Cada uno de estos pasos se describe a continuación. Este proceso ha sido aplicado para la valoración de todos los algoritmos y procesos estudiados.

4.3.1 Marcado

El marcado es el primer paso para la realización del proceso del *troceado.* Toma como parámetros el registro a marcar y la velocidad del variador (medida indirecta y aproximada de la velocidad en frutas/segundo de la cinta de arrastre). Así, se toma la señal de entrada, preprocesada o no, y se marcan los puntos, figura 4.13, en los que cada taza comienza a salir del patín de pesado de la célula de carga. Esto se realiza mediante procesado digital, y las marcas obtenidas son equivalentes a los sincronismos hardware enviados por el encoder incremental del módulo de encoder de la calibradora (sección 5.4.6).

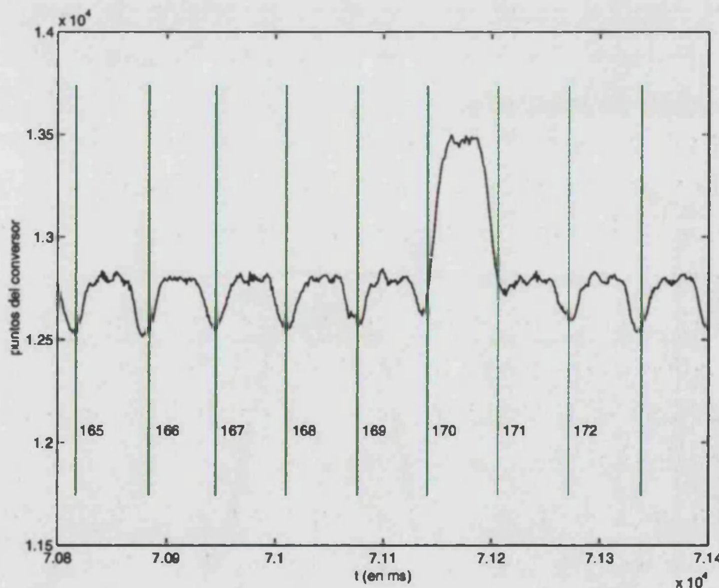


Figura 4.13: Ilustración del marcado y troceado.

La figura 4.2 ilustra el marcado de un registro tomado a 16 frutos/segundo, que forma parte de un experimento en el que se carga la taza número 170 con una pesa de 49gr. Nótese cómo las marcas verdes indican una precisa separación de la señal correspondiente a cada taza.

4.3.2 Troceado

Una vez obtenidas las marcas de la señal, el troceado detecta el paso de la señal de referencia y obtiene el número de taza correspondiente a cada intervalo de señal entre dos marcas. El proceso de troceado realiza, además, una comprobación de la partición realizada por el marcado. Este paso devuelve como salida un conjunto de códigos de error y un vector de números de taza de la misma longitud que el vector de marcas.

4.3.3 Solapado

Como último paso, para extraer los tramos de interés de una determinada taza, se utiliza conjuntamente el vector de números de taza y de marcas, que permite identificar la localización de éstos, posibilitando que puedan tomarse todos los tramos correspondientes a la misma taza en las distintas vueltas de la cinta. El resultado del marcado demuestra que el posicionamiento de la marca oscila en ± 2 muestras alrededor del posicionamiento que realizaría el sincronismo hardware, considerando la correlación de los tramos y asumiendo que los patrones de oscilación son deterministas.

El proceso de solapado finaliza con un alineamiento, por correlación, de los tramos seleccionados para eliminar cualquier pequeña imprecisión del posicionamiento de la marca. Una vez alineados, los tramos se almacenan formando filas de una matriz, salida final del proceso de solapado.

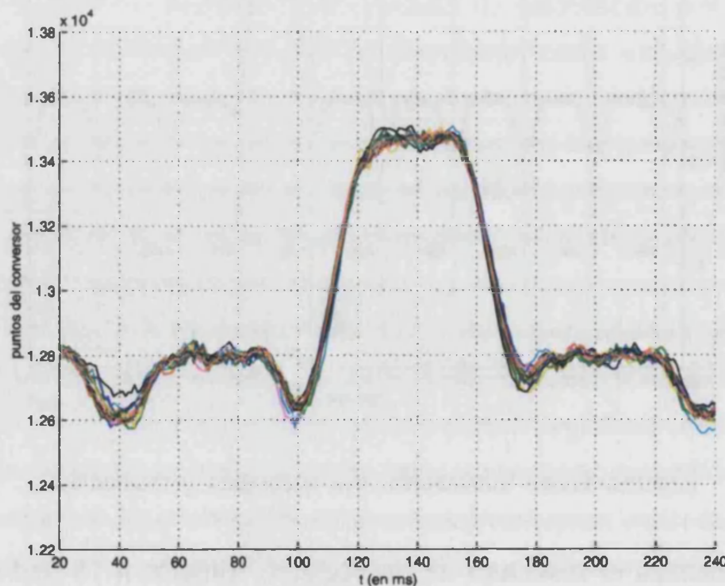


Figura 4.14: Varios tramos correspondientes a distintos casos del mismo experimento.

4.4 Preprocesado de la señal

El primer paso para el tratamiento de la señal es acondicionarla para eliminar, en la medida de lo posible, las oscilaciones que contiene, sobre todo, en las *zonas peso.*

Para el acondicionamiento de la señal se plantean diferentes algoritmos: un sencillo filtrado de 50 Hz basado en promediado, deconvolución mediante el modelo ARMA y diferentes sistemas adaptativos. A continuación se presenta una introducción para cada uno de ellos. En el capítulo 9 se establecerá la selección del algoritmo de preprocesado y los resultados obtenidos (Proakis, 1998; Ifechor, 1997).

4.4.1 Filtrado

La primera de las aproximaciones al preprocesado de la señal, es la aplicación directa de un promediador de orden 20. La figura 4.15 ilustra la respuesta en frecuencia de este filtrado de fase lineal, básicamente un filtrado paso-bajo con ceros en 50Hz y sus múltiplos. De esta forma se ensaya la posible eliminación de los 50Hz de la oscilación característica de la señal y sus armónicos (Parks, 1987).

$$H(z) = \frac{1}{20}\sum_{k=0}^{19} z^{-k} = \frac{1}{20}\frac{1-z^{-20}}{1-z^{-1}} \tag{4.7}$$

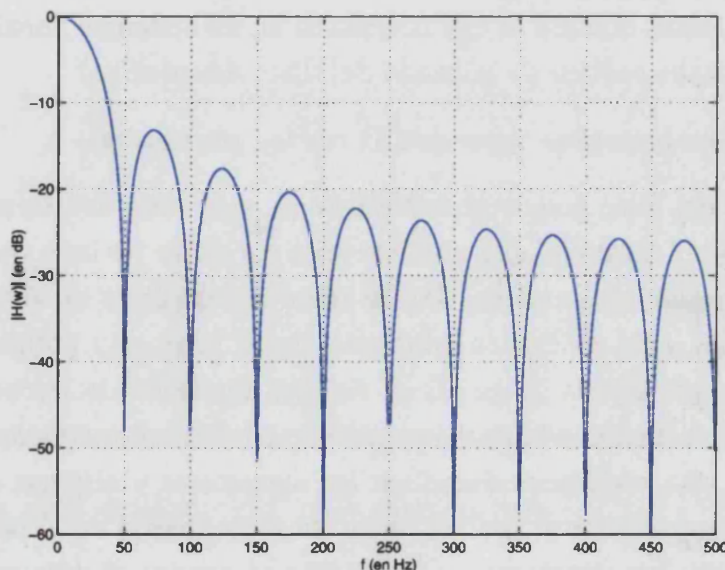


Figura 4.15: Respuesta en frecuencia del promediado de orden 20.

4.4.2 Deconvolución mediante el modelo ARMA

Una segunda aproximación al problema es el uso del modelo ARMA obtenido en la sección 4.2.4. Mediante este método se usa la función de transferencia inversa de la expresión 4.6 para estimar la entrada al sistema a partir de la salida, es decir, de la señal adquirida. Esto resulta un filtro de fase mínima, cuyo retardo de grupo se ilustra en la figura 4.16. La información de la señal se encuentra básicamente hasta 40Hz, por lo que sufriría un retardo de hasta 2 muestras, que no supone una distorsión apreciable del frente de onda y, por tanto, no altera la estimación del peso.

Tras la deconvolución se realiza un promediado de orden 20, expresión 4.7. La figura 4.17 muestra la función de transferencia del preprocesado completo.

4.4.3 Sistemas adaptativos

Los sistemas adaptativos han sido de gran utilidad desde su introducción debida a Bernard Widrow y Marcian Hoff (Haykin, 1996). Se han empleado en un gran número de aplicaciones prácticas que van desde la ecualización de canales hasta la modelización de funciones de transferencia pasando por la cancelación adaptativa de ruido (Clarkson, 1993). Estos sistemas se caracterizan por:

- *Los sistemas adaptativos no son invariante temporales.* Este hecho conduce de forma inmediata a que herramientas de análisis como transformadas Z y de Fourier, principalmente, no son aplicables en este tipo de sistemas.

- *Los sistemas adaptativos no son lineales.* Este hecho conduce a la aparición de comportamientos dinámicos que no existen en los sistemas lineales (por ejemplo comportamiento caótico de la salida del filtro adaptativo).

- *Los sistemas adaptativos dependen de ciertas constantes.*

La obtención del peso puede simplificarse (o más bien sobresimplificarse) a la determinación de un nivel DC que aparece para un corto periodo de tiempo dependiendo de la velocidad de la cadena. Por lo tanto necesitamos un sistema que estime el promedio de una señal en ciertos intervalos, específicamente podríamos concebirlo como un sistema que estime la amplitud de una sinusoide de frecuencia cero. Tal sistema puede ser realizado adaptativamente siguiendo el esquema de la figura 4.18.

A lo largo de esta sección se describen los algoritmos y algunas de las variantes que han sido utilizadas para el análisis de su funcionamiento en el problema que nos ocupa. Básicamente, los algoritmos considerados se pueden dividir en:

- **LMS** *(Least Mean Square).* Algoritmo clásico en sistemas adaptativos. Está basado en la minimización del error cuadrático medio en cada muestra.

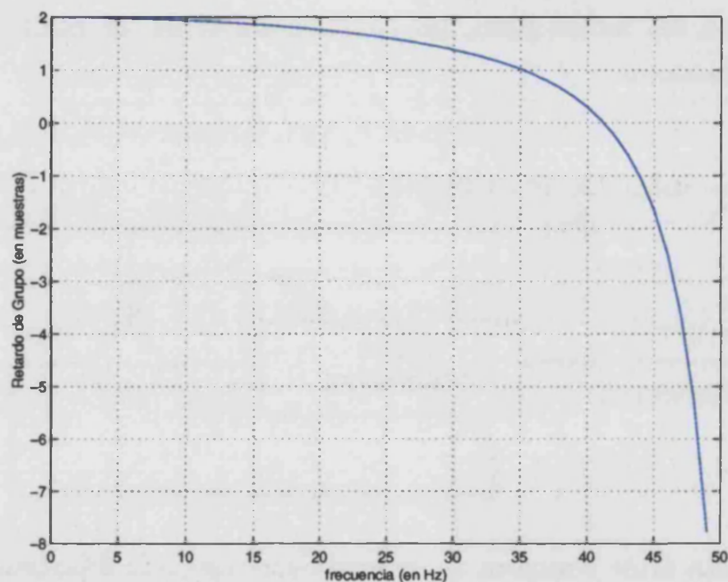


Figura 4.16: Retardo de grupo de la función de transferencia inversa del modelo.

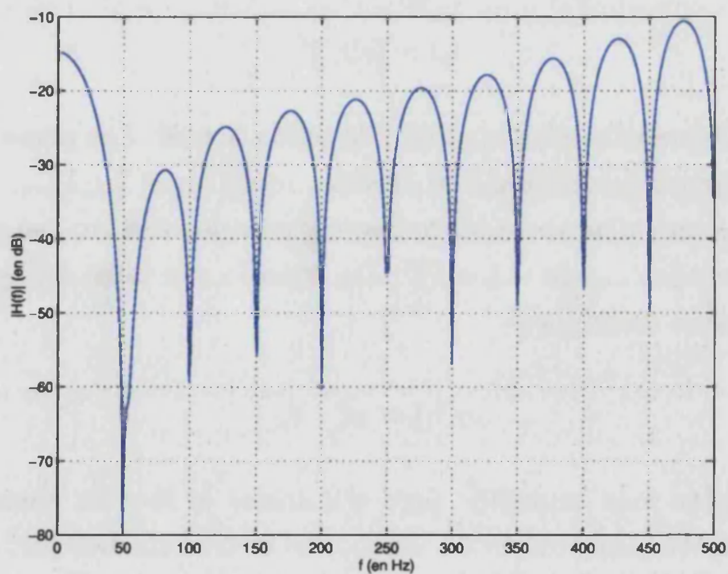


Figura 4.17: Preprocesado completo mediante deconvolución del modelo ARMA y filtro promediador.

- **RLS** *(Recursive Least Squares).* En lugar de minimizar el error instantáneo, este algoritmo realiza un promediado exponencial de todos los errores pasados. Dependiendo del factor peso, las contribuciones de los errores pasados serán mayores o menores.

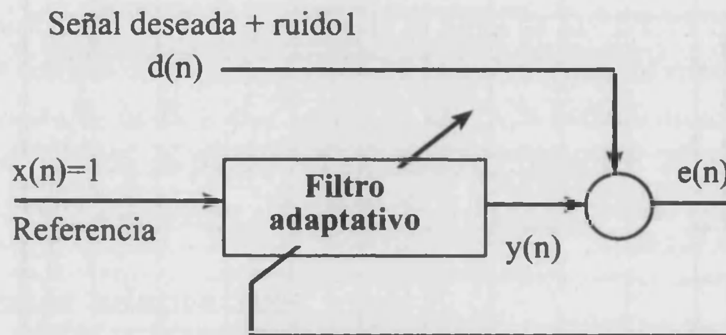


Figura 4.18: Esquema de un cancelador de ruido adaptativo.

4.4.3.1 El algoritmo básico y sus variantes

Los sistemas adaptativos pretenden minimizar una función de coste. La función de coste más comúnmente usada es la señal, J, del error cuadrático:

$$J = [e(n)]^2 \tag{4.8}$$

donde e(n) es la diferencia entre la señal y la salida actual. Los pesos de los filtros en un instante de tiempo determinado se denotan como $w_n = [w(1), \ldots.., w(L)]^t$ (donde t indica transposición) y las entradas presente y pasadas al filtro son agrupadas como un vector $x_n = [x(n), \ldots..., x(n-L+1)]^t$. De acuerdo con estas definiciones, la salida del filtro, $y(n)$, viene dada por:

$$y(n) = w_n^t \cdot x_n \tag{4.9}$$

El procedimiento más extendido para minimizar la función coste J es conocida como *norma delta.* La justificación de esta norma es directa: Los coeficientes del filtro en un instante n deberían ser obtenidos de muestras previas más una cierta cantidad que aproxima al mínimo. Esta cantidad es el gradiente de la función definida en 4.8 con un cambio en el signo (Haykin, 1996):

$$w_{n+1} = w_n - \alpha \cdot \nabla J \tag{4.10}$$

donde α es un parámetro fijo conocido como constante de adaptación. Usando las ecuaciones 4.8 a 4.10 y calculando el gradiente, la expresión obtenida para actualizar el vector de coeficientes es:

$$w_{n+1} = w_n + \alpha \cdot e(n) \cdot x_n \tag{4.11}$$

En nuestro sistema en particular, la entrada es constante e igual a 1 por lo que tenemos un peso sencillo para actualizar y la expresión se reduce a:

$$w_{n+1} = w_n + \alpha \cdot (d(n) - w_n) \tag{4.12}$$

Aplicando la recursión definida en 4.11 la expresión de actualización para la obtención del peso en la muestra *n+1* puede ser escrita como:

$$w_{n+1} = \alpha \cdot \sum_{k=0}^{n} (1-\alpha)^k \cdot d(n-k) \tag{4.13}$$

Por lo tanto, el resultado del filtro es un promediado exponencial de la entrada, con lo que el sistema se comporta como un filtro paso-bajo. Debe destacarse la baja complejidad computacional por la utilización de un único peso para actualizar y filtrar.

Como veremos posteriormente, el algoritmo LMS básico no proporciona buenas prestaciones para la disminución del "ruido", por lo que han sido utilizadas diferentes variantes para intentar mejorar los resultados. Estas variantes fueron seleccionadas de acuerdo a dos criterios: un grupo de ellas realizan algún tipo de promediado para reducir ruido y otras incrementan la velocidad de convergencia para reducir los tiempos de transición entre *zonas peso* a causa del reducido número de muestras disponibles en cada una de ellas (Clarkson, 1993):

- *LMS promediado* (ALMS).
- *Variante de mediana* (MLMS).
- *Variantes rápidas.* Concebidas para incrementar la rapidez de convergencia del algoritmo LMS básico. Han sido consideradas dos variantes:
  - *Variante del momento.* Consiste en actualizar los pesos en el instante n considerando la magnitud del cambio en los pesos de muestras previas.
  - *Variante que explota el lapso temporal entre muestras* (Soria, 1997). Se propone emplear el lapso intermuestral para optimizar los pesos de los filtros. Así,intercalando otro algoritmo entre iteraciones, se incrementa la velocidad de convergencia de un sistema adaptativo basado en LMS. El

algoritmo propuesto para realizarse entre iteraciones pertenece a la familia de los de gradiente conjugado. Este tipo de algoritmos destacan por su gran velocidad de convergencia con un coste computacional moderado. El algoritmo escogido es el de Fletcher-Reeves (Soria, 1998).

A continuación se describen más precisamente las dos variantes más interesantes.

La Variante ALMS

El ALMS propone la siguiente actualización para el peso:

$$w_{n+1} = w_n + \alpha \cdot \frac{1}{N} \cdot \sum_{k=0}^{N-1} e(n-k) \cdot x_{n-k} \tag{4.14}$$

El punto distintivo para esta variante es que filtra el gradiente, reduciendo un posible ruido gaussiano en la señal deseada del filtro adaptativo (la entrada está fijada a 1). De acuerdo a las condiciones de nuestro problema la expresión puede ser escrita como:

$$w_{n+1} = w_n + \alpha \cdot \frac{1}{N} \cdot \sum_{k=0}^{N-1} (d(n-k) - w_{n-k}) \tag{4.15}$$

Si aplicamos la transformada Z a la ecuación 4.15 la expresión obtenida es

$$H(z) = \frac{W(z)}{D(z)} = \frac{\frac{\alpha}{N} \cdot z^{-1} \cdot (1 - z^{-N})}{(1 - z^{-1})^2 + \frac{\alpha}{N} \cdot z^{-1} \cdot (1 - z^{-N})} \tag{4.16}$$

Está claro que, dependiendo de la profundidad del promedio y la constante de adaptación, se obtienen diferentes funciones de transferencia paso-bajo.

La variante MLMS

Muy similar al LMS, pero en lugar de considerar el promediado de los gradientes más recientes, se toma la mediana. Esta variante es deseable para eliminar el ruido impulsional (Clarkson, 1993).

Dadas las características dinámicas de la célula de carga, la alternancia sucesiva de taza y no-taza provoca oscilaciones cuando una nueva taza entra en la área de pesado. Para evitar esto, se ensaya el MLMS, considerando la mediana de los gradientes más recientes. En este caso, la actualización del peso es:

$$w_{n+1} = w_n + \alpha \cdot mediana(d(n-k) - w_{n-k}) \quad 0 \leq k \leq N \tag{4.17}$$

4.4.3.2 Los algoritmos recursivos (RLS)

Una vez considerados el LMS estándar y algunas de sus variantes, una diferente aproximación al problema ha sido la aplicación de una alternativa: *Los sistemas recursivos.* Para éstos, la función de coste a ser minimizada es la definida por (4.18) (Haykin, 1996):

$$J = \sum_{k=0}^{n-1} \lambda^k \cdot e^2(n-k) \tag{4.18}$$

Una de las principales diferencias es que la función de coste no es el error instantáneo sino un promedio sopesado de todos los errores previos. El parámetro de escalado (λ) es conocido como "memoria" del sistema. La minimización de la función de coste nos lleva a las ecuaciones del filtro adaptativo RLS (Recursive Least Square) (Haykin, 1996).

Capítulo 5

Sistema de calibración de alta velocidad

5.1 Introducción

El presente capítulo describe los elementos y la operación de un sistema calibrador de alta velocidad que es fabricado por la empresa MAXFRUT S.L. de Alzira (Valencia), que ha sido desarrollado conjuntamente por ésta, la empresa DISMUNTEL S.A.L. de Algemesí y la Universidad de Valencia a través del proyecto "Contrato de asesoramiento y asistencia técnica en el área de control distribuido, comunicaciones CAN y desarrollo de estrategias de pesada", dirigido por Dr. D. J. Calpe Maravilla y con el proyecto cofinanciado con Fondos FEDER "Desarrollo de subsistemas inteligentes para el tratamiento de señales procedentes de sensores de peso. Aplicación al control distribuido de calidad en frutos" también dirigido por Dr. D J. Calpe. El desarrollo del sistema de análisis visual de la fruta ha sido desarrollado por el Departamento de Informática de la Universidad Jaume I de Castellón.

El sistema calibrador de alta velocidad se describe en este capítulo a modo de introducción para adquirir una cierta familiarización con el modo de funcionamiento y los elementos funcionales. La descripción del subsistema desarrollado por la Universidad de Valencia ocupará los tres capítulos siguientes, ocupándose del hardware, el microkernel de tiempo real creado para gestionar el sistema embebido, y la aplicación en sí. Este subsistema es el módulo de pesado, un sistema de altas prestaciones para la obtención de pesos a alta velocidad.

A través de los siguientes capítulos se ofrecerá una información lo suficientemente profunda del trabajo realizado sin llegar a ofertar piezas completas de código, es-

quemáticos o PCBs. Ésto se debe al compromiso obvio de privacidad con la empresa que comercializa el producto.

5.2 Sistemas calibradores

Los sistemas calibradores son dispositivos con la función de clasificar los diferentes productos hortofrutícolas a partir de sus características físicas. La clasificación de dichos productos en diferentes categorías viene impuesta por la necesidad de distinguir entre las diferentes calidades de los productos hortofrutícolas previo a su distribución en el mercado.

Por regla general, en dichos sistemas se distingue un dispositivo mecánico de transporte de los productos y una electrónica de extracción de las características de los productos, clasificación y control del proceso. El transporte de la fruta se consigue mediante una o varias cadenas en paralelo unidas mediante un engranaje a un motor de alterna. A la cadena van fijados una serie de receptáculos denominados tazas con el objeto de alojar en su interior la fruta. Las tazas pueden ser de diferentes tamaños según el peso del fruto a transportar.

La estructura de una maquina calibradora de frutos, figura 5.1, básicamente consta de tres zonas: alimentación, clasificación y vaciado.

- **Zona de alimentación de línea**: En esta zona los frutos son arrastrados por una cinta transportadora hasta ser depositados sobre las líneas del calibrador. Cada línea está compuesta por una cadena cerrada, unida a un motor de alterna mediante un engranaje. A la cadena van fijados en toda su extensión una serie de receptáculos de unos 10 cm de diámetro que se denominan 'tazas'. Pueden existir varias líneas en paralelo, que se desplazan a lo largo de las zonas de clasificación y vaciado. Mediante un dispositivo como un *encoder* o un detector inductivo se genera un tren de pulsos que permite a la electrónica computar el paso de las tazas.

- **Zona de clasificación**: Sobre esta región del calibrador se encuentran los sensores que miden las propiedades de los frutos. Los elementos sensores más habituales son células de carga para el peso y cámaras para el tamaño y color. Cuando se realiza calibración por color y/o tamaño, esta zona presenta unos mecanismos que se acoplan a las tazas y provocan la rotación de los frutos sobre ellas, de forma que se pueden exponer a la cámara la totalidad de la superficie del fruto desde distintos ángulos.

- **Zona de vaciado**: Sobre esta zona se encuentran las salidas y los mecanismos

que al accionarse, y dependiendo del planteamiento de la cadena, vuelcan o expulsan los frutos a las salidas. Cada salida está constituida por una cinta transportadora situada debajo de la cadena de tazas. Existe una salida por defecto al final de la máquina denominada 'retorno' que devuelve los frutos al principio del calibrador.

Los frutos entran en la máquina por la zona de alimentación, que los distribuye lo más uniformemente posible por las líneas. Una vez que un fruto se encuentra en la zona de clasificación los sensores obtienen información sobre él y se la pasan a un sistema microprocesador. Este sistema determina el valor de la variable sensada mediante algoritmos que pueden implicar el procesado de muchas muestras de peso, imágenes tomadas bajo distintos ángulos, etc., dependiendo de la naturaleza del sensor. Para el correcto funcionamiento de los sensores es necesario que éstos se encuentren sincronizados con el paso de las tazas.

Dada la complejidad del sistema se apuesta por un control distribuido en base a una serie de módulos inteligentes. Estos módulos son básicamente sistemas microprocesadores que procesan la información del fruto y pasan el valor resultante del procesado a otro sistema que ejecuta un programa de clasificación. El programa de clasificación define un conjunto de clases de fruto, cada una de las cuales está caracterizada por unos límites de las variables sensadas. Este programa asigna una clase a cada fruto y mediante una tabla clase/salida le asigna también una salida. La clase del fruto se conserva a efectos de estadísticas y la relación taza-salida es utilizada por la electrónica de la zona de vaciado. Cuando la taza que lleva un fruto clasificado se encuentra sobre la salida que se le ha asignado, la electrónica acciona el mecanismo que la vuelca. Puede darse el caso de que se desee volcar varias clases de frutos sobre una misma salida en una determinada proporción o de que una misma clase de fruta se vuelque por varias salidas, también en una proporción definida. Todo el procesado descrito ha de realizarse a velocidades de hasta 20 frutos por segundo por cada línea, lo que hace necesaria una electrónica, un sistema de comunicaciones y una programación de elevadas prestaciones.

Además de los elementos descritos, un calibrador puede contar con la siguiente maquinaria:

- **Volcadora de *pallets***: Toma las cajas de los *pallets* y vuelca su contenido en la máquina.

- **Lavadora**: Lava los frutos que pasan a su través sobre una cinta transportadora, con la finalidad de eliminar suciedad que pueda producir errores en la clasificación.

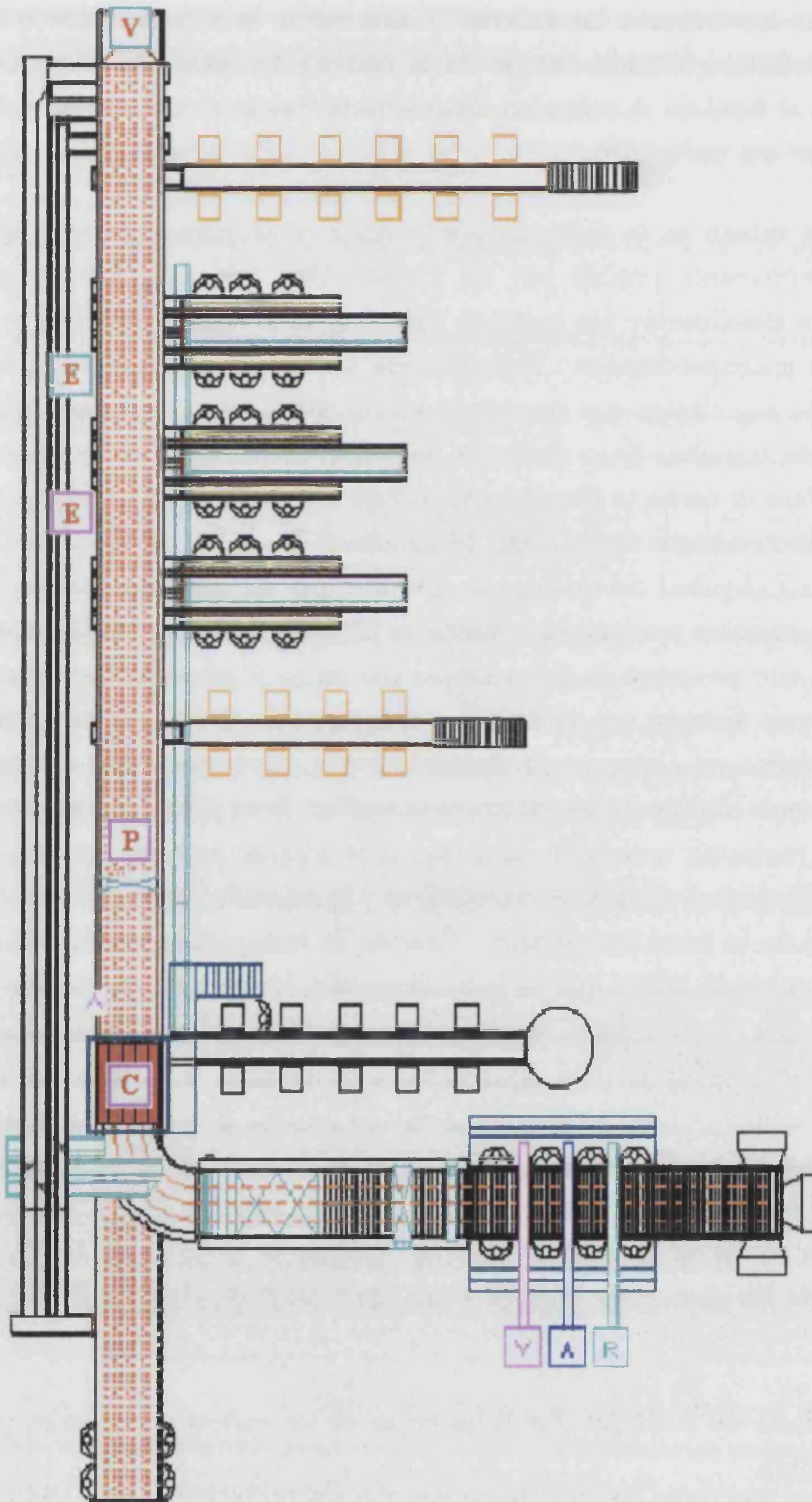


Figura 5.1: Esquema de una máquina calibradora.

- **Secadora**: Realiza el secado de los frutos tras su lavado.

- **Mesa de clasificación**: Para una primera tria manual por parte de los operarios para eliminar hojas y piezas defectuosas.

- **Pulmón**: Almacena los frutos temporalmente. Se emplea para no saturar un módulo que procesa un flujo de frutos inferior al que suministra el módulo anterior.

- **Pesadora**: Confecciona cajas de frutos con un peso determinado. Para ello pesa conjuntos de frutos a la salida del calibrador y escoge la combinación óptima.

- **Confeccionadora de *pallets***. Apila y empaqueta las cajas confeccionadas.

Son necesarios sistemas que coordinen todos estos elementos disjuntos, sobre todo a medida que aumenta el tamaño y la necesidad de automatización de las instalaciones. Existen soluciones parciales, como el uso de autómatas programables, que se emplean para realizar el arranque por condenas de los motores de alterna. Este proyecto nació con el objetivo de facilitar e integrar en una única solución el control, mantenimiento y configuración de estos elementos. Ésto supone una reducción de los costes de funcionamiento de las instalaciones así como de sus posibles ampliaciones.

5.3 Objetivos

La velocidad que puede alcanzar una mecánica avanzada en un calibrador (denominada mecánica de cadena inteligente) es de hasta 16 frutas por segundo y línea. La electrónica propuesta parte de las siguientes especificaciones:

- Velocidad de hasta 20 frutos por segundo.

- Máximo de 10 líneas de transporte de fruta.

- Máximo de 64 salidas de frutas.

- Máximo de 1350 tazas por línea.

Para un calibrador de las características que se han propuesto, se ha decantado por una electrónica de control distribuido en diferentes módulos inteligentes conectados por 2 canales de comunicación digitales. Cada módulo tendrá el acceso a una serie de sensores y/o actuadores para la realización de una serie de tareas.

5.4 Introducción al sistema MAXSORTER de clasificación de productos hortofrutícolas

5.4.1 Elementos del sistema

Dados los requerimientos de velocidad de proceso del sistema, se ha recurrido a una arquitectura de control distribuido para el desarrollo del sistema calibrador MAXSORTER. En dicha arquitectura, el conjunto de las tareas a realizar en el sistema se reparte en diferentes módulos que incorporan un dispositivo programado para desarrollar sus funciones. Como resultado se obtiene un aumento en la velocidad de proceso debido al desarrollo en paralelo de las tareas a realizar en el calibrador.

El correcto funcionamiento de un sistema de control distribuido se debe basar en la implementación de un sistema fiable de comunicaciones que sirva para que el proceso de las distintas tareas se desarrolle de forma coordinada. En este caso, los diferentes módulos del sistema se articulan en dos buses de comunicación distintos, en función de los requerimientos de latencia y de la cantidad de información a transmitir. Se utiliza un bus CAN para mensajes cortos de tiempo real y un bus LAN para mensajes de mayor cantidad de información (Calleja, 1999).

A continuación se discuten brevemente los diferentes subsistemas que componen el sistema de clasificación del calibrador.

5.4.2 El entorno de usuario

Arquitectura. Basado en arquitectura PC en Windows NT y programado bajo Windows (Polo, 1999).

Tareas. La principal tarea del entorno de usuario es permitir la interacción de sistema con el usuario. Entre sus funciones se encuentran:

- El usuario es capaz de realizar tareas de mantenimiento, verificación y análisis remoto del sistema a través del entorno.

- El usuario es capaz de realizar programas de clasificación que transmitirá al módulo encargado de dicha función.

- El usuario puede confeccionar programas de vaciado y reparto de la fruta en las diferentes salidas del sistema atendiendo a su categoría.

- La implementación de una arquitectura PC permite el uso de herramientas estadísticas para el procesado posterior de los resultados de la calibración.

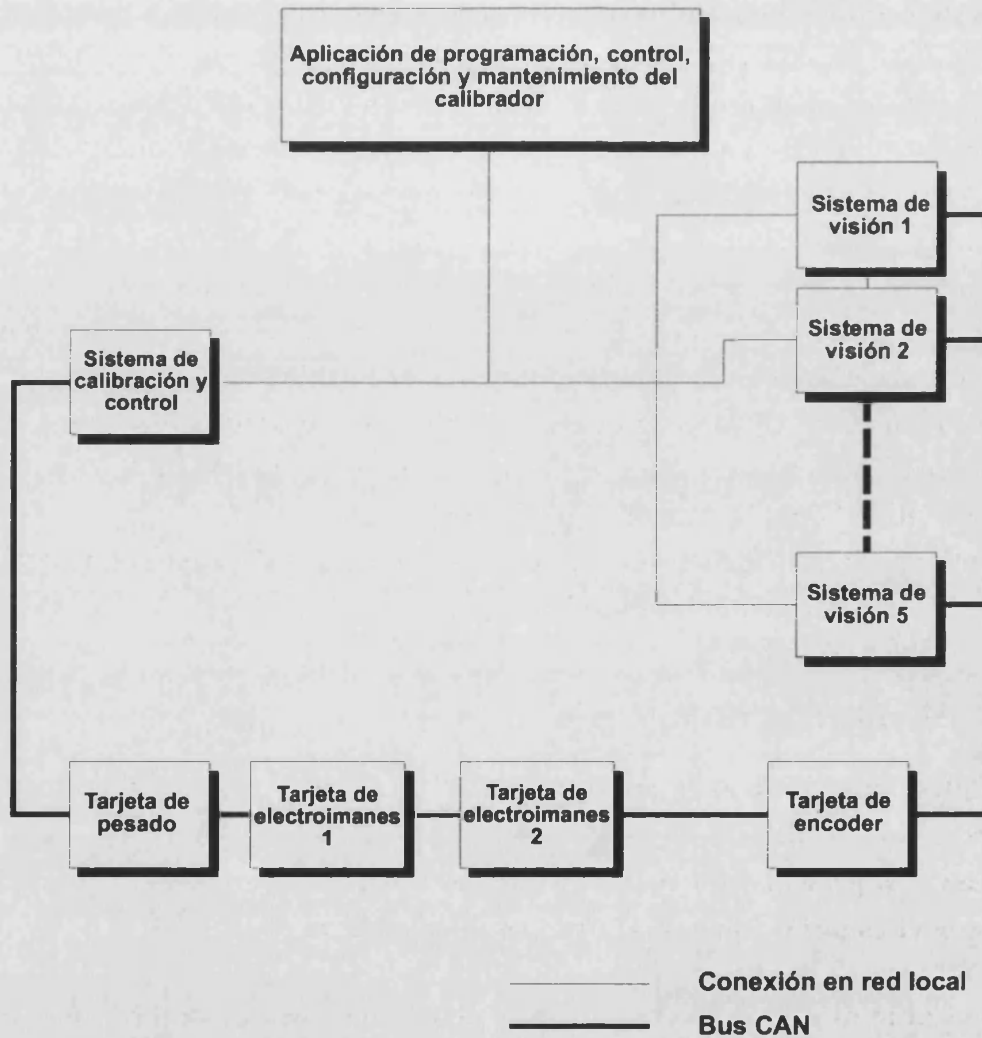


Figura 5.2: Organización del sistema desde el punto de vista del software y las comunicaciones.

Buses de comunicaciones. Utiliza un bus digital Ethernet para implementar la comunicación con un protocolo LAN IPX. Para asegurar la robustez del sistema, se añade una protección de pérdida de tramas basada en una ventana deslizante de tamaño 1.

Las figuras 5.3 a 5.12 ilustran diferentes ventanas del interfaz de usuario que se corresponden con interfaces de mantenimiento, tarado, osciloscopio, gestión de las partidas y las estadísticas, elaboración de grupos de color, tamaño y peso, etc.

Figura 5.3: Formato del interfaz para el mantenimiento.

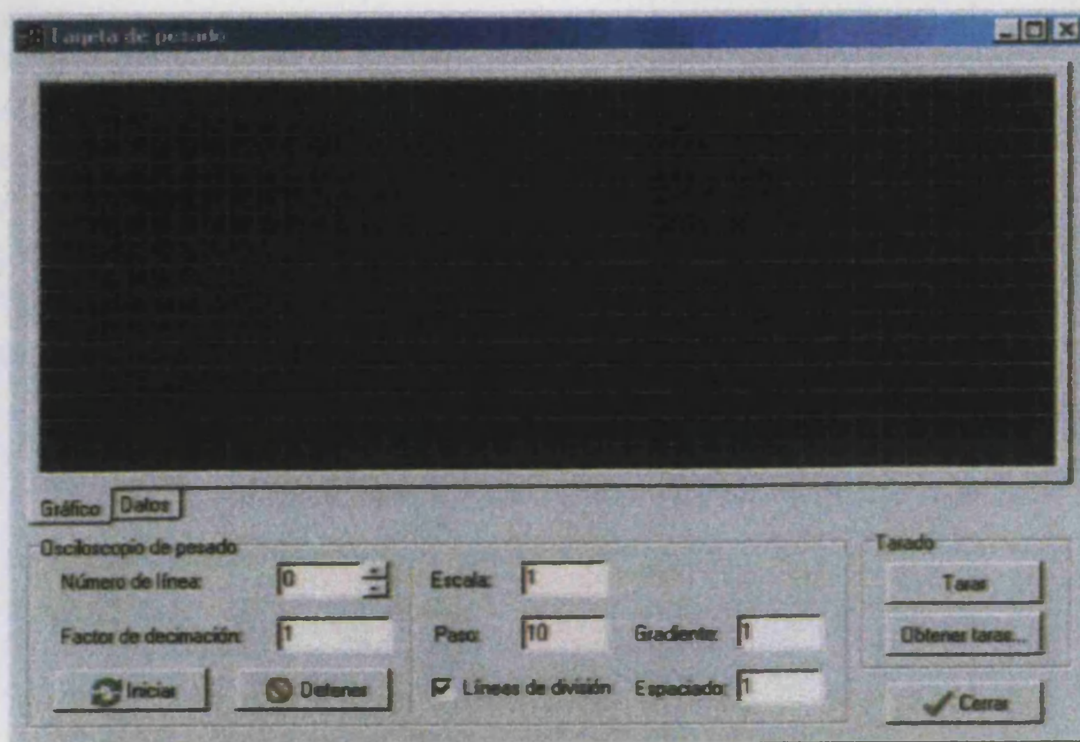


Figura 5.4: Formato del interfaz para el mantenimiento de la tarjeta de pesado.

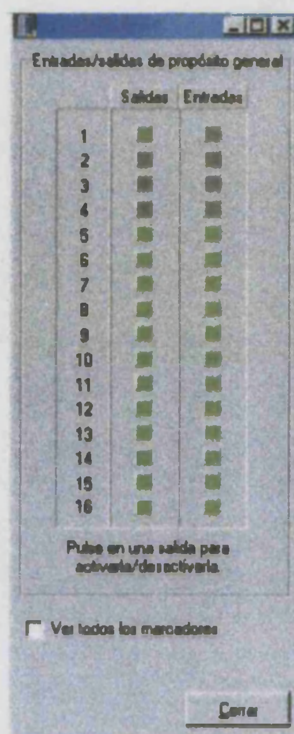


Figura 5.5: Formato del interfaz para la tarjeta de electroimanes.

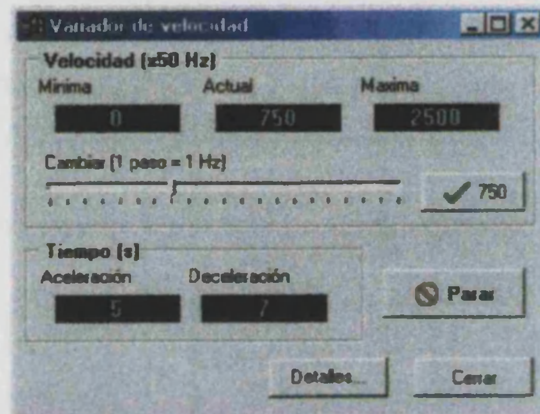


Figura 5.6: Formato del interfaz para el control de los variadores.

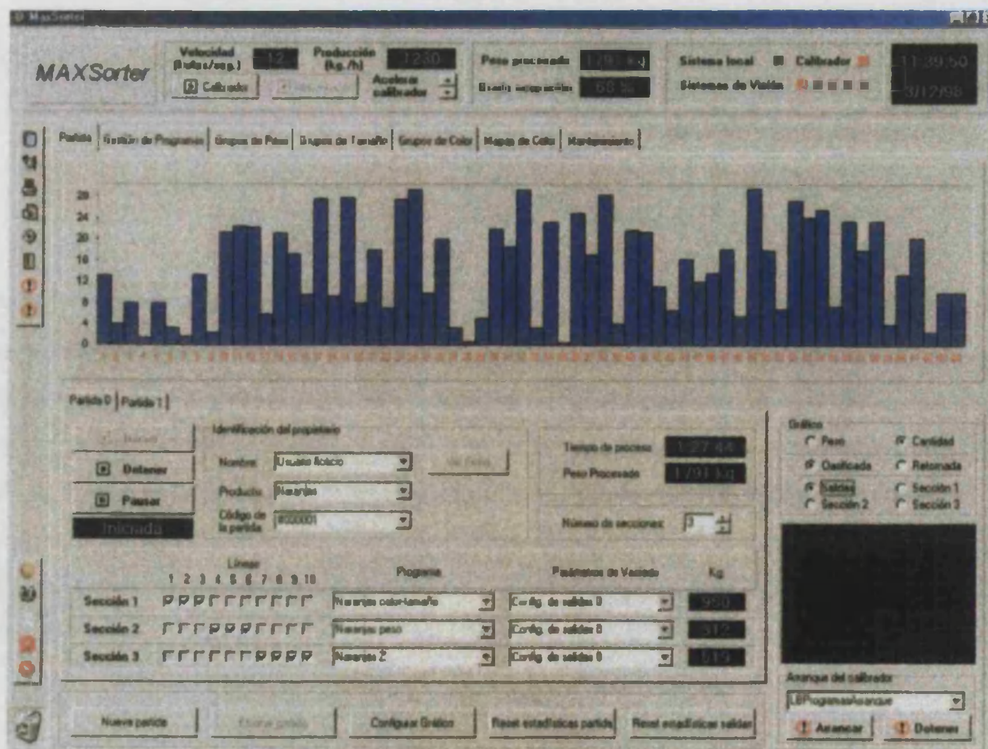


Figura 5.7: Formato del interfaz para la gestión de partidas y la representación gráfica de estadísticas.

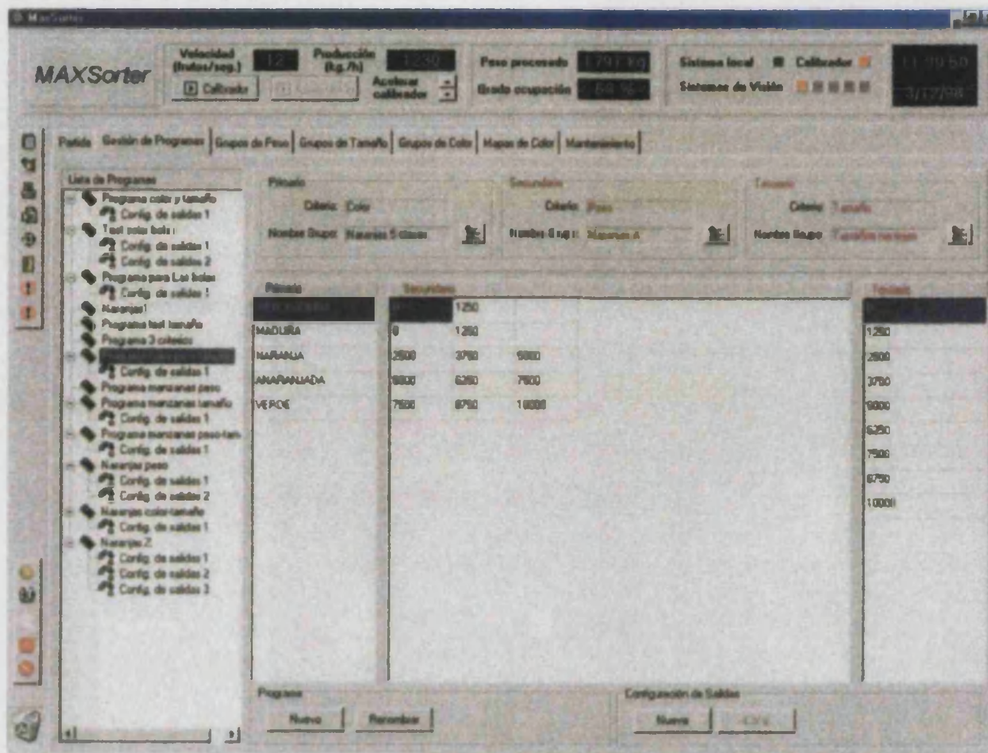


Figura 5.8: Formato del interfaz para la gestión de programas.

Figura 5.9: Formato del interfaz para los mapas de color.

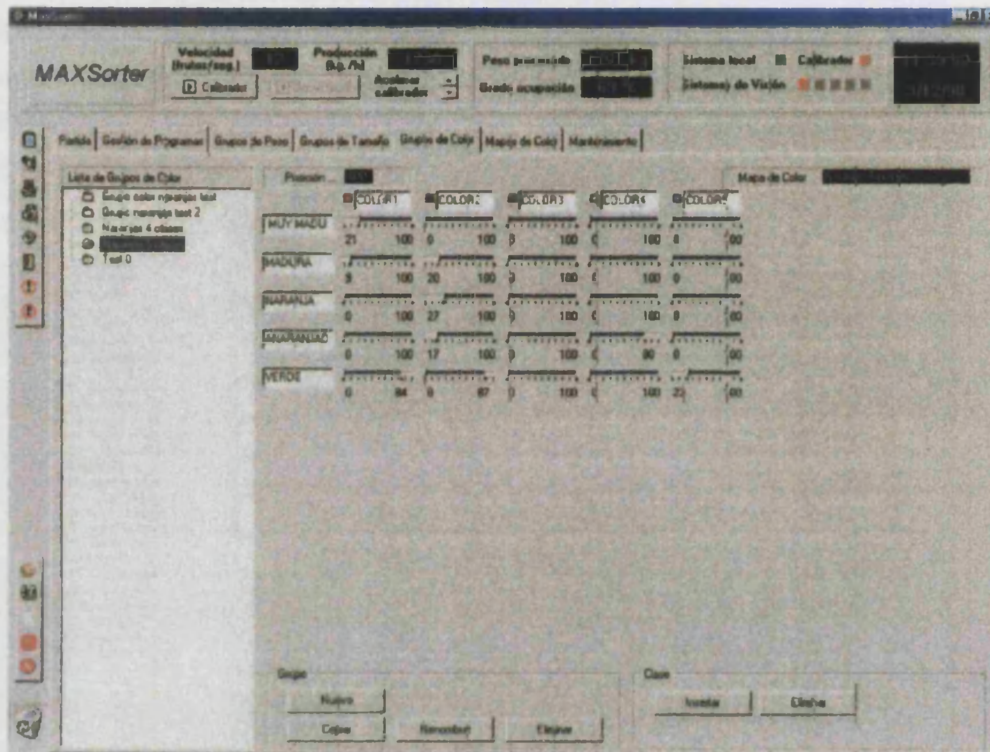


Figura 5.10: Formato del interfaz para los grupos de color.

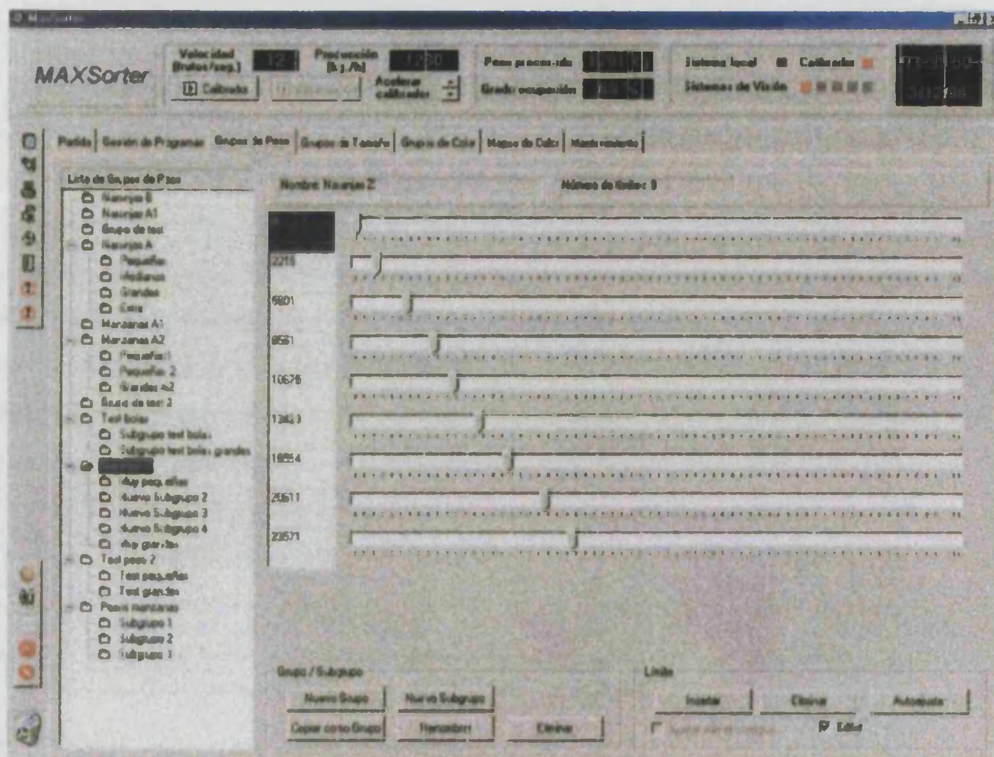


Figura 5.11: Formato del interfaz para los grupos de peso, el formato para los grupos de tamaño es el mismo.

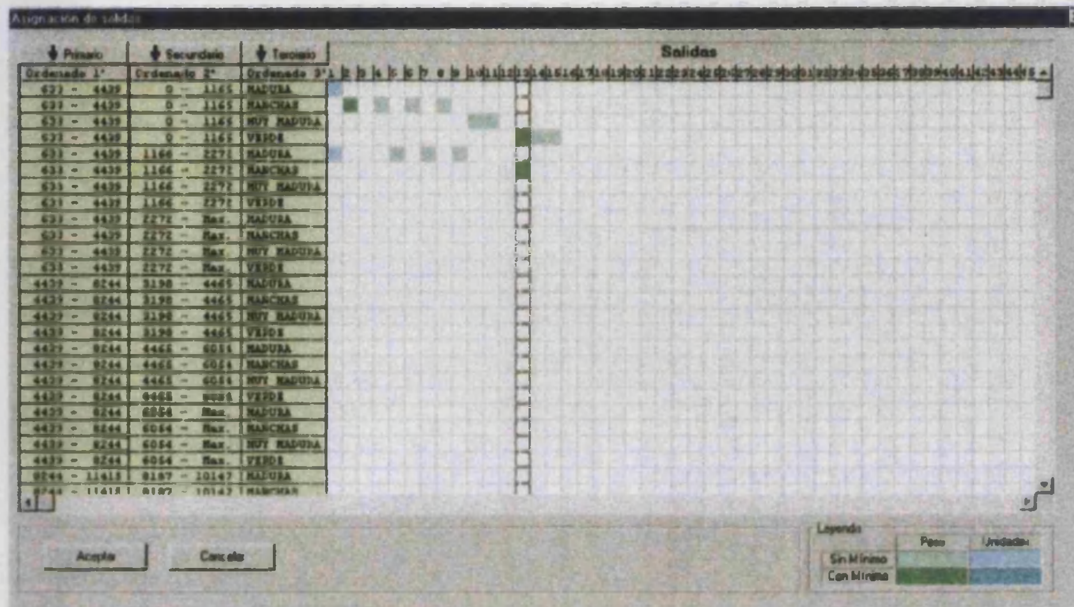


Figura 5.12: Formato del interfaz para la asignación de salidas.

5.4.3 El módulo de clasificación

Arquitectura. Basada en un PC bajo sistema operativo MS-DOS.

Tareas. Clasificación de los frutos a partir del análisis realizado por el módulo de pesado y el sistema de visión. Realiza operaciones de puente de comunicaciones entre el bus CAN y el entorno de usuario. Asimismo realiza el control de diversos dispositivos conectados al bus CAN.

Buses de comunicaciones. Dicho sistema utiliza el bus CAN como medio de transmisión con los diferentes módulos de procesado de la fruta y de la red LAN para la comunicación con el entorno de usuario.

5.4.4 El módulo de pesado

Arquitectura. Basada en una arquitectura DSP de coma fija.

Tareas. Obtención de las características de peso del fruto mediante el procesado de la señal de una célula de carga. Existe un sólo módulo de pesado escalable para gestionar todo el calibrador. Realiza la calibración de los conversores. Debe sustraer a cada peso obtenido el peso de la taza para obtener el peso en gramos de la pieza; para ello ha de mantener una serie de datos de taras o pesos de tazas vacías y, durante funcionamiento normal, si éstas pasan vacías realizar retarado dinámico para evitar que la suciedad depositada enmascare el peso verdadero de la fruta.

Buses de comunicaciones. Utiliza el bus CAN como medio de comunicación.

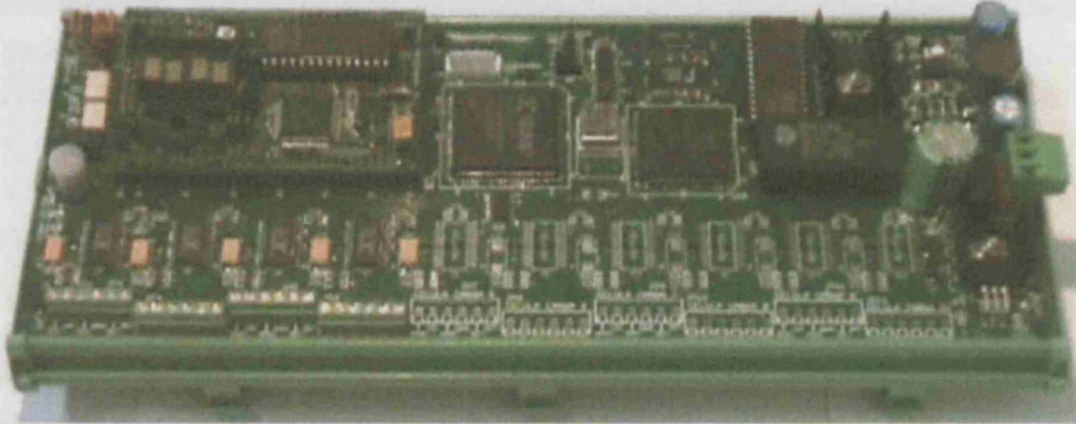


Figura 5.13: Fotografía del módulo de pesado.

5.4.5 El sistema de visión

Arquitectura. Basada en PC con tarjeta de adquisición de imágenes.

Tareas. Obtiene los resultados de color y tamaño de los frutos a partir de unos programas de trabajo predeterminados por el usuario. Además suministra imágenes de los frutos al entorno de usuario y realiza tareas de mantenimiento de las herramientas de adquisición de imágenes.

Bus de comunicaciones. Dicho sistema utiliza el bus CAN como medio de comunicación de los resultados del análisis del color y tamaño de los frutos para transmitirlos al sistema de clasificación. Asimismo posee acceso al bus LAN para la transmisión de imágenes y desarrollo de tareas de mantenimiento por parte del entorno de usuario.

5.4.6 El módulo de encoder

Arquitectura. Basada en una arquitectura 8051.

Tareas. Gestiona el funcionamiento de un encoder incremental y un detector inductivo como inicio de línea. Dichos dispositivos se encuentran asociados al mecanismo de arrastre de la cadena de transporte del calibrador. El encoder incremental proporciona un tren de pulsos que se relaciona con el transporte de tazas del calibrador, sincronizando la posición de las tazas que se encuentran en los diferentes módulos del sistema.

Bus de comunicaciones. Dicho sistema utiliza el bus CAN como medio de transmisión.

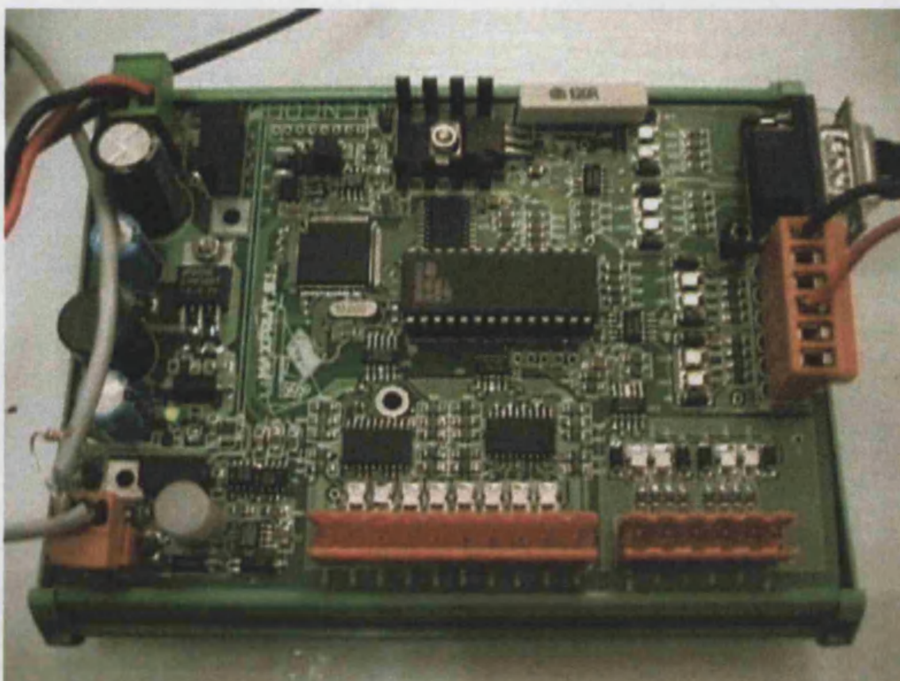


Figura 5.14: Fotografía del módulo de encoder.

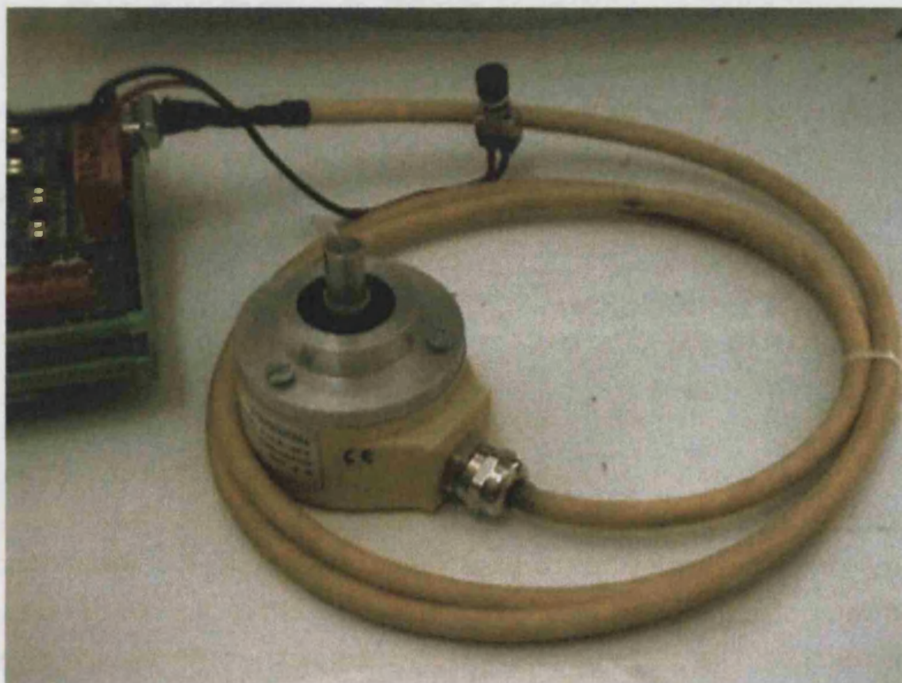


Figura 5.15: Conexión del encoder.

5.4.7 El módulo de electroimanes

Arquitectura. Basada en una arquitectura 8051.

Tareas. Controla la apertura y cierre de los electroimanes permitiendo el volcado de la fruta por las diferentes salidas del calibrador.

Bus de comunicaciones. Dicho sistema utiliza el bus CAN como medio de transmisión.

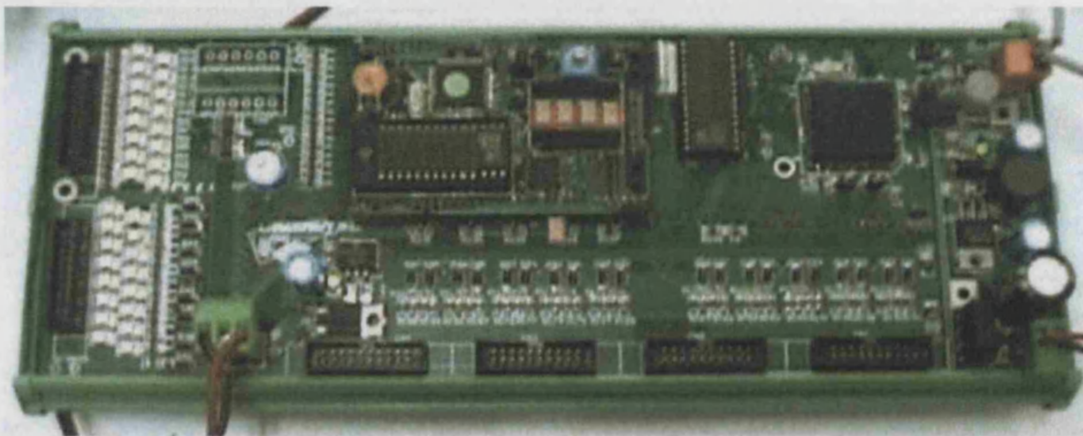


Figura 5.16: Fotografía del módulo de electroimanes.

Capítulo 6

El módulo de pesado

6.1 Introducción

El módulo de pesado es el encargado de realizar la adquisición y el procesado de la señal de hasta diez células de carga, para obtener las características de los frutos que sobre éstas pasan.

Es un módulo inteligente y escalable, pudiendo usarse cualquier número de líneas hasta diez. Si la calibradora necesitara más de diez líneas se colocaría otro módulo de pesado trabajando en paralelo con una identificación diferente en el bus CAN.

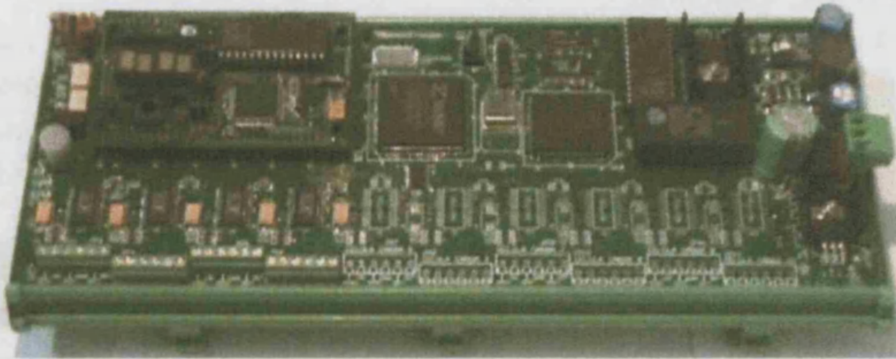


Figura 6.1: Fotografía del módulo de pesado.

El módulo de pesado es una tarjeta hardware de 4 capas, diseñada con planos de alimentación y tierra para eliminar las interferencias EMI (Johnson, 1993). Posee un zócalo para la placa de comunicaciones, que se ha diseñado independientemente para poder ser reutilizada en otros módulos. La placa de comunicaciones es la única forma de comunicación con el exterior en funcionamiento normal. No obstante, tam-

bién implementa una vía de comunicación RS232 con el DSP para realizar tareas de depuración durante el desarrollo de la aplicación.

El DSP TI-TMS320C26 es el núcleo indiscutible del módulo, que realiza tanto tareas de control como de procesado digital, función para la cual está diseñado. Éste debe adquirir la señal de hasta diez células de pesado e identificar los pesos de las frutas que pasan sobre ellas a un ritmo de hasta 20 frutos/segundo. A plena carga esto significa un rendimiento de 200 frutos/segundo.

Otro elemento funcional importante es la CPLD, cuya lógica programable implementa el interfaz del DSP con la memoria, el arbitraje en las comunicaciones serie entre el DSP y los conversores AD7730 y la implementación de un buffer intermedio para la comunicación del DSP con la placa de comunicaciones CAN. Además, es posible la reprogramación de la CPLD sobre el sistema a través de un conector JTAG.

La figura 6.1 es una imagen del módulo de pesado. En el transcurso del presente capítulo se describen, uno a uno, los diferentes elementos funcionales de la tarjeta de pesado.

6.2 Comunicaciones CAN

La placa de comunicaciones ha sido diseñada como tarjeta independiente, del tamaño de una tarjeta de crédito (85 x 56 mm), con dos tiras de pines que le permiten la inserción en otras tarjetas de aplicación específica para dotarlas de un canal de comunicación CAN, figura 6.2.

La placa de comunicaciones puede establecer comunicación serie o paralelo con la tarjeta de aplicación específica sobre la que se basa. La idea es proporcionar una conexión al bus CAN transparente para la tarjeta sobre la que se inserta, la cual simplemente envía y recibe los mensajes vía serie (o paralelo) desentendiéndose de las particularidades del CAN.

La placa de comunicaciones CAN asume las tareas de gestión del bus (como arbitraje para el envío de mensajes CAN, escucha para la recepción, etc.), mantenimiento de listas de errores, estadísticas del tráfico, identificación de la tarjeta en el sistema, visualización del modo de funcionamiento y monitorización del sistema.

La placa de comunicaciones está basada en el microcontrolador C515C, una versión del SAB 80C515A de 8 bits, que proporciona un interfaz serie síncrono SPI y un interfaz de bus CAN (Siemens, 1997). El controlador de CAN es el elemento funcional que proporciona los recursos necesarios para ejecutar tanto el protocolo CAN (identificadores de 11 bits) como el protocolo extendido (identificadores de 29 bits). Éste proporciona un sofisticado mecanismo que libera a la CPU del máximo *overhead*

posible controlando: arbitrio de bus, manejo de errores, generación de interrupciones, etc. y mapeandc

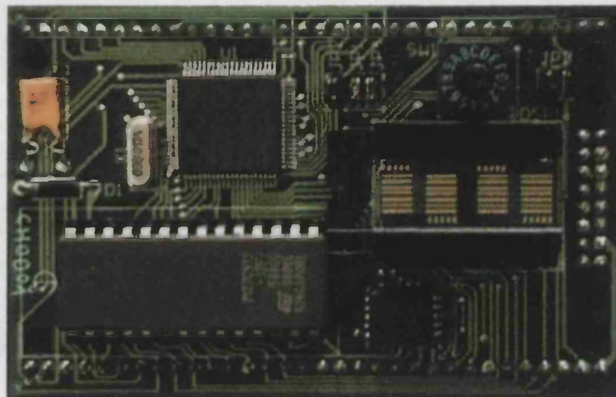


Figura 6.2: Fotografía de la placa de comunicaciones.

Posee un mapa de memoria formado por :

- 1 KB de RAM.
- 64 KB de EPROM.
- 256 Bytes de EEPROM.

La dirección en el bus CAN del módulo es configurable a través de un selector rotativo de 16 posiciones, que lo identificará en una red general, y de un visualizador (*display*) de cuatro caracteres para la presentación de información como resultados del autotest, el estado actual, etc.

La placa de comunicaciones cumple las especificaciones CAN 2.0B. Está diseñada con un plano de tierra para evitar interferencias EMI. Posee diversas protecciones: fusible electrónico para protección contra sobreintensidades, supresor de sobretensión para protección contra transitorios en el voltaje de alimentación y diodo en antiparalelo para prevenir inversiones de polaridad. Utiliza una tensión de alimentación de +5VDC y posee un consumo de 300mA.

Pin	Señal	Descripción
1	CAN_H	Terminal para la conexión de la señal CAN H
2	CAN_L	Terminal para la conexión de la señal CAN L
3	GND_CAN	Terminal para la conexión de la masa del bus CAN

Tabla 6.1: Conexión del módulo al bus CAN

6.2.1 Modos de funcionamiento

Inicialmente se puede distinguir dos modos de funcionamiento, que también afectan al modo de ejecución de la placa de aplicación con la que establece comunicación.

Modo normal de funcionamiento

La placa de comunicaciones *selecciona el modo normal de funcionamiento cuando inicialmente el jumper de test de la tarjeta está abierto.* En este punto, el selector indica el identificador de la placa de comunicaciones en el bus CAN. Nótese que posteriormente, el jumper puede ser cerrado y el selector cambiado sin afectar al modo de funcionamiento ni al identificador de la placa de comunicaciones; éstos son captados durante la inicialización de dicho módulo.

La placa de comunicaciones mostrará en el display el mensaje **"NO APLICACIÓN"** hasta que la tarjeta de pesado establezca comunicaciones mediante el envío del mensaje **"TARJETA_PESADO_PRESENTE"**. Una vez establecida la comunicación, la placa de comunicaciones (o *módulo de comunicaciones*) identifica la vía por la que la tarjeta establece la comunicación: paralelo o serie. Ésta es mostrada en el display indicando **"PARALELO OK"** o **"SERIE OK"**. La tarjeta de pesado establece siempre comunicación paralela.

Para que la inicialización de la comunicación sea completa, el módulo de pesado envía al control el mensaje CAN recibido. Éste debe ser contestado mediante el mensaje **"CONFIRMACIÓN_PRESENCIA"**. El módulo de comunicaciones mostrará el mensaje **"NO CONTROL"** hasta que la comunicación CAN se haya establecido correctamente, esto es, hasta que el control haya contestado a la petición de la tarjeta, declarándose establecida la comunicación control - tarjeta de pesado.

Tras el establecimiento de la comunicación, la tarjeta envía el modo de funcionamiento en que se encuentra. La tabla 6.2 ilustra los mensajes que muestra el display en función del modo de funcionamiento de la aplicación.

MODO DE FUNCIONAMIENTO DE LA APLICACIÓN	DISPLAY
EN ESPERA	"EN ESPERA"
CLASIFICANDO	"CLASIFICANDO"
EN MANTENIMIENTO	"EN MANTENIMIENTO"
RECIBIENDO LA CONFIGURACIÓN	"EN CONFIGURACIÓN"

Tabla 6.2: Mensajes del display en cada modo de funcionamiento.

En los modos EN ESPERA ó CLASIFICACIÓN, pueden darse varios casos:

- Si no surgen errores en la aplicación, se visualizan el mensaje **"OK"**.
- Si la aplicación envía algún mensaje de error para ser mostrado en el *display*, se visualiza el mensaje **"ER X"**, siendo x el número de errores enviados por la aplicación.
- Si la placa de comunicaciones detecta algún error relacionado con las comunicaciones, visualiza el mensaje **"OK #"**. Estos errores son propios de la placa de comunicaciones y no tienen que ver con el proceso realizado en la placa de aplicación. Los errores que pueden detectarse son los siguientes:
  - *Pérdida de sincronismos*: Llega un nuevo sincronismo antes de que la aplicación haya terminado de gestionar el anterior.
  - *Pérdida de mensajes CAN*: No es posible *encolar* los mensajes CAN recibidos porque llegan demasiado rápidos.
  - *Mensajes CAN no encolados*: El buffer de mensajes a transmitir a la aplicación, se ha llenado. Esta circunstancia se da cuando la aplicación no puede procesar a tiempo todos los mensajes recibidos.

En los modos MANTENIMIENTO o CONFIGURACIÓN, el display sencillamente visualiza "EN MANTENIMIENTO" o "EN CONFIGURACIÓN" respectivamente.

Como sabemos, tras la inicialización del módulo de comunicaciones es posible actuar sobre el selector y el jumper de test sin que esto incida sobre el identificador de éste en la red CAN, ni sobre el modo de funcionamiento. Esto se hace para poder utilizar el display con otros fines cuando se cierra el jumper de test. Cerrando éste y actuando sobre el selector se pueden monitorizar, a través del display, información sobre las comunicaciones, tabla 6.3.

La sección 7.3 detalla la estructura de los mensajes CAN y los mensajes internos de la placa de comunicaciones. Los mensajes utilizados en el transcurso de la aplicación se listan en el apéndice J.

Modo de test

Si durante la inicialización de la placa de comunicaciones el jumper de test está cerrado, la placa de comunicaciones entra en este modo de funcionamiento. Al entrar en el modo test, el *display* muestra inicialmente el mensaje **"TEST #X"** y un mensaje adicional informando de las comprobaciones realizadas, siendo X el índice del selector.

Los test realizados se pueden clasificar, tabla 6.4, en:

Posición del selector	Formato información	Información proporcionada
1	"CAN R#"	Número de mensajes CAN enviados por la aplicación (incluyendo los sincronismos)
2	"CAN T#"	Número de mensajes CAN enviados a la aplicación (debería coincidir con el número de mensajes recibidos por el bus, a no ser que se sature la cola de recepción)
3	"BUS R#"	Número de mensajes CAN recibidos, por la placa de comunicaciones, del bus.
4	"BUS T#"	Número de mensajes CAN enviados por la placa de comunicaciones al bus (Debe coincidir con el número de mensajes CAN enviados por la aplicación -posición 1-)
5	"BUS P#"	Número de mensajes CAN perdidos, es decir, que no se han podido leer a tiempo de los registros del controlador.
6	"COM R#"	Número de mensajes internos recibidos por la placa de comunicaciones.
7	"COM T#"	Número de mensajes internos enviados por la placa de comunicaciones a la aplicación.
8	"RxDSP#"	Veces que la cola de mensajes CAN se ha llenado, de mensajes enviados por la aplicación, y ha debido esperar hasta que se transmita alguno para seguir recibiendo.

Tabla 6.3: Información sobre las comunicaciones.

- *Test de la placa de comunicaciones*: No establecen comunicación con la aplicación.
- *Test de la placa de aplicación específica*: La placa de comunicaciones establece comunicación con la aplicación y le señala el índice de test que debe realizar. Esperará entonces a que la aplicación le indique la finalización del test que está realizando.

Si la placa de comunicaciones ha recibido mensajes de error durante el proceso de test, éstos serán mostrados una vez el test haya finalizado. Si no ha recibido error alguno, se mostrará un mensaje "**OK**" en el *display*.

Posición del selector	TEST REALIZADO
Tests de la placa de comunicaciones	
0	Accede al *display* y muestra **"OK"**
1	Envía un mensaje CAN (Id. CAN 0) al Control y espera una respuesta.
Tests para la placa de pesado	
2	Establece comunicación con la Aplicación **vía paralelo**. Envía la orden interna ORDEN_TEST_ COMUNICACIONES y espera recibir el mismo mensaje.
OTRA	Envía la orden interna ORDEN_TEST_ COMUNICACIONES y espera recibir el mismo mensaje.

Tabla 6.4: Tests realizados.

6.3 El procesador digital TMS320C26

El procesador digital de señal TMS320C26 (TI-C2x, 1993), también denominado a partir de ahora 'C26,es el núcleo indiscutible de la tarjeta de pesado. Al igual que el resto de procesadores digitales de señal, DSP, está especialmente *diseñados para implementar algoritmos típicos del procesado digital de señal en tiempo real.* Las operaciones en las que están especializados implican combinaciones de datos relativamente simples que generalmente incluyen multiplicaciones. La "primitiva" más común es:

$$Y = M \cdot X + N \tag{6.1}$$

utilizada tanto en filtrado digital, como en convolución, transformada rápida de Fourier, etc. Esta sencilla ecuación involucra una suma y una multiplicación. La multiplicación es una instrucción compleja en microprocesadores y microcontroladores de propósito general, en los que suele estar microprogramada. Sin embargo, un DSP puede realizar en un ciclo de instrucción una suma y una multiplicación, como indica la ecuación anterior. Así, los DSP optimizan dos características básicas: velocidad y precisión de 16 bits (TI-C2x, 1993; Marven, 1994; Grover, 1998).

La aplicación de pesado se caracteriza por la necesidad de funcionamiento en estricto tiempo real. Ésta debe realizar cierta cantidad de procesado digital, básicamente filtrado digital, que supone un gran peso para el procesador, sobre todo funcionando a plena carga con 10 líneas a 20 frutas/segundo. Ésta es la clave de la elección de un procesador digital de señal como núcleo del sistema. Éste será el encargado de realizar tanto las funciones de control, como el procesado en tiempo real de la aplicación.

6.3.1 Características básicas del TMS320C26

El TMS320C26 implementa una arquitectura de tipo Harvard, manteniendo internamente dos estructuras de bus independientes para datos y programa. El C26 dispone de tres grandes bloques de memoria RAM interna (B0, B1 y B3), cada uno de los cuales puede ser configurado independientemente como memoria de datos o de programa. Un cuarto bloque (B2) de memoria interna está siempre configurado como memoria de datos, figura 6.4.

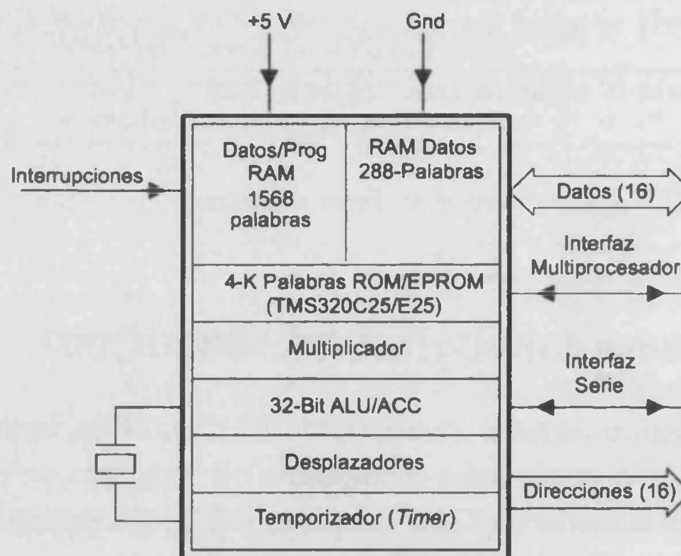


Figura 6.3: Diagrama simplificado de bloques del TMS320C26.

El TMS320C26 dispone una PROM interna que contiene un *cargador* de 256 palabras. La capacidad de direccionamiento total es de 64 Kwords tanto para memoria de datos como para memoria de programa, pudiendo así direccionar gran cantidad de memoria externa. Los programas pueden ser ejecutados a toda velocidad cuando programa y datos están situados en memoria interna o cuando el programa está situado en memoria externa y los datos en memoria interna (asumiendo que la memoria externa posee cero estados de espera). Es por ello que la aplicación y el sistema operativo, véase capítulos 7 y 8, se sitúan en memoria externa y la memoria interna se asigna a los datos más frecuentemente usados (TI-C2x, 1993).

El 'C26 utiliza una anchura de bus, tanto interno como externo, de 16 bits. Opera con aritmética entera de complemento a dos, disponiendo de una ALU y acumulador de 32 bits. La ALU es la encargada de realizar las operaciones aritméticas y lógicas. Para ello utiliza operandos que puede obtener a partir de memoria (16 bits) o el resultado de los productos del multiplicador interno (32 bits). El acumulador almacena los resultados de la ALU y es a su vez la segunda entrada de la misma.

El multiplicador es el elemento fundamental del DSP. Éste realiza productos entre palabras de 16 bits codificadas como enteros con signo. El multiplicador consta de dos registros, T (16 bits) y P (32 bits), y el array multiplicador. El registro T almacena temporalmente un multiplicando, mientras que el P almacena el resultado de la última multiplicación. La utilización de un multiplicador rápido es esencial para implementar operaciones básicas de procesado digital tales como convolución, correlación y filtrado.

El registro de desplazamiento permite un desplazamiento hacia la izquierda de 0 a 16 bits de los datos que van a ser usados por la ALU o enviados al acumulador. Así, se puede usar sin penalización temporal, simultáneamente con otra acción. Esta capacidad de desplazamiento permite al procesador realizar importantes operaciones del procesado digital como escalado numérico, extracción de bits, aritmética extendida,prevención de desbordamientos, etc. El desplazador tiene una entrada de 16 bits desde el bus de datos y una salida de 32 bits conectada a la ALU.

El interfaz con la memoria local consiste en un bus de datos de 16 bits (D15-D0), un bus de direcciones de 16 bits (A15-A0), tres pines para seleccionar entre acceso a memoria de datos, programa o espacio de E/S (/DS, /PS y /IS), y varias señales de control (R/W, /STRB, READY, etc).

El TMS320C2x puede controlar un espacio global de memoria de datos, a través de un registro de localización de memoria global (GREG), que permite especificar hasta 32 KW como memoria global externa de datos. El control de accesos a esta memoria se realiza mediante las señales /BR (*Bus Request*) y READY.

También se dispone de ocho niveles internos de pila utilizados en el servicio de interrupciones. El control de las operaciones está basado en un *timer* interno de 16 bits, un contador de repetición de instrucciones (8 bits), tres interrupciones externas del usuario (enmascarables), e interrupciones internas generadas por el puerto serie interno y el timer.

Dispone de un canal serie *full-duplex* permite conexión directa con algunos codecs, conversores ADC serie y otros sistemas. La comunicación puede funcionar en formato 8/16 bits. Cada registro serie tiene una entrada de reloj, una entrada de sincronismo y los registros de desplazamiento asociados (TI-C2x, 1993).

El puerto serie es usado, en la tarjeta de pesado, para la comunicación con los conversores AD7730. En el apéndice B se revisará más detalladamente su operación y funcionamiento.

6.4 Lógica programable

Toda la lógica necesaria del sistema y la implementación de interfaces y buffers, se ha integrado en una CPLD Xilinx XC95108, programable en el sistema vía JTAG.

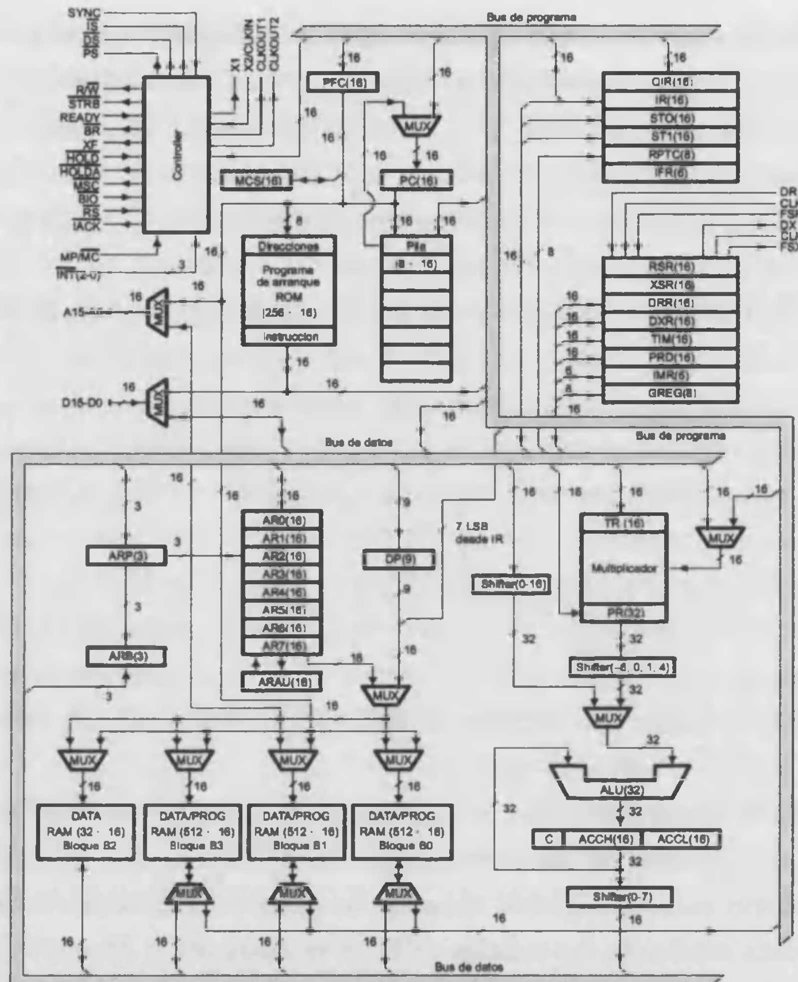


Figura 6.4: Diagrama de bloques del TMS320C26.

Ésta es una CPLD del tipo jerárquico con bloques que realizan una función lógica y están interconectados entre sí a través de una matriz central de interconexiones, figura 6.5. Cada bloque contiene una matriz AND conectada a macrocélulas lógicas. Estos dispositivos poseen menor capacidad que las FPGA, pero son mucho más flexibles que un PLD (Xilinx, 1998).

Este dispositivo está formado por seis bloques funcionales 36V18, proporcionando 2400 puertas reutilizables y con una velocidad de propagación de la señal de 7.5ns entre entrada y salida. Posee varias entradas de reloj y soporta una alta corriente de salida (24mA). La cantidad de bloques lógicos varía de 2 a 8 así como la complejidad según el modelo.

Los rasgos principales son los siguientes:

- Velocidad de propagación: 7.5 ns de retardo entre pin y pin, para todos los pines.

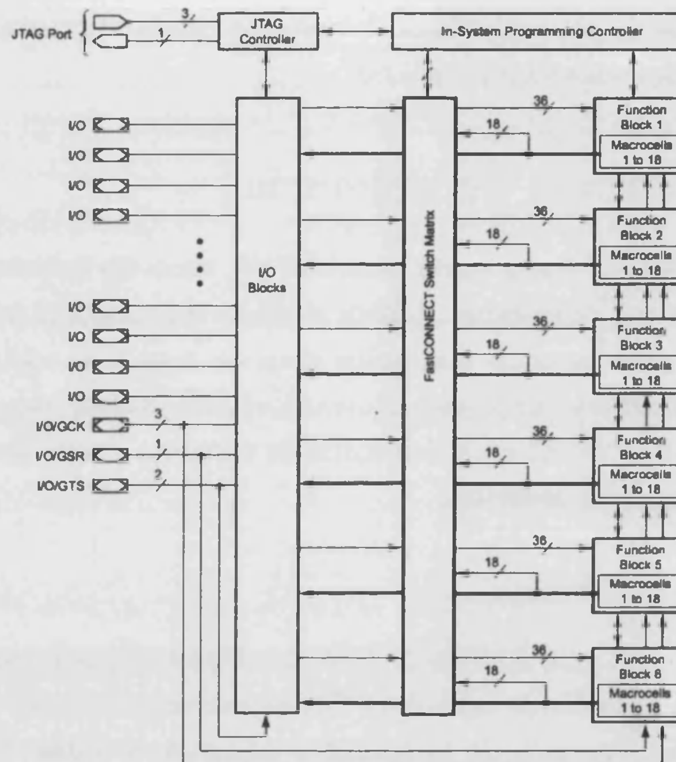


Figura 6.5: Arquitectura de la XC95108.

- Soporta velocidades de reloj desde continua hasta 125 Mhz.
- 108 macrocélulas con 2400 puertas útiles.
- Hasta 108 pines de I/O.
- Programable en sistema (ISP).
- Bloques funcionales flexibles 36V18.
- Soporte para JTAG IEEE Std 1149.1
- Modo de reducción de potencia programable para cada macrocélula.
- Soporta altas corrientes de salida, hasta 24 mA.

Uno de los rasgos fundamentales de la CPLD es que ésta se puede programar incluso montada en la tarjeta, a través de un conector de seis pines, cuatro de señal y dos de alimentación (Xilinx, 9997). Esto permite realizar reprogramaciones e introducir mejoras sin desoldar el componente. La programación implementada en la CPLD de la tarjeta de pesado, se referenciará a lo largo del capítulo, describiéndose su utilización en: interfaz de memorias, arbitrio del interfaz serie, señales de control

de los conversores, e implementación del *buffering intermedio* para la comunicación entre la placa de comunicaciones y el DSP.

6.5 Organización de memoria

La tarjeta de pesado posee cierta cantidad de memoria externa que, sumada a la memoria interna del procesador, genera el mapa de memoria del sistema. Dicho mapa de memoria está formado por varios tipos de memoria, según las necesidades de volatilidad, velocidad y capacidad. Además, el DSP utiliza una arquitectura tipo Harvard, por lo que no existe un único mapa de memoria monolítico, sino diferentes espacios de direcciones de memoria.

6.5.1 Memoria interna

El 'C26 implementa una arquitectura de tipo Harvard para maximizar la potencia del procesado manteniendo separadas dos estructuras de buses de memoria para acelerar la ejecución: memoria de programa y memoria de datos. Obviamente, existen instrucciones que permiten realizar transferencias de datos entre ambos espacios. Externamente, la memoria de datos y de programa se multiplexa a través de un único bus para poder acceder a todo el rango de direcciones minimizando el número de pines del dispositivo, y con ello precio y espacio ocupado.

Memoria on-chip

El procesador TMS320C26 dispone de un total de 1568 palabras de 16 bits en memoria interna RAM, divididas en cuatro bloques separados (B0, B1, B2 y B3) y de las cuales 32 son siempre memoria de datos, mientras que todas las restantes pueden configurarse mediante software como memoria de programa o de datos.

El 'C26 proporciona tres espacios de direcciones (recordemos que la arquitectura Harvard proporciona espacios separados de memoria interna) para memoria de programa, de datos y puertos de E/S. Cada uno de estos espacios puede direccionar hasta 64K palabras (64KW). Estos espacios son controlados externamente por las señales de los pines /PS, /DS e /IS (selección del espacio de programa, datos y E/S respectivamente. Estas señales junto con /STRB se activan únicamente durante los accesos externos al bus para indicar a qué espacio se refiere el acceso externo.

El 'C26 ofrece una posibilidad de controlar el mapa de memoria a través de la señal de control del pin MP/MC (microprocesador/microcomputador). Ésta selecciona el modo de funcionamiento del procesador como microcontrolador o como microprocesador, permitiendo que se utilice la ROM y la tabla de vectores de interrupción interna

si se selecciona como microcomputador, o externa si es configurado como microprocesador.

El modo microprocesador permite el uso de una ROM externa para arrancar. Sin embargo, ***nuestro sistema utiliza el modo microcontrolador*** para aprovechar el *bootloader* multipropósito grabado en la ROM del 'C26, pero sobre todo porque esto permite diferentes tipos de arranque: desde EPROM si se inicializa el sistema en modo normal, o inicialización serie si se arranca en modo depuración (apéndice F).

A partir de este momento todas las consideraciones de memoria interna parten de la premisa de que el DSP está configurado en modo microcontrolador.

Mapas de memoria del 'C26

EL 'C26 tiene la propiedad de poder utilizar hasta cuatro disposiciones de la memoria interna diferentes según la configuración seleccionada. La figura 6.6 presenta únicamente la configuración de memoria interna que es usada por el DSP a lo largo de su operación en el sistema de pesado. Esta configuración es la que dispone la instrucción CONF1.

En la figura 6.6 puede apreciarse:

- El bloque de registros mapeados en memoria y el B2 de 32 palabras.
- La tabla de vectores de interrupción y un sencillo *bootloader* multipropósito están grabados en una pequeña ROM de 256 palabras.
- El espacio de direcciones de memoria de programa desde 0 hasta 0FFFh es interno.
- El espacio de direcciones de memoria de datos desde 0 hasta 07FFh es interno.
- Si uno o más bloques son configurados como memoria de programa, se usa el espacio de direcciones de programa desde FA00h a FFFFh, que internamente está reservado para estos bloques y a los que no puede acceder como memoria externa de programa.
- Los únicos bloques reconfigurables de memoria de datos a memoria de programa son B1, B2 y B3 tienen igual tamaño y residen (cuando están en memoria de datos) en la páginas 4 a 15 [1].

[1]La página hace referencia a los 7 bits más significativos de una dirección. Éste es un concepto usado por el direccionamiento directo del 'C26, y facilita el direccionamiento de datos dentro de una misma página, apéndice G.

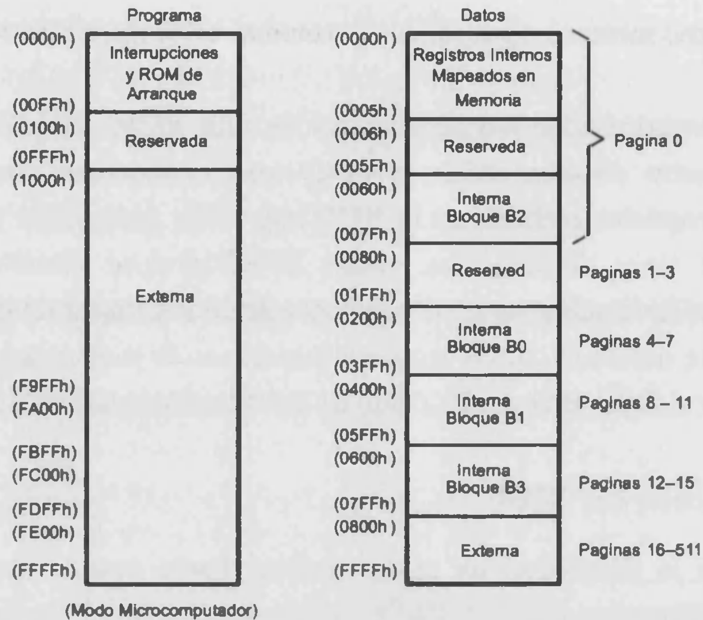


Figura 6.6: Mapa de memoria interna después de la configuración CONF1.

6.5.2 Memoria externa y memoria global

El espacio de programa externo se selecciona mediante la señal /PS (selección de programa), y puede ser usado a partir de la dirección 1000h. La memoria externa de datos, se selecciona mediante /DS (selección de datos) y siempre comienza en la dirección 800h, sea cual sea el modo de configuración de la memoria interna.

Las señales de selección de espacio de direcciones (/PS, /DS e /IS) se utilizan junto con /STRB para saber cuando se está referenciando una posición de memoria externa, dado que únicamente se activan durante los accesos externos al bus para indicar el espacio al que se refiere el acceso externo.

El acceso a memorias lentas se puede realizar gestionando la señal de control READY para introducir estados de espera, y a memorias rápidas (tiempo de acceso £ 25ns) directamente y sin estados de espera. La memoria interna (RAM/ROM) es una memoria de cero estados de espera que permite el acceso en un solo ciclo de reloj.

El 'C26 es capaz de gestionar un espacio global de memoria de datos, pensado para aplicaciones multiproceso, aunque como veremos más adelante, ésta aplicación lo usa con propósitos diferentes. El tamaño de memoria global viene determinado por el contenido de uno de los registros mapeados (GREG) en memoria que, en nuestro caso, fijaremos a GREG=080h para determinar un espacio de datos global de 32KW (concretamente de 08000h a 0FFFFh).

Los accesos a memoria local externa de datos son similares al acceso a memoria

global de datos, ya que en ambos casos se señala con la señal /DS la selección del espacio de datos y con /STRB que se está seleccionando una posición de memoria externa. Sin embargo, la distinción entre memoria local y global se obtiene por la activación de la señal /BR siempre que el DSP pretende acceder a memoria global de datos.

Como vemos, la decodificación, a través de un poco de lógica, de las señales indicadas posibilita el mapeo de cualquier tipo de memoria en cualquier espacio de direcciones del mapa de memoria externa del 'C26. En el caso de memorias lentas, la lógica de control utilizará la entrada READY para prolongar el acceso el tiempo necesario para que éste ocurra con éxito. En nuestro caso, todo el interfaz con las memorias se ha realizado mediante lógica programable de la CPLD.

Veamos a continuación los tipos de memoria que el sistema posee en el mapa de direccionamiento externo del DSP.

EPROM

La EPROM proporciona autonomía al sistema. El sistema utiliza una EPROM de 32KB mapeada en memoria global de datos a partir de la dirección 08000h. Ésta es una memoria de sólo lectura que contiene el código de la aplicación y el microkernel de sistema operativo, que debe ser cargado cada vez que arranca el sistema. La carga se realiza a través del *bootloader* del 'C26 y de un *bootloader* secundario que éste carga (apéndice F).

La figura 6.7 ilustra el interfaz de conexión de la EPROM al DSP. Nótese que la EPROM se selecciona ante accesos a la mitad superior del rango de direccionamiento de memoria de datos, siempre que se acceda a memoria global.

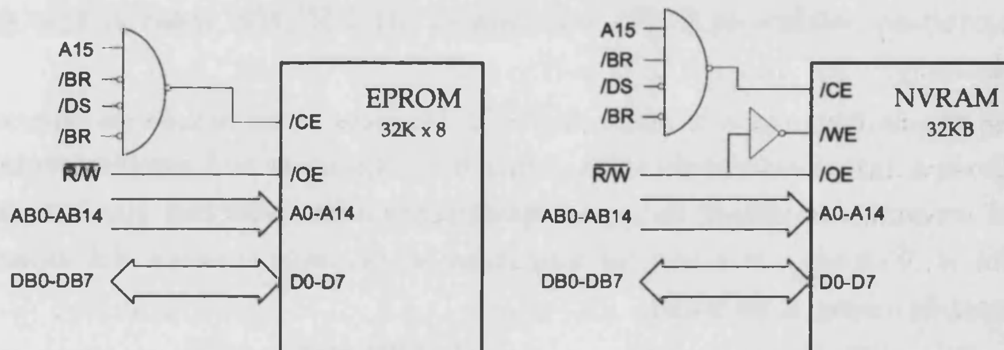


Figura 6.7: Interfaz del DSP con a) la memoria EPROM, b) la memoria NVRAM.

Dado que la EPROM posee un tiempo de acceso menor de 140ns, sólo necesitará un estado de espera en los accesos. La figura 6.8 ilustra un mecanismo simple para la generación de un estado de espera usando la salida /MCS del 'C26. Este mecanismo

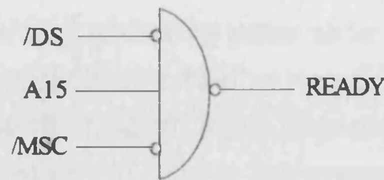


Figura 6.8: Mecanismo de generación de un único estado de espera.

debe implementarse junto con la decodificación de la selección de la memoria, para poder prolongar el tiempo de acceso durante un estado de espera. Nótese que dicho estado de espera sólo se otorgará a las memorias mapeadas en la segunda mitad del rango de direccionamiento de memoria externa de datos, es decir, afectará únicamente a la memoria EPROM y a la memoria NVRAM, *justamente aquéllas que necesitan un estado de espera.*

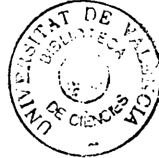
La utilización de los estados de espera ralentiza los accesos a la memoria. Sin embargo esto no es crítico en el caso de la EPROM, dado que sólo se utiliza inicialmente para el arranque y carga de la aplicación en la memoria RAM del sistema.

NVRAM

Como su nombre indica (*Non-Volatile RAM*), ésta es una memoria no volátil que incorpora una bateria de litio para proporcionar a los datos una gran persistencia. Posee una capacidad de 32KB y está mapeada en la mitad superior del rango de direcciones de memoria local de datos. Al igual que ocurre con la EPROM, al tener una longitud de palabra de 8 bits, cada acceso del DSP sólo usará el byte menos significativo.

De la misma forma que la memoria EPROM, necesita un estado de espera para lograr llevar a cabo una transferencia, figura 6.7b. Como se citó anteriormente, este esquema introduce un estado de espera únicamente a las memorias que lo necesitan (EPROM y NVRAM), que son las mapeadas en la parte superior del espacio de direcciones de memoria de datos.

Los estados de espera prolongan la duración de la instrucción en curso, con la consiguiente penalización en tiempo de ejecución del código. Es por esto, por lo que no conviene abusar de los accesos a NVRAM (sin contar que para acceder a una palabra se han de realizar dos accesos y posteriormente formar la palabra a partir de los dos bytes obtenidos), sobre todo en procesos de alto nivel de prioridad y procesos de servicio de interrupciones.



SRAM

El sistema utiliza una memoria SRAM (RAM estática) de 16KB. Con un tiempo de acceso de 20ns de respuesta rápida, el DSP puede acceder a ella con 0 estados de espera, evitando la prolongación de los ciclos de instrucción siempre y cuando el código sea externo y los datos internos.

Se utilizan un par de éstas, obviamente de la misma capacidad, para obtener una longitud de palabra de 16 bits. Éstas almacenan los bytes menos significativo (LSB) y más significativo (MSB) independientemente. La lógica mapea una parte de la memoria SRAM, 4KW, en memoria de datos y 12KW en memoria de programa. La decodificación se realiza de la misma forma que ilustra la figura 6.7, la lógica activa la selección cuando se accede una dirección de memoria que ésta contiene.

Esta memoria soporta gran parte del código del sistema operativo y procesos de la aplicación. La parte de datos contiene las variables y estructuras de datos que no se usan muy frecuentemente.

6.5.3 Mapa de memoria del sistema

Una vez descritos los tipos de memoria que componen el mapa de memoria externo del DSP y sus características, vamos a pasar a la descripción del mapa de memoria. Nótese que el mapa de memoria interna queda definido por una determinada configuración, en nuestro caso CONF1, figura 6.6.

Como puede verse, la figura 6.9 presenta los diferentes espacios de direcciones de memoria externa. Si tomamos el **espacio de memoria de programa**, podemos ver que consta de:

- Memoria ROM interna. Memoria de sólo lectura, donde se encuentra grabado el arranque y la tabla de principal de vectores de interrupción. Nótese que esto es el efecto de configurar el DSP como microcontrolador.

- Los primeros 12KW de la SRAM se encuentran mapeados en memoria externa de programa a partir de la dirección 08000h. Esta será la encargada de soportar gran parte del código del sistema operativo y la aplicación.

- En la parte superior de los 64K direccionables se direcciona el bloque B0 de memoria interna configurado como memoria de programa. En éste se encuentra la tabla secundaria de vectores de interrupción, a la que apunta la tabla de vectores de interrupción grabada en ROM. En esta tabla secundaria los vectores pueden ser modificados libremente, por estar en RAM. También residirá en B0 un *bootloader* secundario, que es lo primero que carga el *bootloader* primario

para realizar la carga definitiva del código de la aplicación sobre el sistema, y un monitor residente para depuración si la inicialización del sistema se realiza vía serie.

Por otra parte, en el espacio de **memoria local de datos** tenemos:

- Registros mapeados en memoria interna.
- Dos bloques internos de memoria RAM, B1 y B3, alineados de 0400h a 07FFh.
- Los últimos 4KW de la SRAM se encuentran mapeados como memoria externa de datos, de 03000h a 03FFFh.
- Los 32KW superiores del espacio de direccionamiento local de memoria de datos está ocupado por la NVRAM. Como sabemos, ésta es una memoria de 8 bits de longitud de palabra, por lo que en los accesos del DSP sólo contribuirá el byte menos significativo.

El espacio de **memoria global de datos** está ocupado por una EPROM, encargada de mantener el código del sistema operativo y la aplicación entre diferentes encendidos de la calibradora. Ocupa los 32K superiores del espacio de direccionamiento, y dado que la longitud de palabra es de 8 bits, sólo se aprovecha de cada acceso el byte menos significativo.

Nótese que en la figura 6.9 no se incluye el espacio de direccionamiento de E/S, el cual es usado por la CPLD para mapear varias direcciones que ésta implementa para el control de la transferencia de los datos entre la placa CAN y el DSP, sección 6.7, y para el control de los conversores y la adquisición, sección 6.8.

6.6 Conversores ADC

El AD7730 es un solución ADC completa de Analog Devices para aplicaciones de pesado y medida de presión (AD7730, 1998). La figura 6.10 ilustra el diagrama funcional de bloques del AD7730. El dispositivo acepta señales de bajo nivel directamente de un transductor y proporciona la salida a un registro interno de datos. La señal de entrada pasa por un buffer diferencial antes de atacar al amplificador de ganancia programable, como base para el modulador Sigma-Delta (obsérvese que se dispone de un DAC de 6 bits que puede ser utilizado para eliminación de tara). La salida del modulador es procesada por un filtro digital paso-bajo programable para obtener finalmente la salida que, como ya sabemos, se almacena en el registro interno de datos, dispuesta para ser leída por un microcontrolador o microprocesador. Cuando el AD7730 actualiza el citado registro con una nueva muestra, la salida /RDY se

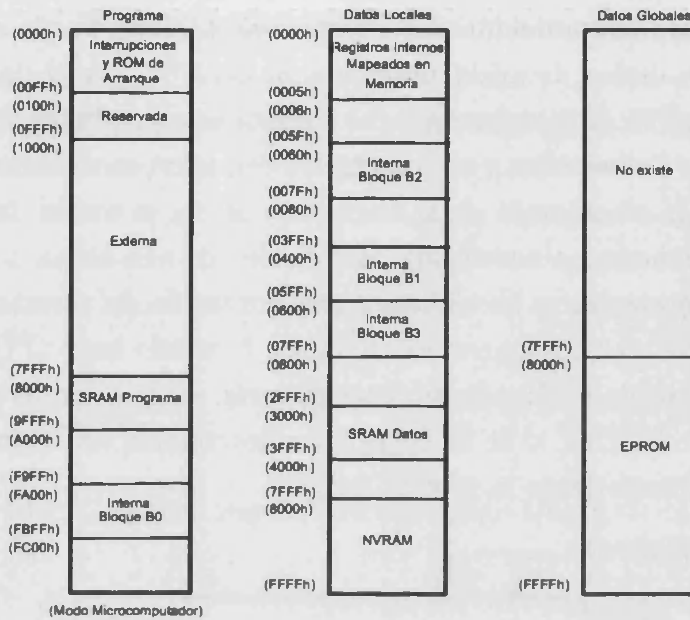


Figura 6.9: Mapa de memoria.

pone a nivel bajo hasta que la muestra sea leída o se produzca una nueva actualización del registro de datos. Así, esta señal puede ser utilizada para atacar directamente a una interrupción.

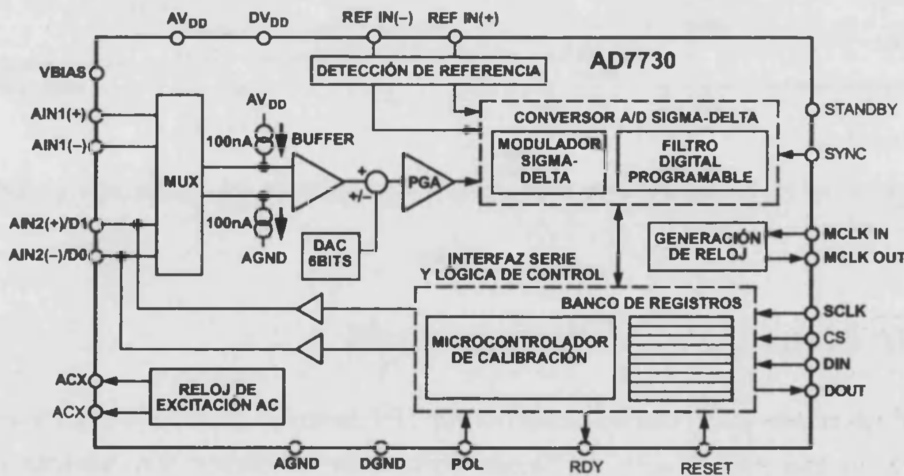


Figura 6.10: Diagrama funcional de bloques del AD7730.

El AD7730 opera con alimentación unipolar de +5VDC. Acepta rangos de entrada unipolar: 0mV a +10mV, +20mV, +40mV y +80mV y bipolar: ±10mV, ±20mV, ±40mV y ±80mV. La resolución de pico a pico directamente obtenible es de 1/230000. El DAC de 6 bits permite la eliminación de la tensión de tara. También dispone de

señales de reloj para la excitación del puente. El AD7730 puede realizar autocalibración, posee una deriva de offset de menos de 5nV/°C y una deriva de ganancia de menos de 2ppm/°C. El interfaz serie del componente puede ser configurado por la operación en modo de tres hilos y es compatible con microcontroladores. Finalmente, cabe destacar que la eliminación de la frecuencia de red es mayor de 150dB.

La figura 6.11 muestra, simplificada, la conexión de una célula de carga al dispositivo conversor. Nótese que se ha utilizado una excitación del puente en contínua. La señal de entrada en los conversores se configura a estado bajo (POL=0). Por otra parte, puede verse cómo las líneas del interfaz serie, reset, standby y sincronismo se dirigen o proceden del DSP o de la CPLD. La descripción de la implementación de este interfaz se posterga hasta la sección 6.8.

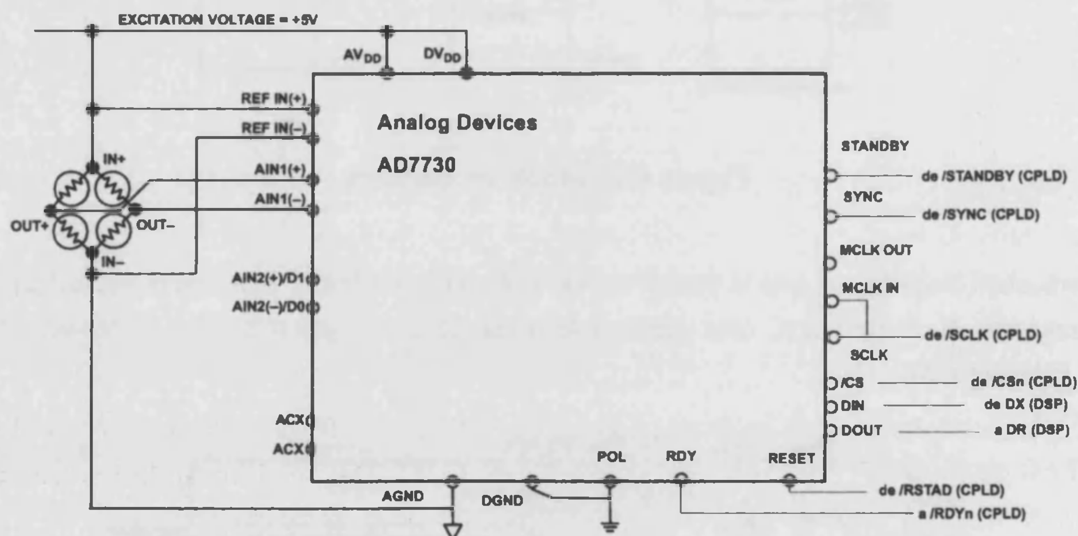


Figura 6.11: Conexión simplificada para un puente de excitación en DC.

6.7 Interfaz DSP-Conversores

El DSP no posee una comunicación serie SPI como el AD7730, sino un puerto de comunicación serie propietario de Texas Instruments. Estos son básicamente similares pero con ciertas diferencias, que imposibilitan la conexión directa DSP-AD7730. Por ello se utiliza la lógica programable para adaptar la comunicación entre ambos dispositivos.

Cabe remarcar que no se pretende una conexión punto a punto; el DSP debe estar conectado a una decena de conversores AD7730 y debe poder elegir con cual de todos desea establecer una comunicación serie. Aquí se recurre también a la lógica programable para habilitar los mecanismos que permitan al DSP tanto la comunicación

individualizada con cada uno de los conversores, como una comunicación *broadcast* a todos ellos (esta última sólo es posible para realizar una transmisión simultánea a todos ellos, útil durante la programación de los AD7730).

En el sentido de la comunicación individualizada con diferentes dispositivos, algunos DSP proporcionan, además del típico puerto serie de comunicación punto a punto, un puerto serie TDM (*Time-Division-Multiplexed*) que posibilita la conexión entre múltiples fuentes y múltiples destinos de datos a través de un bus serie común. Este formato divide cada intervalo temporal en subintervalos. Cada uno de estos representa un canal de comunicación. El número de canales de cada intervalo puede determinarse vía software. Entre los procesadores DSP que permiten esta multiplexación (para permitir esquemas de conexión más flexibles) se encuentra el TMS320C50. No obstante, éste no ha sido seleccionado para su utilización en el módulo de pesado porque sólo puede multiplexar un máximo de siete dispositivos, y porque los conversores AD7730 no son compatibles con este esquema.

Los apéndices B y C describen los detalles del funcionamiento tanto del puerto serie del DSP como del AD7730.

6.7.1 Implementación de la comunicación serie TMS320C26-AD7730

Los protocolos de comunicación serie del AD7730 y TMS320C26 son muy similares pero coinciden exactamente , véase apéndices B y C. Así, es imposible realizar una conexión vía serie entre ellos sin un mínimo de lógica.

Otro problema que se plantea es que, si bien la longitud de palabra que usa el DSP es configurable a 8 ó 16 bits, las longitudes de palabra de los registros internos del AD7730 son de 8, 16 y 24 bits. Esto plantea la imposibilidad de casar transmisiones y recepciones cuando se usan 24 bits. El interfaz diseñado para la comunicación serie debe ser lo suficientemente flexible como para que una operación de 24 bits pueda ser fragmentada en dos operaciones por parte del DSP de forma transparente al AD7730.

El último y gran problema que plantea la comunicación vía serie es que más allá de una comunicación punto a punto, el objetivo es establecer una comunicación serie del DSP con 10 dispositivos conversores. Es obvia la necesidad de establecer una comunicación independiente entre el DSP y el conversor deseado. Éste es el punto clave ya que ambos puertos han sido diseñados para una comunicación punto a punto. Nótese que una operación serie tradicional no establece información sobre el punto al que ha de ser enviada la información ni de que punto debe leerse. En este punto entramos en un conflicto, por lo que debe asumirse que no podemos usar el puerto según su operación normal.

El problema de la comunicación con múltiples dispositivos se soluciona con ayuda de la lógica programable de la CPLD, mediante la decodificación de una serie de accesos a puerto de la siguiente forma:

Transmisión

- Escribir numero n en puerto E/S 0h:
  - Si $n \geq 0$ y $n < 10$ $\longleftrightarrow$ Transmisión de 16 bits al conversor n.
  - Si $n = 15$ $\longleftrightarrow$ Transmisión de 16 bits a todos los conversores.
- Escritura de numero n en puerto E/S 1h:
  - Si $n \geq 0$ y $n < 10$ $\longleftrightarrow$ Transmisión de 8 bits al conversor n.
  - Si $n = 15$ $\longleftrightarrow$ Transmisión de 8 bits a todos los conversores.

Recepción

- Escribir numero n en puerto E/S 2h:
  - Si $n \geq 0$ y $n < 10$ $\longleftrightarrow$ Recepción de 16 bits al conversor n.
- Escritura de numero n en puerto E/S 3h:
  - Si $n \geq 0$ y $n < 10$ $\longleftrightarrow$ Recepción de 8 bits al conversor n.

La lógica interpretará a partir de los accesos mostrados, la operación que se desea realizar y el conversor (o los conversores) que se encuentra asociado a dicha operación. Así, la CPLD tomará en su mano el arbitrio de las señales de control necesarias para que la operación se produzca adecuadamente. Para ello en el DSP, se configura la transmisión a 16 bits (FO=0), los pulsos de sincronización de marco (FSM=1) para la operación del puerto serie, y se configura el pulso de sincronización de marco como una entrada (TXM=0), véase apéndice B. Nótese que esto último indica que todas las señales de control serán entradas para el DSP (CLKX, CLKR, FSX y FSR).

Por otro lado, la señal de entrada POL en los conversores se configura a estado bajo, y todas las señales de control del puerto serie (es decir, /CS y SCLK) son también entradas para el AD7730.

De esta forma, la CPLD controlará TODAS las señales de control de los conversores y del procesador, de forma que, ante una petición de operación serie, será la encargada de habilitar los mecanismos necesarios para que la operación sea realizada con éxito.

Antes de observar más detenidamente la operación serie, cabe remarcar que la CPLD divide la señal de salida CLKOUT de 10MHz, del DSP, en una señal de 5MHz que será la señal de reloj de la operación serie utilizándose como entrada tanto de

CLKX y CLKR (señales de reloj de la transmisión y recepción del puerto serie del 'C26) como de SCLK (señal de reloj del interfaz serie del A7730). Nótese que si esta salida fuera conectada a las 12 entradas citadas, se produciría una carga excesiva, sobre todo para una señal de reloj. Por ello, la señal dividida se usa como entrada a un buffer del que se obtienen diferentes salidas de reloj similares, que ya pueden atacar a las entradas correspondientes sin problemas de carga. La frecuencia de 5MHz en la operación de la comunicación serie se ha elegido por ser la mayor frecuencia de funcionamiento del puerto serie del DSP. A partir de este momento podremos hablar indistintamente de SCLK, CLKX y CLKR, ya que aquí todas ellas son la misma señal.

Para poder operar con cada conversor por separado, ha de poderse seleccionar individualmente cada conversor, y también detectar su línea /RDY a estado bajo, lo cual significará una interrupción para el DSP. Para ello, la CPLD toma como salidas las señales de selección de los conversores, /CS0 a /CS9, y como entradas las líneas READY de estos, /RDY0 a /RDY9. Estas últimas serán pasadas a través de una puerta AND de diez entradas para generar la entrada de la interrupción del conversor del DSP. Nótese que, de esta forma, el DSP puede identificar el conversor que lo ha interrumpido inspeccionando su estado. No obstante, como en funcionamiento normal todos los conversores están programados con la misma frecuencia de muestreo y además funcionan sincronizados, todos los conversores actualizaran simultáneamente el registro de salida con el dato obtenido de la conversión, dentro de un margen mínimo equivalente a un ciclo de reloj master del conversor.

Veamos ahora el mecanismo establecido por la CPLD para la comunicación. Para ello, nótese que tanto las recepciones como las transmisiones necesitan un pulso de sincronización de entrada (en FSR o FSX respectivamente, apéndice B). Consideremos como ejemplo el caso de una transmisión de 16 bits al conversor 2. El mecanismo para realizar dicha transmisión es el siguiente:

- Se almacena en el registro de transmisión DXR el dato que se pretende transmitir.

- Se realiza una escritura del valor 2 al puerto 0h de E/S. Esto es decodificado por la CPLD, que entiende que el DSP desea generar una transmisión de 16 bits al conversor 2.

- Al haber decodificado en el paso anterior la necesidad de una transmisión de 16 bits, la CPLD se dispone a gestionar los mecanismos necesarios para que ésta ocurra con éxito. Para ello, en el siguiente flanco ascendente de SCLK (nótese que SCLK,CLKX y CLKR son la misma señal) la CPLD pone FSX a estado alto durante un ciclo de la señal de reloj serie SCLK, figura 6.12, generando así

el pulso de sincronización de marco que indica la transmisión desde el puerto serie del DSP. A partir del siguiente flanco ascendente de SCLK, el DSP sacará un nuevo bit a la línea DIN que será recogido e introducido en el registro de desplazamiento del AD7730 en los flancos descendentes de SCLK (por ello se ha configurado POL a nivel bajo). En este proceso el DSP envía primero el bit más significativo, que también es recogido por el AD7730 como bit más significativo.

- No obstante, aunque la CPLD desencadena el proceso desde el punto de vista del DSP, también debe seleccionar el conversor que debe recoger el dato. Para ello activa, durante la transmisión, la señal de selección que en nuestro caso corresponde al conversor 2, /CS2. Nótese que esta señal debe comenzar después del siguiente flanco descendente de SCLK que sigue a la subida del pulso de sincronización de marco para que el primer bit enviado sea recogido con éxito, véase figura 6.12. Por otra parte, la finalización de la señal de selección debe ocurrir tras el flanco descendente de SCLK, que señala la recogida del último bit del marco, y antes de un nuevo flanco ascendente.

El mecanismo descrito inicia una transmisión de 16 bits al conversor seleccionado. Nótese que dicha transmisión sería del tipo *broadcast*, a todos los conversores, si se hubiese utilizado el valor 15 en la escritura al puerto 0h. El cronograma de la figura 6.12 ilustra esta transmisión.

En el caso de las transmisiones de 8 bits, la operación es similar a la ilustrada en el cronograma de la figura 6.12, no obstante el inicio de esta transmisión se realiza mediante un acceso al puerto 1h. En este caso debe tenerse en cuenta que se debe configurar el 'C26 para una longitud de palabra de 8 bits antes de la operación, y volver a restablecer la configuración para longitudes de 16 bits tras el envío. Como vemos, la señal de selección /CS se ajusta a una transmisión de 8 bits.

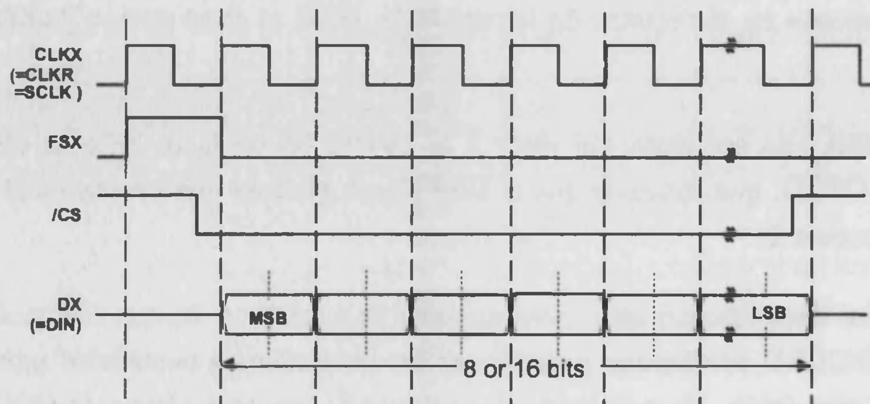


Figura 6.12: Escritura en el AD7730.

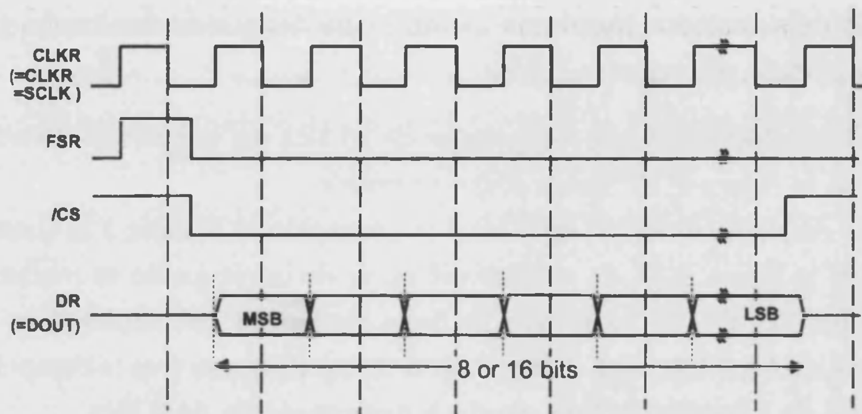


Figura 6.13: Lectura del AD7730.

El mecanismo que habilita la recepción de datos (desde el punto de vista del DSP) también es gestionado por la CPLD a partir de una petición a un puerto de E/S. El cronograma de esta operación es muy similar, pero con sutiles diferencias, por lo que volverá a ser expuesto. Para ello vamos a considerar el ejemplo de una recepción de datos de 16 bits desde el conversor 1. El mecanismo es el siguiente:

- Se realiza una escritura del valor 1 al puerto 2h de E/S. Esto es decodificado por la CPLD, que entiende que el DSP desea realizar una recepción de 16 bits del conversor 1.

- Al haber decodificado la necesidad de una recepción de 16 bits, la CPLD se dispone a habilitar los mecanismos necesarios para que ésta ocurra. Para ello, en el siguiente flanco ascendente de SCLK pone FSR a estado alto durante *tres cuartos de la señal de reloj serie SCLK*, figura 6.13. Esta generación se basa tanto en la señal SCLK, dividida de CLKOUT, como en la propia señal CLKOUT. Así, se genera el pulso de sincronización de marco que indica el comienzo de la recepción. A partir del siguiente flanco ascendente de SCLK, el AD7730 pondrá un nuevo bit en DOUT que será recogido e introducido en el registro de desplazamiento del puerto de recepción del 'C26 en los flancos descendentes de SCLK (ello se ha configurado la entrada POL a nivel bajo). En este proceso el AD7730 envía primero el bit más significativo, que también es recogido por el 'C26 como bit más significativo.

- La CPLD activa la selección del conversor que debe enviar el dato, en este caso /CS1, durante el flanco descendente de FSR, figura 6.13. La finalización de la señal de selección debe ocurrir tras el flanco descendente de SCLK, que señala el envío del último de los 16 bits, y antes de un nuevo flanco ascendente.

- El DSP debe esperar hasta que la recepción haya sido finalizada para poder recoger el dato del registro DRR.

El mecanismo descrito inicia recepciones de 16 bits del conversor seleccionado. El cronograma de la figura 6.13 ilustra dicha recepción.

En el caso de las recepciones de 8 bits, la operación es similar a la ilustrada en el cronograma de la figura 6.13, no obstante el inicio de la recepción se realiza mediante un acceso al puerto 3h. En este caso, se debe configurar previamente en el DSP la longitud de palabra a 8 bits, que deberá volver a restablecerse tras la recepción. Como vemos, la señal de selección /CS se ajusta a una recepción de 8 bits.

Recapitulando, se observa cómo la CPLD gestiona el control de la operación serie multipuerto, seleccionando el conversor requerido para la comunicación con el DSP. El modo de transmisión utilizado es de operación por paquetes (*burst-mode*). Sin embargo, el software DSP debe prevenir una nueva petición hasta que cualquier transferencia en curso haya finalizado. Este control lo establece, de forma transparente al usuario, el microkernel de sistema operativo que será descrito en el capítulo 7.

Inicialmente se planteó el problema de las transferencias de 24 bits. El interfaz realiza una implementación tan flexible de las transferencias, que una operación de 24 bits puede fragmentarse en dos operaciones consecutivas de 16 y 8 bits respectivamente, o incluso en 3 operaciones de 8 bits.

Además de los mecanismos hardware descritos, utilizados para posibilitar transferencias entre DSP y cualquiera de los conversores, ha sido necesario habilitar otros mecanismos hardware que actúen sobre el estado global de estos. Dado que el DSP carece de puertos con salidas mantenidas, la opción es, una vez más, hacer que sea la CPLD la que, decodificando los accesos a puerto, habilite dichos mecanismos. Estos son:

- **Reset.** Se decodifica el puerto 4h de E/S. Un acceso, tanto de escritura como de lectura, a este puerto activa a nivel bajo la señal /RSTAD, salida de la CPLD, conectada a todas las entradas /RESET de los conversores. Esta salida permanece activa 100ns, permitiendo superar el tiempo necesario de 50ns para que se complete la operación.

  El reset de los conversores inicializa su lógica interna, filtro digital y modulador analógico, mientras todos los registros son forzados a volver a sus valores por defecto. Durante este tiempo la línea /RDY es conducida a alto y el AD7730 ignora todas las comunicaciones a cualquiera de los registros mientras esta se encuentra activa a nivel bajo. Cuando la entrada /RESET vuelve a estado alto,

es necesario volver a configurar registros y llevar a cabo toda la calibración tras un comando /RESET.

- **Sincronización.** Se decodifica el puerto 5h de E/S. Un acceso, tanto de escritura como de lectura, a este puerto activa a nivel bajo la señal /SYNC, salida de la CPLD, conectada a todas las entradas /SYNC de los conversores. Esta salida permanece activa 100ns, permitiendo superar el tiempo necesario de 50ns para que se complete la operación. La entrada /SYNC reinicializa el modulador y el filtro digital sin afectar a ninguna de las condiciones del componente. Esto permite comenzar tomando muestras de la entrada analógica a partir de un punto conocido en el tiempo, el flanco de subida de la señal /SYNC.

  Ante la sincronización de los conversores (programados similarmente en cuanto a frecuencia de muestreo, etc.), las actualizaciones del registro de salida ocurrirán sincronizadas con una posible diferencia máxima, entre las actualizaciones de salida de los AD7730 individuales, de un ciclo del reloj MCLK IN, reloj master del AD7730.

- **Standby.** Se decodifica el puerto 6h de E/S. Un acceso, tanto de escritura como de lectura, a este puerto conmuta el nivel lógico de la señal /STANDBY, salida de la CPLD, conectada a todas las entradas /STANDBY de los conversores. Tras el arranque, esta salida se inicia a nivel alto. Tras cada acceso al puerto indicado, la salida conmuta y permanece indefinidamente hasta que otro acceso conmute su estado.

  La señal /STANDBY permite situar al AD7730 en modo de bajo consumo de energía, **powerdown**, cuando éste no es requerido para proporcionar resultados de conversión. El AD7730 retiene los contenidos de todos los registros internos (incluyendo el registro de datos) mientras se encuentra en este modo. Cuando /STANDBY vuelve a nivel alto, el componente vuelve a la operación que ha estado realizando antes de que la señal pasara a nivel bajo. La entrada /STANDBY no afecta al interfaz digital. Ésta, sin embargo, coloca la señal /RDY a estado alto. Cuando el /STANDBY vuelva de nuevo a nivel alto, /RDY permanecerá a nivel alto hasta una nueva conversión o calibración.

6.8 Interfaz DSP-Placa de comunicaciones

Es sabido que la placa de comunicaciones y el DSP intercambian mensajes, byte a byte, en ambas direcciones y en paralelo. Sin embargo, supóngase que el DSP posee un byte que debe enviar a la placa de comunicaciones, si éste avisara activando una

interrupción, el DSP tendría que esperar durante un tiempo, la latencia de interrupción del microcontrolador de la placa de comunicaciones, para que éste lo aceptara. Éste no es un planteamiento acertado, ya que ocupa a cualquiera de las partes que pretende transmitir hasta que la otra pueda responder, aceptando el byte.

El planteamiento implementado introduce un buffer intermediario en el proceso de comunicación en ambas direcciones. Un solo registro intermedio para recepción y otro para transmisión incrementa notablemente la agilidad y la eficiencia de la comunicación entre las partes.

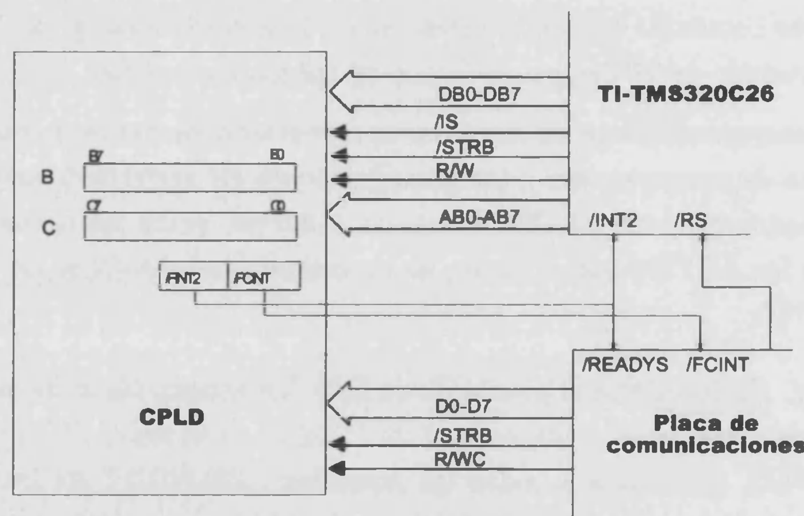


Figura 6.14: Esquema del mecanismo de buffer intermedio implementado por la CPLD.

En el diseño del *buffering* intermedio se ha evitado la introducción de componentes discretos, encapsulando toda la lógica y registros necesarios en la CPLD. El *buffering* intermedio utiliza tres registros, dos registros de 8 bits, y otro registro de 2 bits. El denominado registro B es el utilizado para la recepción, es decir, la comunicación en dirección al DSP. El registro C es utilizado para la transmisión, es decir, en la comunicación en dirección a la placa de comunicaciones. El registro de dos bits contiene dos flags denominados /FCINT y /FINT2, figura 6.14.

Como puede verse en la figura 6.14, la CPLD saca permanentemente el contenido de estos flags a través de los pines configurados como salida que llevan su mismo nombre. Así, /FINT2 se conecta a la entrada de READY de la placa de comunicaciones y a la interrupción /INT2 del DSP. Nótese que existe un jumper con el que se selecciona si dicha interrupción va a ser ocupada por el puerto serie de un PC host o por el interfaz con la placa de comunicaciones. Por otra parte /FCINT está conectado a una interrupción de la placa de comunicaciones.

La CPLD mantiene estos registros mapeados en el espacio de E/S del DSP con la

siguiente correspondencia:

- Puerto 07h ⟷ Registro de flags.
- Puerto 08h ⟷ Registro B.
- Puerto 09h ⟷ Registro C.

Así, estos registros pueden ser accedidos tanto para lectura como para escritura. Cuando se accede a los registros B o C, se desecha el byte más significativo de la palabra de 16 bits. Cuando se accede al registro de flags, únicamente contribuyen los bits menos significativos, siendo /FINT2 el bit 1 y /FCINT el bit 0.

El registro C y el flag /FCINT son usados en el proceso de transmisión. Para describir este mecanismo asumamos inicialmente /FCINT a nivel alto. Si el DSP lee el registro de flags y éste está a nivel alto quiere decir que la placa de comunicaciones no tiene byte pendiente de lectura y, por lo tanto, puede escribir en el registro C el siguiente byte a transmitir. Entonces, el DSP escribe en el puerto 09h una palabra cuyo LSB byte es el dato que pretende transmitir. Esta escritura es decodificada por la CPLD de forma que el dato, LSB byte, se escribe en el registro C, y activa el flag /FCINT a nivel bajo. De esta forma, se activa la interrupción que indica a la placa de comunicaciones que tiene un byte por leer en el buffer intermedio. Antes de escribir otro valor, el DSP debe asegurarse de que el flag /FCINT no está a nivel bajo. Cuando la placa de comunicaciones accede al buffer intermedio para lectura, toma el valor del registro C y automáticamente /FCINT se pone a nivel alto, desactivando la interrupción de la placa de comunicaciones y señalando al DSP que pueda seguir enviando información. Como vemos, los accesos al registro C se realizan en paralelo desde ambas partes tanto para lectura como para escritura.

El registro B y el flag /FINT2 son usados en el proceso de recepción. Para describir el mecanismo asumamos inicialmente /FINT2 a nivel alto. Cuando la placa de comunicaciones quiere enviar un byte al DSP, inspecciona la señal READY, dado que la placa de comunicaciones tiene expuestos los dos flags y puede inspeccionarlos sin acceder al registro de flags. Si READY está a nivel alto indica que puede enviar el byte. Entonces, la placa de comunicaciones escribe un dato al buffer intermedio, con lo que éste se escribe en el registro B (las escrituras de la placa de comunicaciones al buffer se producen en el registro B y las lecturas se obtienen del registro C) el byte que desea transmitir. Esta escritura es decodificada por la CPLD escribiendo el byte en el registro B y activando automáticamente el flag /FINT2 a nivel bajo. De esta forma, se activa la interrupción que indica al DSP que tiene que recoger un dato, al mismo tiempo que se desactiva la señal de READY por la cual la placa de comunicaciones no puede escribir otro byte antes de que READY pase a nivel alto. Cuando el

DSP lee el dato del registro B, la CPLD pone automáticamente a nivel alto /FINT2, desactivando la interrupción y permitiendo a la placa de comunicaciones que pueda seguir enviando información.

Anteriormente se ha citado que existe un jumper con el que seleccionar si la interrupción /INT2 del DSP va a ser ocupada por el puerto serie de un PC host o por el interfaz con la placa de comunicaciones. Por defecto, el sistema usa el jumper colocado para su uso por el debugger de un PC host, ya que el polling periódico da por sí solo buenas prestaciones.

6.9 Interfaz de comunicación RS232 a PC *host*

El TMS320C26 se diferencia del resto de su familia TI-TMS320C2X de procesadores digitales, en que lleva grabado en la PROM interna un sencillo código que le permite la carga del programa y el arranque del sistema a través de tres mecanismos diferentes, cuando funciona en modo microcontrolador. Uno de ellos es la carga vía serie utilizando los pines /BIO y XF. La tarjeta de pesado soporta esta opción cuando se elimina la EPROM de su zócalo, ya que el arranque desde EPROM es prioritario.

Así, se proporciona una opción de arranque a partir de una CPU host que, conectada a la tarjeta de pesado a través de un cable serie DB9, ofrece posibilidades tanto de carga como de depuración. Durante la carga, el *host* también envía al módulo de pesado un monitor que residirá permanentemente en el bloque B0 de memoria interna. La plataforma *host* dispone de un software de depuración (*debugger*) con el que se puede interaccionar con el monitor residente. De esta forma se puede obtener información sobre:

- Contenidos de memoria de datos.
- Flags de estado.
- Registros internos.
- Registros mapeados, etc.

Como el *host* y el sistema están conectados en todo momento mediante el cable serie, el *debugger* permite realizar interacciones con el sistema embebido. Mediante éste podemos realizar:

- Carga del SO y la aplicación.
- Ejecución.
- Ejecución paso a paso.

- Modificación de registros.
- Puntos de ruptura en la ejecución del programa,etc.

Tanto las peticiones como la visualización se realiza interactivamente en la CPU host, que invoca a través de la interrupción hardware /INT2 al monitor residente para ejecutar la acción pedida (siempre que esté colocado el jumper correspondiente a la comunicación RS232 serie). Durante la comunicación vía RS232, el DSP usa niveles TTL estándar en los pines /BIO y XF. Para realizar la conversión entre niveles RS232 y TTL, el pin /BIO recibe el dato de un receptor de linea RS232, y el pin XF envía la información vía driver de línea RS232, figura 6.15.

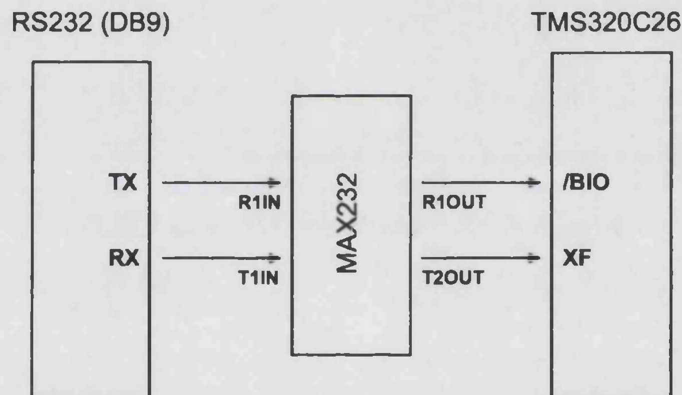


Figura 6.15: Comunicación RS2323 con el DSP.

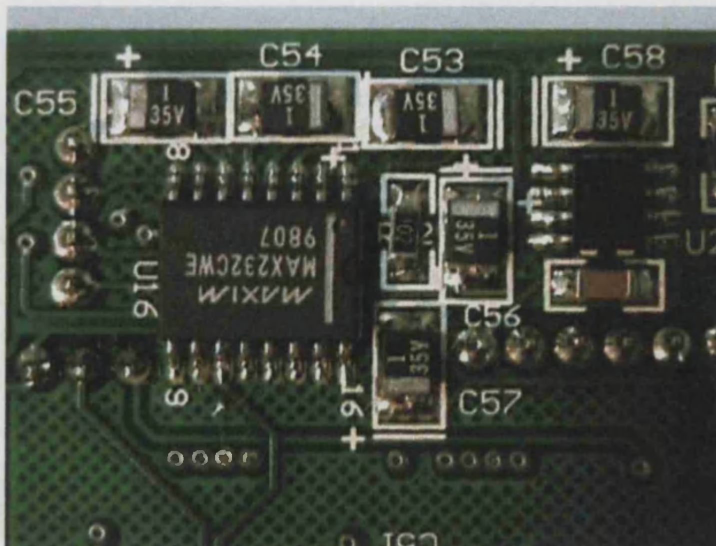


Figura 6.16: Detalle de la comunicación RS2323.

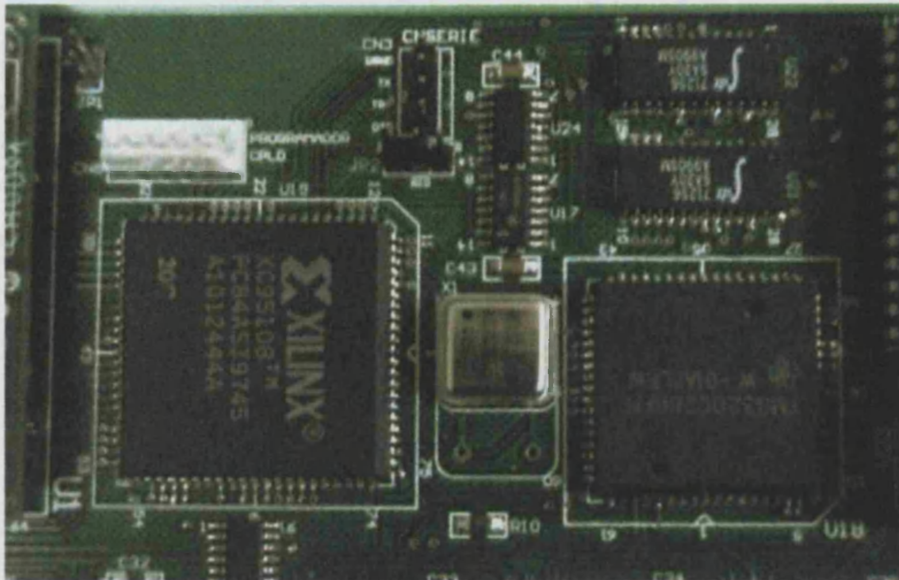


Figura 6.17: Detalle del 'C26 y la CPLD.

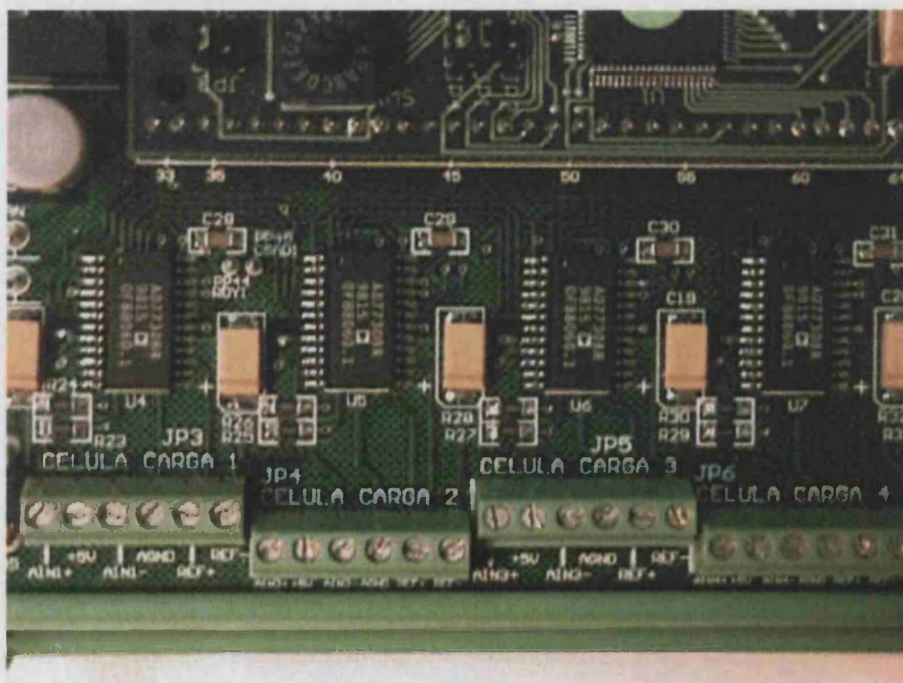


Figura 6.18: Detalle de la zona de los conversores AD7730.

Capítulo 7

El sistema operativo EMMOS

En un sistema embebido basado en DSP, la necesidad de un sistema operativo (SO) depende de las tareas que se realicen. En tareas simples donde se ejecuta siempre una función simple de manera repetitiva, la utilización de sistemas operativos puede suponer un *overhead.* Sin embargo, cuando el número y la complejidad de las tareas del DSP se incrementa, resulta ventajoso el uso de un sistema operativo para asignar los recursos del sistema y permitir multitarea.

El uso de un sistema operativo básico de tiempo real permite al DSP soportar un gran número de interrupciones, junto con una combinación de funciones de control y procesado. Es decir, por un lado realizará un tratamiento en tiempo real de los datos, y por otro lado, manejará funciones de control, E/S, etc. que son funciones a nivel de sistema y por lo tanto de nivel superior.

Este capítulo está dedicado a la descripción del sistema operativo EMMOS (*Embedded Multitasking Microkernel Operating System*), por lo que el presente capítulo está estructurado según los diferentes elementos funcionales que forman el SO.

7.1 Introducción

En general, un sistema operativo se puede considerar como una colección organizada de software que extiende al hardware y que consta de rutinas de control para operar una computadora y proporcionar un entorno para la ejecución de programas. Los programas utilizan las facilidades proporcionadas por el sistema operativo para acceder a los recursos del sistema de la computadora, tales como archivos, dispositivos de entrada/salida (E/S), etc. Los programas invocan los servicios del sistema operativo mediante *llamadas del sistema.* Además, los usuarios pueden interactuar

con el sistema operativo directamente mediante las *órdenes del sistema operativo*. En ambos casos, el sistema operativo actúa como una interfaz entre los usuarios y el hardware de un sistema de computadoras (Shay, 1997).

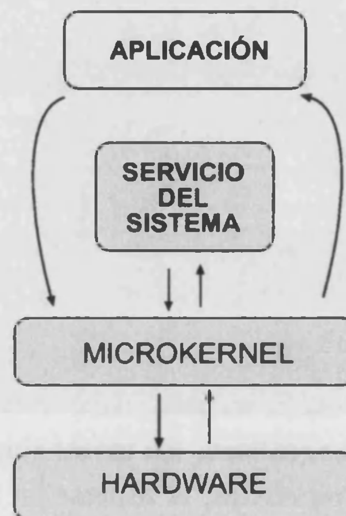


Figura 7.1: La aplicación hace uso del SO para acceder a los recursos del sistema.

Sin embargo, el número de servicios que proporciona un sistema operativo y su complejidad dependen de un buen número de factores, entre otros las necesidades y características del entorno las que determinan en gran medida las funciones de un sistema operativo. Por ejemplo, un sistema operativo de escritorio, basado en ventanas y destinado al desarrollo de programas en un entorno interactivo puede tener una complejidad y un conjunto de llamadas del sistema y órdenes, muy diferentes a las de un sistema operativo diseñado para soportar en tiempo real una aplicación dedicada, como el control del motor de un coche.

7.1.1 Sistemas operativos serie y multiprogramados

Un sistema operativo puede realizar su trabajo en serie o concurrentemente. Es decir, se pueden dedicar todos los recursos de un sistema a un programa único hasta completarlo, *sistema operativo serie*, o se pueden reasignar dinámicamente entre una colección de programas activos en diferentes estados de ejecución. Estos últimos sistemas operativos son conocidos usualmente como *sistemas multiprogramados* por su capacidad de ejecutar múltiples programas de manera intercalada y, obviamente, poseen un mayor grado de complejidad (Milenkovic, 1988; Tanenbaum, 1992).

Procesamiento en serie

La mayoría de los programas en curso de ejecución oscilan entre fases de computación intensiva y de E/S intensiva. Esto se indica en la figura 7.2a, donde el esfuerzo de cálculo intensivo se indica con rectángulos sombreados y las etapas de E/S, en las cuales la CPU está infrautilizada, con los rectángulos blancos punteados.

La ejecución serie idealizada se desarrolla en la figura 7.2b. Como puede verse se supone, con vistas a la comparación con la multiprogramación, que ambos tipos de ejecución tienen idéntico comportamiento con respecto a tiempos de procesador y de E/S y a sus distribuciones relativas.

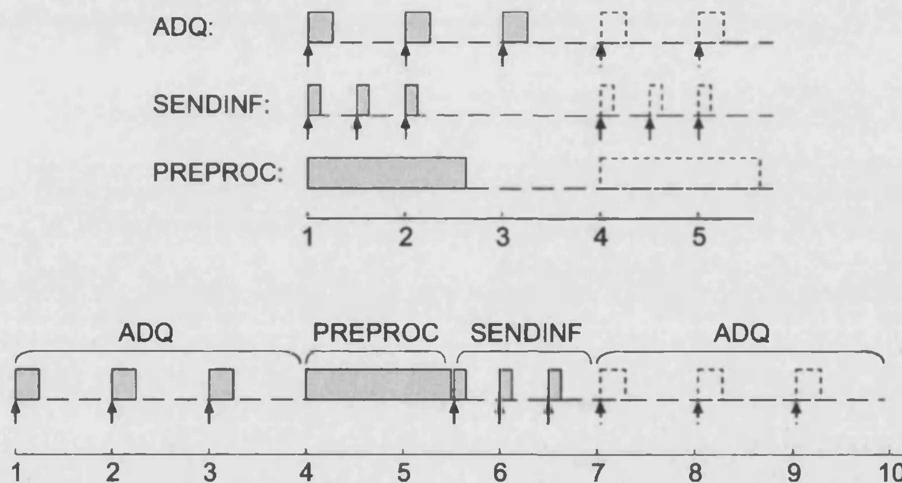
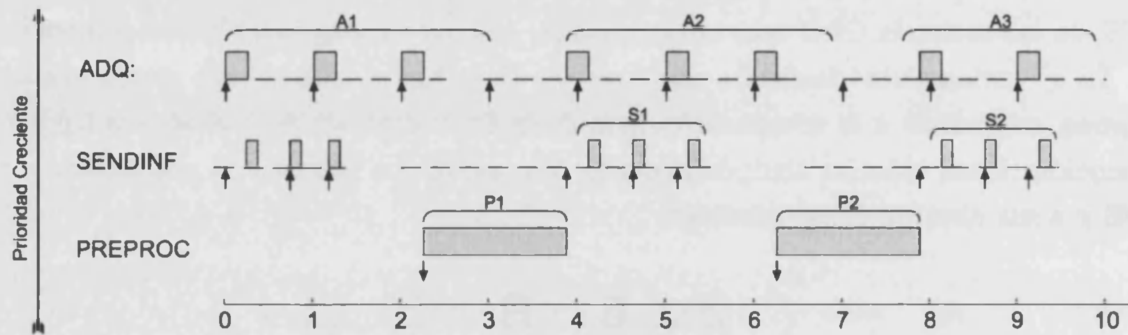


Figura 7.2: a) Fases de ejecución de varios programas. b) Ejecución serie de estos.

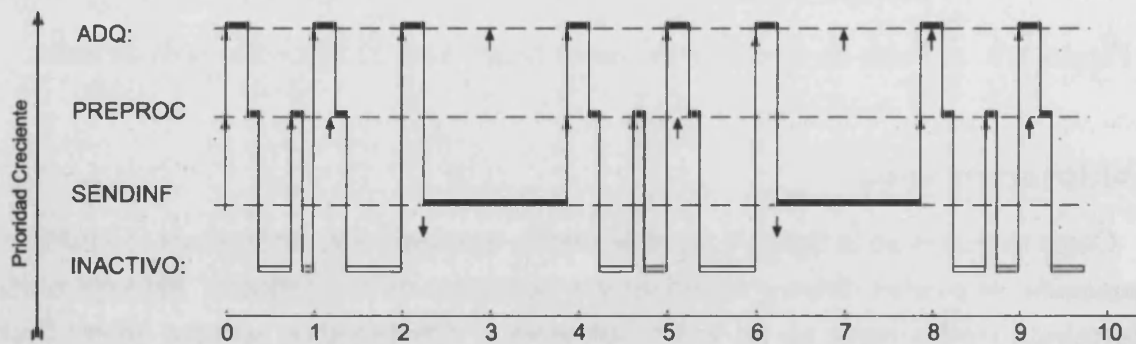
Multiprogramación

Como se ilustra en la figura 7.3a, si se realiza una ejecución intercalada, o *multiprogramación*, se pueden obtener significativas ganancias de rendimiento. Esta ejecución intercalada tendrá lugar en un único procesador. Obviamente, esto no quiere decir que éste pueda ejecutar los dos programas a la vez. Sólo un único programa puede estar bajo el control del procesador a la vez dado que la ejecución paralela de programas no es posible con un procesador simple. Lo que el ejemplo de la figura 7.3a sugiere es la utilización al 100% del procesador alternando ambos programas. Así, cuando un programa alcanza una etapa de E/S, en lugar de infrautilizar la CPU con una espera, se puede utilizar ésta para desarrollar una etapa computacionalmente intensiva del otro programa, y así sucesivamente.

La figura 7.3a ilustra como la ejecución intercalada de los programas es más eficiente que la ejecución serie. Nótese como en 7.3a sólo se pierde uno de cada cuatro



a)



b)

Figura 7.3: a) Ejecución concurrente. b) Diagrama *prioridadtiempo proceso*

eventos de adquisición (25%) contra 3 de cada 6 que pierde la ejecución serie (50%). Esta eficiencia se puede incrementar mediante una cuidada selección de la política de planificación. En la figura 7.3b se ilustra un diagrama ***prioridad - tiempo de proceso*** que representa básicamente la misma información que la figura 7.3b. A cada programa se le asigna una prioridad, el diagrama muestra el tiempo de ocupación de la CPU para cada una de las prioridades.

7.1.2 Sistemas operativos de multiprogramación

La ejecución de programas como se ilustró en la figura 7.3, se denomina *ejecución concurrente.* Esta ejecución posee un significativo potencial de mejora del rendimiento total del sistema y la utilización de recursos con respecto a la ejecución serie de los programas. Este potencial se realiza, o al menos se explota, mediante una clase de sistemas operativos que multiplica los recursos del sistema entre una multitud de programas activos; tales sistemas operativos tienen normalmente en sus nombres el prefijo *multi*, como multitarea o multiprogramación.

Se denomina *proceso* o *tarea* a un caso de un programa en ejecución. Un sistema operativo *multiproceso,* también llamado *multitarea*, se distingue por sus habilidades para soportar dos o más procesos activos simultáneamente. El término *multiprogramación* denota un sistema operativo que, además de soportar múltiples procesos concurrentes, permite que residan simultáneamente en memoria las instrucciones y los datos procedentes de dos o más procesos disjuntos (Milenkovic, 1988).

En general, todos los sistemas de multiprogramación se caracterizan por una multitud de programas activos simultáneamente que compiten por los recursos del sistema, incluyendo la unidad central de proceso (CPU), la memoria y los dispositivos de E/S. El sistema operativo monitoriza el estado de todos los programas activos y recursos del sistema, y se activa para asignar recursos y proporcionar ciertos servicios de su repertorio cuando ocurre un cambio de estado importante, o cuando es llamado explícitamente.

Cabe señalar que la multiprogramación implica la operación de multiproceso, pero la operación de multiproceso (o multitarea) no implica multiprogramación.

Como veremos, los requerimientos del entorno específico que se va a servir influyen en la elección de los objetivos y las estrategias del sistema operativo asociado.

Sistemas de tiempo compartido

Son sistemas multiacceso dedicados típicamente a la ejecución esencial de un programa único de una gran aplicación. Sirva como ejemplo típico una aplicación bancaria con cientos de terminales funcionando bajo el control de un programa único y

accediendo a una base de datos común.

La filosofía de este sistema operativo se refleja en la elección del algoritmo de planificación. El sistema operativo define una fracción de tiempo que utiliza para interrumpir la ejecución de un proceso cuando éste está en posesión de la CPU y lo pone al final de la cola de procesos en espera. Cuando el tiempo de ejecución de un proceso es mayor que la fracción de tiempo definida por el sistema, el proceso en ejecución se suspende Esta suspensión previene la monopolización del procesador por un único proceso. Este modo de operación proporciona, generalmente un tiempo de respuesta rápido a los programas interactivos.

Sistemas de tiempo real

En aplicaciones industriales, como es el caso que nos ocupa, se usan generalmente los denominados ***sistemas operativos de tiempo real.*** Éstos se caracterizan por su utilización en entornos donde se deben aceptar y procesar en breve tiempo y sin tiempos muertos un gran número de sucesos, en su mayoría externos al sistema. Este tipo de sistemas operativos se usa en telefonía, control de vuelo, simulaciones en tiempo real, etc.

El objetivo primario de un sistema operativo de tiempo real es proporcionar tiempos rápidos de respuesta y así hacer frente a los tiempos muertos de planificación. En éstos, son asuntos secundarios la conveniencia del usuario y/o la utilización de recursos. No es raro para este tipo de sistemas operativos esperar el proceso súbito de miles de interrupciones por segundo ***sin perder un solo suceso.*** Este último es un requerimiento indispensable de las aplicaciones que estos sistemas soportan. Estos requerimientos no pueden abordarse, normalmente, únicamente con la multiprogramación, por lo que los sistemas operativos en tiempo real (RTOS, *Real Time Operating Systems*) usualmente utilizan ciertas estrategias y técnicas específicas para hacer su trabajo.

En sistemas de tiempo real, se encuentran con cierta frecuencia procesos definidos por el programador. Básicamente, se encarga a un proceso separado de manejar un único suceso externo. Este proceso se activará al ocurrir el suceso relacionado, indicado frecuentemente por una interrupción. El multiproceso se consigue planificando los procesos independientemente unos de otros. A cada proceso se asigna un cierto nivel de prioridad que corresponde a la importancia relativa de los sucesos que sirve. Así, el planificador asigna normalmente la CPU al proceso con más alta prioridad de entre los que están listos para ejecutarse, por lo que los procesos de más alta prioridad toman por derecho la ejecución de los procesos de prioridad inferior. Esta forma de planificación, llamada planificación ***basada en la prioridad preferente,*** es la usada por

EMMOS y la mayoría de los sistemas en tiempo real (Tanenbaum, 1992; Milenkovic, 1988).

Usualmente, y dependiendo del entorno en el que operen, estos sistemas operativos no disponen de ciertos módulos o elementos presentes en cualquier sistema operativo de sobremesa. Un ejemplo de ello es el gestor de archivos, que se encuentra normalmente sólo en grandes instalaciones de sistemas en tiempo real, dado que sistemas reducidos en tiempo real, como un controlador autopropulsado a bordo en aviónica, pueden, y es lo lógico, no tener ningún tipo de almacenamiento secundario.

Otra característica de los sistemas en tiempo real es que el gestor de memoria está comparativamente menos solicitado que en otros tipos de sistemas de multiprogramación. La razón principal es que muchos procesos residen permanentemente en memoria para proporcionar rápidos tiempos de respuesta. Al contrario que otros tipos de sistemas operativos, p.ej. tiempo compartido, la población de procesos se caracteriza por su localización casi estática y hay comparativamente poco movimiento de programas entre el almacenamiento primario y secundario. Por otra parte, los procesos en sistemas de tiempo real tienden a cooperar estrechamente, así necesariamente soportan la separación y compartición de la memoria.

Además de las formas sofisticadas disponibles de gestión de interrupciones y almacenamiento intermedio, los sistemas operativos en tiempo real proporcionan frecuentemente llamadas del sistema que permiten a los procesos (programas) de usuario conectarse directamente a vectores de interrupción y sucesos de servicio.

Sistemas operativos combinados

De lo indicado hasta ahora se desprende que cada tipo de sistema operativo está optimizado, o al menos muy acoplado, a las necesidades de sus entornos específicos. Sin embargo, en la práctica, un entorno dado puede no encajar exactamente en ninguno de los tipos de SO existentes, por lo que algunos sistemas operativos comerciales proporcionan una combinación de los servicios descritos. Sirva como ejemplo, el caso de un sistema que proporcionan servicios de tiempo compartido a sus usuarios de terminales y está conectado en red; en este caso, puede utilizarse un sistema operativo de tiempo compartido que soporte algunos sucesos de tiempo crítico como la recepción y transmisión de paquetes de datos en una red; propio de sistemas de tiempo real.

7.1.3 Requerimientos funcionales

En el transcurso del presente capítulo se describe la funcionalidad y funcionamiento del sistema operativo de tiempo real EMMOS, creado para su funcionamiento en el subsistema pesado de la calibradora. EMMOS es una abreviatura de *Microker-*

nel Embebido de Sistema Operativo Multitarea (Embedded Multitasking Microkernel Operating System).

En primer lugar se describen las estructuras: buffers circulares, colas de mensajes, listas, etc. Las estructuras de datos pueden ser privadas, cuando su gestión la realiza el sistema operativo de forma transparente al usuario, y públicas, cuando son declaradas en los procesos para que puedan ser usadas por éstos. Posteriormente se tratará la gestión de E/S que, al contrario de un sistema operativo convencional, no consiste en ficheros, display, teclado,... sino exclusivamente en mensajes CAN.

Los procesos son los elementos que hacen funcionar al sistema. Estos realizan peticiones de los servicios y recursos del sistema en el curso de su evolución hasta completarse. Esta sección describirá el concepto de proceso y describe el procedimiento que hay que realizar desde la creación de los procesos durante el desarrollo de la aplicación hasta la carga de la aplicación en el sistema.

A lo largo del capítulo se introducirá el concepto de planificación. Éste será desarrollado finalmente en la sección dedicada a la planificación, que describe posteriormente el tipo de planificación elegido en EMMOS para agilizar los cambios de contexto, eliminando los descriptores de los sistemas operativos convencionales.

Finalmente se revisan algunas cuestiones relacionadas con EMMOS y el grado de gestión de memoria y mecanismos de sincronización que éste implementa.

7.2 Estructuras de datos

EMMOS posee una serie de estructuras privadas de datos, como tablas, colas de mensajes, etc. Estas estructuras se alojan en memoria principal entre otros motivos porque el subsistema de pesado dispone únicamente de almacenamiento primario, esto es, de una determinada cantidad de memoria principal y ningún tipo de almacenamiento secundario.

Sin embargo, por razones de velocidad, volatilidad, etc., el mapa de memoria está constituido por diferentes tipos de memoria. A continuación estudiaremos distintas variables y estructuras de datos clasificadas por el tipo de memoria que la soporta.

7.2.1 EPROM: Las tablas de inicialización

Dado que el sistema no dispone de almacenamiento secundario, existe una memoria EPROM para almacenar el código del sistema operativo más la aplicación. Ésta, como el almacenamiento secundario, es un almacenamiento persistente de la información pero de acceso más lento que el acceso al almacenamiento primario. No obstante,

la memoria EPROM es de sólo lectura pudiendo únicamente escribirse mediante un grabador adecuado.

La utilización de la EPROM implica la persistencia de la información aunque el sistema no esté alimentado. En ésta se almacena el código, pero también una tabla de los valores con los que cargar las variables globales inicializadas que el programador usa en la aplicación. La inicialización del sistema operativo usará esta tabla para inicializar los valores de las aquellas variables globales de la aplicación que lo necesiten, véase apéndice K.

7.2.2 NVRAM: Los datos no volátiles

Como se describió en 6.5.2, la NVRAM se utiliza para almacenar la información no volatil de la aplicación. Se dispone de una NVRAM de 8 bits x 32K, por lo que cada palabra de 16 bits se debe almacenar en dos posiciones de memoria consecutivas. Teniendo en cuenta que para acceder a una palabra hay que realizar dos accesos y componer la palabra a partir de ellos, y que el acceso a esta memoria se caracteriza por tener un estado de espera, es recomendable minimizar el número de accesos a la NVRAM.

Las estructuras de datos que la NVRAM contiene no forman parte del sistema operativo, sino que más bien son estructuras definidas para la aplicación. En esta memoria se almacena:

- *Coeficientes de calibración*: Hasta 20 estructuras de coeficientes de calibración por línea de la calibradora para que la conversión de puntos a gramos pueda variar con la línea y la velocidad de la máquina.
- *Datos de las características de cada uno de los conversores del subsistema*: offset y otros parámetros que se utilizarán para que un conversor determinado funcione siempre de la misma forma entre diferentes puestas en marcha de la calibradora.
- *Datos fundamentales de la calibradora*: número de tazas de cada línea, número de líneas de la calibradora, etc.
- *Peso de las tazas.*

La mayor estructura de datos es la del peso de las tazas. Ésta contiene los coeficientes para hasta 10 líneas y 1400 tazas por línea (caso de máxima carga del subsistema). El peso de la taza es importante para restar este valor del valor obtenido de una pesada (pieza+taza), y obtener así el peso de la pieza. Además del peso, cada taza posee varios flags como el error en la obtención de dicho peso durante el procedimiento de tarado (indica el grado de confianza del peso de la taza), y el número

de veces que ha sido repetida la prueba en tarado para obtener dicho peso. Ambos valores sirven para validar en qué medida el peso es correcto, buscar tazas rotas, etc.

Todos los datos en NVRAM poseen (excepto los pesos de las tazas) un flag testigo que permite comprobar si los valores contenidos en cada una de estas estructuras han sido inicializados.

Las estructuras de la NVRAM no forman parte del sistema operativo, sino que han sido definidas para ser usadas por la aplicación. Dada la alta penalización en tiempo de acceso que resulta del acceso a un dato en NVRAM, durante la inicialización del SO se realizan *copias* en memoria principal de algunos de éstos para obtener un acceso más rápido. Esto implica una redundancia de datos en memoria principal y NVRAM, por lo que cuando un proceso quiera modificarlos, el SO proporciona los servicios adecuados para mantener la coherencia entre el original y la copia.

Durante la inicialización del sistema, si alguno de los datos de NVRAM no es validado por su flag testigo, se lanza un error al control informando de cuál es el parámetro no inicializado, indicando que ello puede causar problemas en la ejecución de la aplicación.

7.2.3 SRAM externa y memoria interna de datos

Existen un conjunto de variables y estructuras que, por ser muy usadas, estan en memoria interna. Esto se hace así no gratuitamente, sino porque dado que el programa reside casi en su totalidad en memoria externa, el DSP no produce retrasos en la ejecución de las instrucciones si tiene los datos en memoria interna. Por ello, las estructuras y datos intensivamente usados se sitúan en memoria interna.

Entre los datos que residen en memoria interna de datos, podemos establecer una clasificación entre aquéllos que pertenecen al sistema operativo y los pertenecientes a la aplicación. Las estructuras que residen en memoria interna y las residentes memoria externa se diferencian básicamente en la cantidad de accesos a las que han de ser sometidas.

A continuación se describen una serie de estructuras clave para el funcionamiento de la aplicación.

Buffers circulares de muestras y resultados intermedios

Los buffers de muestras son estructuras públicas de la aplicación que se utilizan para almacenar las muestras entrantes en cada uno de los canales o líneas de adquisición. Para cada una de las líneas existe un buffer circular y un puntero de control. El planteamiento consiste en la introducción paulatina de las muestras entrantes, en

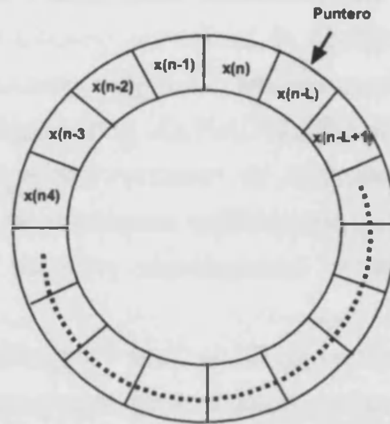


Figura 7.4: Buffers circulares para el manejo de muestras o resultados de filtrado.

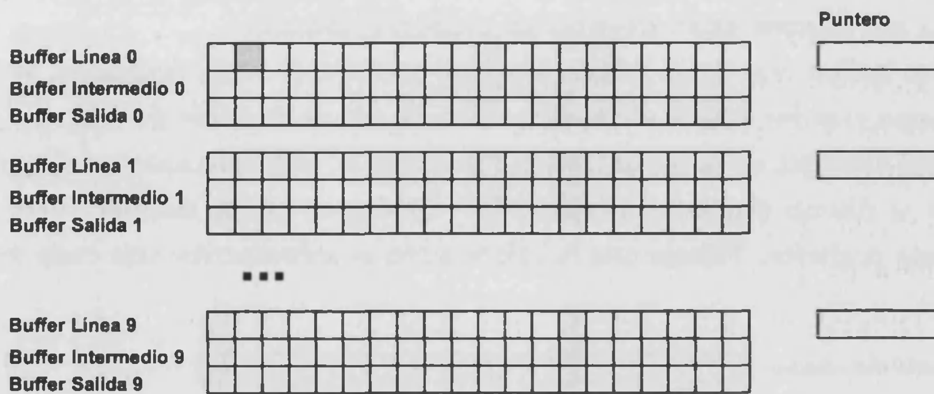


Figura 7.5: Representación lineal de los buffers circulares de muestras y resultados intermedios.

el buffer de cada línea. Una variable 'puntero' indica la posición del buffer en la que se almacenará la próxima muestra.

En un buffer circular, el dato más reciente, $x(n)$, es la posición previa a la dirección donde apunta el puntero, y las muestras anteriores ($x(n-1)$, $x(n-2)$, etc.) se encuentran siempre recorriendo el buffer en sentido circular contrario al de las agujas del reloj. El número de muestras del buffer circular es directamente escalable cambiando la variable privada LONGCANAL y recompilando el sistema operativo. Entonces, la ocupación de memoria se reestructura según las nuevas necesidades. Aunque esto implica que los nuevos buffers empiezan en direcciones diferentes y no tienen la misma longitud, esto es transparente para los procesos que hacen uso de estas estructuras.

Debe tenerse en cuenta que se han de realizar comprobaciones necesarias, vía software, para reposicionar el puntero cuando éste sobrepasa el direccionamiento y establecer la circularidad, ya que el 'C26 no posee mecanismos hardware para realizarlo (otros DSP como el 'C50 contemplan el direccionamiento de buffers circulares sin overhead en el cálculo de la nueva posición cuando se sobrepasa el direccionamiento).

Sin embargo, no sólo hay un buffer para cada línea, hasta ahora se han descrito los buffers de muestras de la adquisición, pero además de éstos existen dos buffers más para cada canal, el buffer intermedio, que se utiliza para sacar la salida de un primer filtrado, y el buffer de salida, que se utiliza para almacenar la salida del filtrado de las muestras del buffer intermedio. Los cálculos de filtrado se deben realizar durante el servicio de la interrupción de la adquisición, y dado que estos cálculos son muy rápidos en los procesadores DSP, no supone una gran latencia (considerando además que dichas estructuras están situadas en memoria interna).

Los tres buffers son de la misma longitud y como, si están activados, se rellenan con la misma rapidez, todos ellos comparten la misma variable de control 'puntero' respecto del principio de la estructura en memoria. El elemento anterior al que apunta la cola es el último elemento introducido. Cualquier buffer circular corre siempre hacia offsets positivos. Nótese que la información se sobreescribe tras cada vuelta del puntero.

Figura 7.6: Representación lineal de los buffers circulares de sincronismos y tiempos.

También están contemplados un buffer, también circular, de sincronismos (planteado para introducir el sincronismo en el que llega cada muestra entrante y poder identificar cambios de sincronismo, etc.) y el de la cola de tiempos (pensado para introducir

el valor del *timer* en el que llega la muestra para poder analizar posibles pérdidas, etc.). En realidad, estos buffers no se usan en la aplicación pero existen genéricamente por si el programador desea hacer uso de ellos.

Estructuras de recepción de mensajes

El buffer de recepción es una estructura clave que se usa para la recepción de mensajes. Básicamente, para formar los mensajes de entrada se usa la función 'CAN-Receive'.

Como describimos en sección 6.8, la placa de comunicaciones interrumpe al DSP cada vez que tiene un dato (byte) para él. El servicio de recepción va tomando los bytes enviados por la placa de comunicaciones para componer el mensaje, bien sea mensaje CAN o propio de la placa de comunicaciones.

La función 'CANReceive' hace uso de la estructura para la recepción de mensajes esquematizada en la figura 7.7, llamada "molde de recepción" por ser la encargada de controlar la formación del mensaje enviado desde la placa de comunicaciones. Cuando se forma un mensaje ya se puede apilar a la cola de mensajes de recepción o a la cola de prioridad que corresponda, dependiendo del modo activo de recepción de mensajes, sección 7.3.3.

El "molde de recepción" está formado por un buffer de 20 bytes, una variable que controla el número de bytes de mensaje en la estructura y otra que controla el número de bytes que restan por recibir del mensaje actual. La cola de recepción está formada por un buffer y dos variables: cabeza y cola.

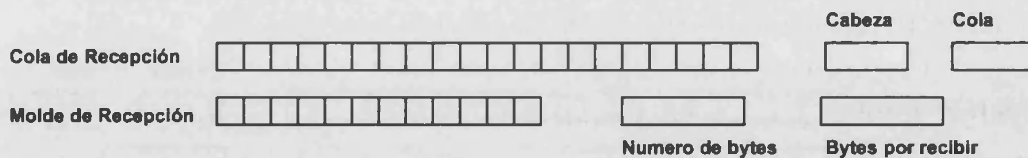


Figura 7.7: Esquemático de la cola de recepción, molde de recepción y variables asociadas.

Este mecanismo está continuamente formando y bombeando los mensajes recibidos ya completos. El envío del mensaje al buffer de recepción se realiza con el servicio 'putmens'. Además se pone a 1 el bit de 'fintprior' correspondiente a la cola de mensajes destino, para indicar que hay al menos un mensaje en dicha cola.

La cola de recepción, el "molde de recepción" y sus variables relacionadas, son estructuras privadas del sistema operativo, por lo que éste realiza completamente su gestión. La variable de flags 'fintprior' es una variable pública del sistema operativo. Los procesos de la aplicación deben utilizar ésta como variable de sólo lectura para evitar corromper el funcionamiento del sistema.

Estructuras de transmisión de mensajes

La cola de transmisión de mensajes es una estructura clave usada para la transmisión de los mensajes. Para gestionar la salida de mensajes se usa, básicamente, la función 'CANTransmit'.

Como sabemos, la CPLD contiene dos registros, uno de transmisión a la placa del CAN (placa de comunicaciones) y otro de recepción, sección 5.8. El que ahora nos interesa es el de transmisión. Sabemos que cuando el flag /FCINT se pone a nivel alto, significa que la placa de comunicaciones ha leído del registro de transmisión el último byte que se escribió, y que está dispuesta para que otro byte sea enviado. Esta señal es examinada por el servicio de interrupción del *timer*, que periódicamente inspecciona dicha línea para ver si puede transmitirse. Si la placa de comunicaciones no está pendiente de lectura y tiene un mensaje por acabar de transmitir, se transmite el siguiente byte. Si el último mensaje ha sido enviado completamente y existe alguno en la cola de transmisión, se desapila de aquélla y se coloca en el "molde de transmisión", inicializando dicha estructura y enviando el primer byte.

La función 'CANTransmit' hace uso de la estructura esquematizada en la figura 7.8, para la transmisión del mensaje. Ésta es llamada "molde de transmisión" por ser la encargada de contener el mensaje a transmitir e ir enviando paso a paso cada uno de sus bytes. El "molde de transmisión" está formado por un buffer de 20 bytes, una variable que controla el número de bytes del mensaje contenido en la estructura, y otra que controla el número de bytes que restan por transmitir. La cola de transmisión de mensajes está formada por un buffer y dos variables asociadas: cabeza y cola.

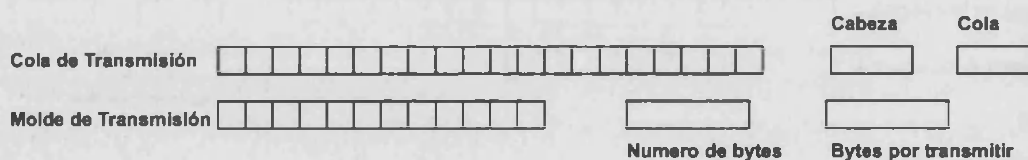


Figura 7.8: Esquemático de la cola de transmisión, molde de transmisión y variables asociadas.

Colas de prioridad

La interrupción /INT0, es la interrupción del conversor y su servicio usa los buffers de muestras, intermedios y de salida. La interrupción /INT2 atiende las peticiones del depurador. La interrupción /INT1 es invocada cuando se ha de realizar una recepción, su servicio usa el "molde de recepción" y la cola de recepción de mensajes.

La interrupción del temporizador, TINT, se usa para realizar *polling* del estado de la placa de comunicaciones para atender transmisiones pendientes; sin embargo esta interrupción también se aprovecha para realizar, si procede, la planificación de procesos.

Dada la jerarquía de interrupciones citada en el párrafo anterior, se puede hablar de una prioridad establecida por hardware entre los procesos de adquisición, recepción y transmisión y planificación. Sin embargo, además de estos, EMMOS establece cuatro niveles de prioridad por software, con los que realizar multitarea entre los procesos de servicio de los mensajes en las distintas colas de prioridad. Hay 3 colas de prioridad y una de trabajos en *background* que es la que menor prioridad tiene. El orden de prioridad de mayor a menor se ilustra en la figura 7.9.

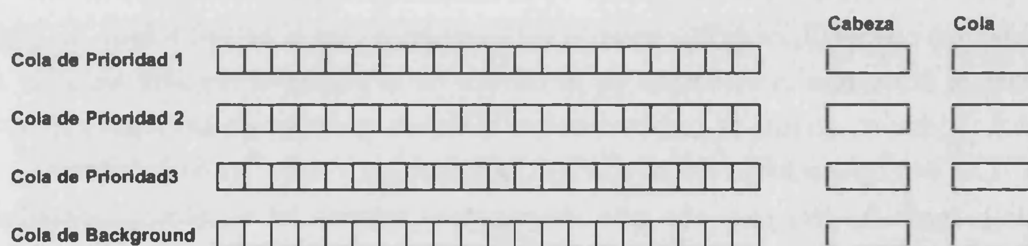


Figura 7.9: Esquemático de las distintas colas de mensajes priorizadas.

La idea es que los mensajes entrantes tengan asignada una prioridad y en función de ésta sean *bombeados* a una cola de prioridad u otra. Así, cuando no hay interrupciones hardware pendientes (las cuales de por sí están priorizadas) se prioriza por software la ejecución de los servicios de los mensajes existentes en las colas de prioridad. Cada mensaje enviado a una cola de prioridad es una orden que se tiene que servir y significa un proceso a realizar. Si el planificador detecta que existe un mensaje en una cola de prioridad superior a la que posee la CPU, debe almacenar el contexto y asignar la CPU al servicio del mensaje de mayor prioridad. Por ejemplo, supongamos que un servicio de un mensaje de la cola de nivel 3 posee la CPU, y la recepción bombea un mensaje a la cola de nivel 2. Cuando el planificador detecte la presencia de éste, asignará la CPU al servicio del mensaje de la cola de nivel 2, que (si no existe otro evento) progresará en su ejecución hasta finalizar restaurando el contexto y devolviendo la CPU al servicio de la cola de nivel 3. Nótese como el módulo planificador, sección 7.5, es el que prioriza la ejecución de los procesos de servicio bombeados.

Array de punteros

Es una estructura localizada en memoria interna, utilizada para realizar una gestión más eficaz en los servicios 'getmens' y 'putmens', del apilamiento y desapilamiento de los mensaje en las distintas colas. Este proceso se realiza a partir de un puntero obtenido de un array de punteros, dado que conociendo la disposición de las estructuras en memoria y la prioridad de la cola de mensajes a la que acceder, puede accederse rápidamente al buffer y las variables asociadas a la cola (cabeza, cola, longitud, etc.), optimizando así el espacio y la rapidez. Esta estructura se usa para desapilar/apilar mensajes especificando únicamente la cola de mensajes origen/destino.

La pila de memoria

La pila es esencialmente una parte de la unidad de memoria interna, accedida por una dirección que siempre se incrementa o decrementa tras el acceso a ésta. El registro que siempre almacena la dirección de la cabeza de la pila es el registro auxiliar AR1, apéndice G, que en forma de puntero indica siempre la dirección del ítem superior de la pila. Las dos operaciones de la pila son la inserción y desecho de los ítems.

Existe también una pequeña pila de registros interna en el microprocesador, de una profundidad de ocho bytes. No obstante, debido a la cantidad de información que debe ser apilada (bastantes más de 8 palabras), se utiliza el apilamiento en memoria interna. Una pila puede crecer y ocupar todo el espacio de memoria si hace falta. Sin embargo, generalmente se le suele reservar un tamaño, que en nuestro caso es de 350 palabras y que para nuestra aplicación resulta más que suficiente. La saturación de la pila produce *overflow* y pasa a invadir y corromper otras estructuras contiguas. Esto no puede protegerse ya que el DSP no dispone de mecanismos de paginación, protección, etc. La única protección consiste en dimensionar la pila adecuadamente para que no pueda ocurrir un *overflow*.

Otras variables del sistema operativo

Además de las estructuras de datos mencionadas, el sistema operativo utiliza muchas otras variables privadas. La lista siguiente enumera algunas de las variables globales más importantes utilizadas por EMMOS:

- Contador de mensajes perdidos.
- Modos de actuación ante mensajes perdidos.
- Flags de transmisión y recepción.

- Flags de presencia de mensajes en las distintas colas ('fintprior').
- Variables temporales.
- Estado y disponibilidad de las líneas hardware de pesado.
- Identificador que el control asigna al subsistema de pesado.
- Estado de los mecanismos de depuración a nivel de mensaje.
- ...

Variables propias de la aplicación

Entre las variables predefinidas en memoria RAM, se encuentran las copias de la información almacenada en la NVRAM (a fin de acceder más rápidamente a ella). Las variables globales propias de la aplicación son definidas por el programador.

Además de variables globales, cada proceso utiliza variables locales que residen temporalmente en la pila.

7.3 Gestión de E/S

Generalmente, los sistemas operativos convencionales permiten la entrada de datos mediante teclado o ficheros, y la salida a fichero, terminal o impresión. Sin embargo, el caso que nos ocupa es un tanto particular, puesto que se trata de un sistema embebido en el que no existen archivos ni entradas y salidas típicas. El subsistema de pesado puede comunicarse con el control a través de mensajes CAN, siendo ésta su única posibilidad de E/S.

El CAN, es un protocolo de comunicación en red mediante paquetes, que puede transmitir a una velocidad de 1Mbps. La placa de comunicaciones actúa como enlace entre CAN y subsistema, actuando como intermediaria en el envío y recepción de mensajes entre el sistema de pesado y el control. La función de la placa de comunicaciones es inicializar la comunicación CAN e identificar los paquetes destinados al subsistema de pesado, que almacenará en un buffer e irá transmitiendo al DSP. En el otro sentido, también almacena los mensajes que recibe del 'C26 para gestionar su envío vía CAN.

7.3.1 Estrcutura de los mensajes recibidos

La placa de comunicaciones transmite al DSP tanto mensajes que recibe del bus CAN como mensajes internos, originados por la propia placa de comunicaciones con propósitos de información, comunicación de estado y sincronización.

Mensajes recibidos del bus CAN

El tamaño de los mensajes recibidos del bus CAN posee es variable de 4 a 12 bytes, dependiendo del número de datos que el mensaje incorpore. La estructura del mensaje se se ilustra en la tabla 7.1.

	CONFIGURACIÓN	
Byte 1	0000 DDDD	DDDD: Longitud del campo de datos del mensaje (de 0 a 8)
	INFORMACIÓN DEL IDENTIFICADOR	
Byte 2	XXXX XXXX	XXXX XXXX: Orden (MSB)
Byte 3	XXXL LLLL	XXX: Orden (LSB) LLLLL: Índice de línea (de 1 a 10)
Byte 4	0000 00000	Byte de relleno
	DATOS	
Byte 5	dddd dddd ...	dddddddd: Byte de datos (Orden LSB - MSB) (de 0 a 8 según se ha especificado en el primer byte)

Tabla 7.1: Estructura de los mensajes CAN recibidos por el DSP.

Nótese que el mensaje posee una cabecera que indica :

- *Número de bytes de datos que se envían.* Necesario para conocer el tamaño exacto del mensaje entrante.
- *Identificador del mensaje.* Se corresponde con la orden que debe ser servida.
- *Línea del calibrador sobre la que recaerá la acción.*

Tras la cabecera se envían los bytes de datos, que recogerá el proceso para realizar el servicio de la orden.

Mensajes internos recibidos de la placa de comunicaciones

Los mensajes internos enviados por la placa de comunicaciones poseen el formato que ilustra la tabla 7.2.

El propósito de este tipo de mensaje es establecer una comunicación interna entre la placa de comunicaciones y el DSP. Es decir, son mensajes que no proceden del bus CAN. Esto permite el paso de cierta información, test de comunicaciones, comunicación de estado, sincronización, etc. Nótese que en el byte 1 de la tabla 7.2, XXXX será distinto de 0000 para distinguir éste de un mensaje CAN.

	CONFIGURACIÓN	
Byte 1	XXXX DDDD	XXXX: Orden interna. DDDD: Longitud de este mensaje, incluido este byte (de 1 a 3)
	DATOS	
Byte 2	dddd dddd	0, 1 ó 2 bytes de datos según se especifica en DDDD
Byte 3	dddd dddd	

Tabla 7.2: Estructura de los mensajes internos recibidos por el DSP.

7.3.2 Estructura de los mensajes transmitidos

El DSP transmite a la placa de comunicaciones tanto mensajes destinados al control u otros subsistemas, que deben ser enviados vía CAN, como mensajes internos destinados a la propia placa de comunicaciones con objeto de inicialización, borrado de buffers, etc.

Mensajes transmitidos al bus CAN

Cuando la aplicación decide enviar un mensaje al control o a cualquier otro subsistema de la calibradora (TTTT en la estructura del mensaje), debe formar un mensaje con el formato de cabecera y datos ilustrado en la tabla 7.3.

	CONFIGURACIÓN	
Byte 1	0000 DDDD	DDDD: Longitud del campo de datos del mensaje (de 0 a 8)
	INFORMACIÓN DEL IDENTIFICADOR	
Byte 2	XXXX XXXX	XXXX XXXX: Orden (MSB)
Byte 3	XXX0 TTTT	XX: Orden (LSB) TTTT: Tipo de tarjeta destino.
Byte 4	0nnn nnnn	nnnnnnn: Índice de la tarjeta destino.
Byte 5	000L LLLL	LLLLL: Número de línea o Número de tarjeta.
	DATOS	
Byte 6	dddd dddd ...	dddddddd: Byte de datos (Orden LSB - MSB) (de 0 a 8 según se ha especificado en el primer byte)

Tabla 7.3: Estructura de los mensajes CAN transmitidos por el DSP.

- Cantidad de datos (bytes) que se envían.
- La orden o identificador del mensaje indicará a la tarjeta destino una acción a realizar y/o el tipo de datos que se envían.

- El tipo de tarjeta destino y el índice (TTTT y nnnnnnn) determinan unívocamente la tarjeta de la calibradora a la que se envía el mensaje. Usualmente para envíos al control, se usa TTTT=0000 y nnnnnnn=0000000.
- Número de línea o tarjeta.

La cabecera contendrá información sobre:

Mensajes internos transmitidos a la placa de comunicaciones

Los mensajes internos enviados por el DSP adoptan un formato similar al de los enviados por la placa de comunicaciones.

		CONFIGURACIÓN
Byte 1	XXXX DDDD	XXXX: Orden interna. DDDD: Longitud de este mensaje, incluido este byte (de 1 a 3)
		DATOS
Byte 2	dddd dddd	0, 1 ó 2 bytes de datos según se especifica en DDDD
Byte 3	dddd dddd	

Tabla 7.4: Estructura de los mensajes internos transmitidos por el DSP.

El propósito de este tipo de mensaje es establecer una comunicación interna entre el DSP y la placa de comunicaciones. Es decir, son mensajes que no transcienden al bus CAN. De este modo, el DSP podrá programar, en la medida de lo posible, la placa de comunicaciones, iniciar las comunicaciones, etc. Nótese que en el byte 1 de la tabla 7.4, XXXX será distinto de 0000 para que la placa de comunicaciones distinga éste de un mensaje CAN.

7.3.3 Gestión de la recepción de mensajes

La placa de comunicaciones interrumpe al subsistema de pesado cada vez que éste tiene disponible un dato (byte) para él. El servicio de la interrupción /INT1 es el encargado de tomar los bytes enviados para componer el mensaje, bien sea mensaje CAN o interno de la placa de comunicaciones. Una vez completado el mensaje lo despacha bien a la cola de recepción mensajes, bien a una cola de mensajes de prioridad, dependiendo del modo de despacho de mensajes establecido.

El servicio de la interrupción de recepción, /INT1, se realiza completamente con las interrupciones deshabilitadas. Éste usa básicamente el servicio 'CANReceive' para gestionar los mensajes de entrada. Este proceso queda latente, surgiendo tras cada interrupción de recepción para formar paulatinamente el mensaje entrante haciendo

uso de la estructura para la recepción de mensajes esquematizada en la figura 7.7, llamada "molde de recepción", encargada de almacenar los mensajes enviados por la placa de comunicaciones durante su formación. Esta estructura clave, está formada por un buffer de 20 bytes, una variable de control del número de bytes de mensaje en el buffer y otra que controla el número de bytes del mensaje entrante que restan por recibir.

El molde de recepción es una estructura privada del sistema operativo, gestionada por el servicio 'CANReceive', a la que no pueden acceder los procesos de la aplicación.

El servicio 'CANReceive', realiza los siguientes pasos cuando es invocado para la recepción del byte entrante:

- Lee el byte entrante.
- Realiza el punto que corresponda:
  - Si el número de palabras restantes para la formación del mensaje es cero, implica que el que se recibe es el primer byte de un mensaje. En este caso inspecciona los cuatro bits MSB para averiguar si es un mensaje CAN o interno de la placa de comunicaciones. Identificando cual es la fuente del mensaje puede calcularse la longitud del mensaje a partir de los cuatro bits LSB, valor en el que se basará para inicializar las variables que controlan el número de bytes del mensaje, el número de bytes que quedan por recibir ('wordsaunr') y el puntero al siguiente espacio vacío del buffer ('nextwordr').
  - Si, por el contrario, el número de palabras restantes para la formación del mensaje no es cero, el byte entrante es uno más de los que forman el mensaje que actualmente ocupa al buffer de recepción.
- Se coloca el dato en el "molde".
- Se incrementa el puntero 'nextwordr' y se decrementa el número de bytes que quedan por recibir.
- Si el byte entrante era el último byte del mensaje, se despacha éste apilándolo en la cola de mensajes que proceda, dependiendo del estado del modo de recepción de mensajes, tras desentrelazar algunos parámetros del mensaje entrante si se trata de un mensaje CAN.

Este mecanismo está continuamente formando y bombeando mensajes a las colas de mensajes. El apilamiento del mensaje se realiza con el servicio 'putmens'. Además, se activa a 1 el bit de 'fintprior' correspondiente a la cola de mensajes utilizada para indicar que existe en ésta al menos un mensaje.

Modo de despacho de los mensajes tras la recepción

El modo de recepción del sistema describe qué se ha de hacer cuando el "molde de recepción" completa la formación de un mensaje. Por una parte, se puede apilar el mensaje en la cola de mensajes de recepción para que la aplicación pueda programar un intérprete de órdenes. Por otra, es posible utilizar el modo de despacho automático, que es el modo utilizado por defecto.

El modo de despacho de mensajes recibidos es controlado por la variable pública del sistema operativo 'modedispatchmessage', que se pone a TRUE por defecto señalando despacho directo a las colas de prioridades. Puesta a FALSE señala despacho del mensaje entrante a la cola de recepción de mensajes.

A la hora de despachar un mensaje formado, el servicio 'CANReceive' considera el modo de recepción. Si está activo el despacho automático de mensajes, se utiliza el servicio 'GetCaracteristicas' para obtener la prioridad del mensaje y comprobar la existencia del proceso de servicio de esta orden. Nótese que aunque exista el proceso que lo sirve, si el sistema no trabaja en el modo de funcionamiento para el que este servicio está definido, el mensaje se deshecha.

7.3.4 Gestión de la transmisión de mensajes

El servicio de transmisión tiene como misión fundamental ir transmitiendo a la placa de comunicaciones los bytes que forman los mensajes existentes en la cola de transmisión, byte a byte, mensaje tras mensaje.

Como sabemos, la CPLD contiene dos registros, uno de ellos es el de transmisión. Para enviar un byte a la placa de comunicaciones, el DSP utiliza el registro de transmisión como almacenamiento intermedio. Cuando el 'C26 escribe un byte en este registro, se activa una señal de interrupción de la placa de comunicaciones y se habilita un flag del registro de flags de la CPLD indicando que el dato está a espera de ser leido por ésta. Si otro byte fuera escrito antes de que la placa de comunicaciones leyera el registro, el anterior byte se perdería. Por esta razón, el DSP ha de esperar siempre a que la placa de comunicaciones lea el byte anterior antes de transmitir otro.

La estrategia para la transmisión de los mensajes es la realización periódica de *polling* del citado flag testigo para saber cuándo puede transmitirse el próximo byte. Así, se utiliza el temporizador para realizar invocaciones periódicas de su servicio, que examina la posibilidad de una nueva transmisión, la cual será realizada si no existe byte a espera de ser leído por la placa de comunicaciones y existe información que enviar.

El tiempo entre invocaciones del servicio del timer es de PRD*100ns. La aplicación puede regular la frecuencia de invocación del planificador mediante el servicio del sis-

tema operativo 'WriteTimer', con el que se puede dar otro valor de carga del *timer* (modifica el registro PRD). Debe tenerse cuidado ya que el cambio de este parámetro puede afectar significativamente las prestaciones del sistema y la aplicación. El valor por defecto es $0x200$. Si éste fuera muy bajo siempre estaría interrumpiendo y colapsaría el sistema, si fuera muy alto podría afectar a la multitarea y las características de tiempo real de la aplicación. La regulación de la invocación del servicio de transmisión se realiza mediante la variable pública del sistema operativo 'fmult'.

El planificador también es invocado periódicamente por el servicio del temporizador. Por ello, se ha establecido un mecanismo por el cual se puede realizar una invocación del servicio de transmisión cada 'fmult' invocaciones del timer. Esto es, se tiene un contador 'contt' que se decrementa cada invocación del timer, invocandose la rutina de transmisión cada vez que 'contt'=0 e inicializando de nuevo éste con el valor de 'fmult'. Así, se consigue que cada invocación del timer realice una planificación y cada 'fmult' veces una planificación + transmisión, véase figura 7.22.

El "molde de transmisión", figura 7.8, es la estructura clave que se usa para la transmisión de mensajes (tanto CAN como mensajes internos). El servicio 'CANTransmit' hace uso de esta estructura para la transmisión de mensajes. Ésta es la encargada de contener el mensaje e ir enviando paso a paso cada uno de sus bytes. El "molde de transmisión" está formado por un buffer de 20 bytes, una variable que controla el número de bytes del mensaje, otra que controla el número de bytes que restan por transmitir ('wordsaunt'), y una última que apunta al siguiente byte que se ha de transmitir ('nextwordt').

El molde de transmisión es una estructura privada del sistema operativo, a la que no pueden acceder los procesos.

El servicio 'CANTransmit' realiza los siguientes pasos cuando es invocado para la transmisión del siguiente byte:

- Si el número de bytes que restan por transmitir ('wordsaunt') es cero, significa que ya se transmitió todo el mensaje. En este caso se toma el siguiente mensaje, que se desapilará de la cola de transmisión para pasar a ocupar el molde de transmisión. Tras un proceso de entrelazado de la orden y el tipo de tarjeta, caso de ser un mensaje CAN, se inicializan las variables que controlan la transmisión del mensaje.

- Se envía el dato al que apunta ('nextwordt').

- Se decrementa el número de bytes a transmitir.

- Se incrementa el puntero al siguiente byte ('nextwordt').

El servicio de transmisión se realiza en su totalidad con las interrupciones deshabilitadas.

7.3.5 Gestión de mensajes

Todas las colas de mensajes son estructuras privadas del sistema operativo. Sin embargo, éste permite apilar y desapilar mensajes en las colas de prioridades, background, recepción y transmisión a través de los servicios 'getmens' y 'putmens'. Éstos son unos de los servicios del sistema más usados y poseen el siguiente prototipo de función:

```
int getmens(int prioridad,int *pnewpos)
int putmens(int prioridad,int *parray)
```

Ambos retornan un valor booleano para indicar si la acción se ha realizado con éxito, siendo una excelente herramienta para conocer si se está intentando desapilar un mensaje de una cola vacía ('getmens' retorna FALSE) o apilar en una cola llena ('putmens' retorna FALSE).

En el servicio 'getmens', 'pnewpos' es un puntero al lugar reservado para almacenar el mensaje a desapilar. Este servicio no sólo obtiene el mensaje de la cola en cuestión y lo coloca a partir de la dirección dada, además inspecciona si éste era el último mensaje de la cola, caso en el que pone a cero el bit correspondiente a ésta en 'fintprior' (variable pública utilizada para ver si existen mensajes en cualquiera de las colas de mensajes). De la misma forma, el parámetro 'parray' de 'putmens' es un puntero que indica a este servicio la dirección en la que está el mensaje que se ha de apilar en la cola indicada. Para pasar un mensaje de una cola a otra, obviamente se ha de pasar por un buffer intermedio.

La prioridades indican la cola de mensajes a la que asociar la acción:

1. Background
2. Nivel 3
3. Nivel 2
4. Nivel 1
5. Transmisión de mensajes
6. Recepción de mensajes

Nótese como es el mismo servicio el que reconoce la longitud del mensaje a partir de la información de su cabecera.

7.4 Procesos

Cuando los recursos de un sistema se multiplexan entre varios programas activos se pueden conseguir significativos aumentos de rendimiento, por lo que es práctica común organizar actividades relativamente independientes en procesos separados (tareas) y hacer que el sistema operativo las ejecute simultáneamente.

En esencia, el *proceso* o *tarea* es un caso de programa en ejecución. Es el trabajo mínimo que es susceptible de ser planificado individualmente por un sistema operativo. Cada sistema operativo con multiprogramación sigue de cerca todos los procesos activos y les asigna los recursos del sistema según políticas (o normas de actuación) concebidas para cumplir los objetivos de rendimiento teórico.

Sin embargo, cabe destacar que el término *multiproceso* se utiliza en todo momento para describir un entorno con un solo procesador y múltiples procesos que se ejecutan de manera simultánea, y no debe confundirse con el término, también *multiproceso*, que se usa para describir un sistema con múltiples procesadores hardware.

Otro concepto importante es la diferencia entre *multiprogramación* y *multiproceso*. El término multiproceso se usa para indicar cuando un sistema operativo soporta la ejecución simultánea de programas en un solo procesador, sin realizar complicadas formas de gestión de memoria y gestión de archivos. A esta forma de funcionamiento se la conoce también como *multitarea*. La multiprogramación es un concepto más general que indica que un sistema operativo proporciona gestión de memoria y gestión de archivos, además de soportar la ejecución concurrente de programas. Así pués, un sistema operativo multiprogramación es también un sistema operativo multiproceso (o multitarea), mientras que a la inversa no es cierto (Milenkovic, 1988; Deitel, 1990).

7.4.1 El concepto de proceso

Desde el punto de vista del sistema operativo, los procesos compiten entre sí por la asignación de la CPU, memoria y E/S. Típicamente, en un sistema multiproceso (o multitarea) podemos encontrar diferentes procesos en diferentes etapas de ejecución en cualquier momento. Usualmente, un proceso determinado se repite pasando cíclicamente por diversos estados como *ejecución*, *suspensión* y *preparado* (que se describirán posteriormente), antes de finalizar.

Generalmente, cada proceso posee ciertos *atributos* que ayudan al SO a gestionarlo. Los atributos incluyen el estado actual, la prioridad de planificación, los derechos de acceso, y otra información. La división del trabajo en tareas, que se efectuaran como procesos independientes, y la asignación inicial de atributos de proceso pueden ser realizados tanto por el sistema operativo como por el programador, según el tipo de sistema operativo y el entorno de ejecución. En otras palabras, lo que

constituirá un proceso separado en el momento de la ejecución, puede ser *definido por el sistema o por el programador.*

El programador suele ser el encargado de definir los límites de los procesos cuando se desea un elevado rendimiento. Para definir el proceso, atributos, naturaleza de residencia en memoria, prioridad, etc. el programador suele utilizar un lenguaje de programación como C o Ada. De esta forma, el programador controla aspectos importantes del comportamiento y la gestión del proceso en tiempo de ejecución.

7.4.2 El proceso desde el punto de vista del programador de sistemas

Para dar un paso hacia delante en la forma de la gestión de los procesos vamos a utilizar un ejemplo que, a modo de demostración, nos introducirá en este campo.

Procesos secuenciales y multiproceso

Esta sección pretende ilustrar las diferencias de la gestión de los procesos en tiempo de ejecución. Como veremos, éstas dependen de la política de planificación elegida. Para realizar la descripción tomaremos el hilo argumental de un caso práctico:

> *Supóngase que un programador necesita realizar un sistema que ha de adquirir una serie de muestras, procesarlas de tres en tres para obtener resultados y enviarlos en forma de mensaje al control. Este proceso debe realizarse durante todo el período de funcionamiento del sistema.*

Tal y como se enunció el problema, puede pensarse en una resolución secuencial, donde un programa realizaría uno tras otro estos procesos, que denominaremos: **ADQ** representará la adquisición de una muestra, **SENDINF** identifica el envío de un resultado al control, y finalmente **PREPROC** el procesado necesario para extraer el resultado de las tres últimas muestras no procesadas.

La figura 7.2a muestra el comportamiento, en cuanto a tiempos de ejecución, de cada uno de los procesos identificados. En éstos, las casillas de trazo continuo indican las porciones de cada proceso, las flechas indican los momentos en que se producen las interrupciones, y las flechas hacia abajo indican los instantes en que el proceso **PREPROC** queda listo para ser ejecutado. Nótese que la amplitud de los intervalos temporales no son proporcionales a los tiempos reales que usarían estas actividades para ejecutarse, éstos se han exagerado para conseguir una exposición más clara.

Como puede verse en la figura 7.2a, una pasada de **ADQ** consiste en la adquisición de tres muestras. Cada una de estas adquisiciones viene indicada por la interrupción del conversor A/D. Cada servicio de la adquisición tarda 1/4 de ciclo, y la interrupción

que lo invoca se repite cada ciclo. Por otra parte, el procesado **PREPROC** se realiza cada tres nuevas adquisiciones, procesando éstas para obtener un resultado. Cada procesado de las muestras tarda 13/8 de ciclo. Finalmente, el envío del mensaje está gobernado por un evento que indica cuando puede ser enviado el siguiente byte del mensaje. Nótese que un mensaje estará formado por varios bytes que han de ser transmitidos sucesivamente. Sin embargo, tras el envío de un byte se necesita esperar al siguiente evento, que comunica que el anterior byte ha sido leído y que el siguiente puede ser enviado. Así, el envío del mensaje **SENDINF** se simboliza por el envío de 3 bytes gobernados por un evento indicativo del momento adecuado para el envío del siguiente. El envío de cada byte tarda 1/8 de ciclo, mientras que el mínimo tiempo para el envío del mensaje completo es 9/8 de ciclo.

Dado que en este punto sólo interesa el comportamiento de los procesos, el valor de la unidad de tiempo se deja sin especificar. Sin embargo, para aquéllos que necesiten cuantificar cualquier coordenada, la unidad temporal podría tomarse del orden de décimas de milisegundo.

Puede pensarse que una posible resolución del problema consiste en realizar secuencialmente cada uno de los procesos: primero se toman tres muestras, tras ello se procesan, para finalmente enviar el resultado al control en forma de mensaje. La figura 7.2b ilustra esta realización secuencial, de donde se desprende que no sería el procedimiento más adecuado ya que la realización de una pasada ocupa siete ciclos, con lo cual perdemos 3 muestras, las que se hubieran adquirido en los tiempos 4, 5 y 6, por lo que éste dejaría de ser un sistema de estricto tiempo real.

Nótese que la suma de los tiempos de ejecución de los tres procesos necesarios son:

- 1/4 * 3 (tiempo de ejecución de los tres servicios de interrupción de cada pasada)
- 13/8 (tiempo de ejecución del procesado)
- 1/8 + 1/8 + 1/8 (tiempo consumido para enviar los tres bytes del mensaje)

El tiempo total de ocupación de la CPU para cada grupo de muestras es 22/8. Teniendo en cuenta que se disponen de tres unidades de tiempo para tres muestras (igual a 24/8), el sistema puede resolverse en tiempo real y sin pérdida de muestras. Durante la realización secuencial, la eficiencia es muy baja, la CPU está inactiva en un 55,2% del tiempo. Queda pues claro que, siendo la realización secuencial una primera aproximación al problema, ésta no es la resolución óptima, por lo que abordaremos otra que aproveche el paralelismo del proceso y permita la planificación de otros procesos en los tiempos muertos.

Una segunda aproximación al problema es la asignación de prioridades a los procesos. De forma similar a las prioridades jerarquizadas de las rutinas de servicio de

las interrupciones hardware, a cada proceso se le asigna un nivel de prioridad por software. La finalidad de las prioridades por software es indicar al sistema operativo la importancia relativa de un proceso específico. En particular, cuando dos o más procesos solicitan un recurso simultáneamente, ese recurso se asignará a aquel proceso que tenga la prioridad más elevada.

En el diagrama de la figura 7.10a, se asume que las prioridades, por orden decreciente, son las siguientes: **ADQ**, **SENDINF** y **PREPROC**. La asignación de prioridad es un componente importante del ajuste del rendimiento de los sistemas que tienen requisitos críticos en cuanto a tiempo. Esta asignación la suele realizar el programador de forma razonada. Sin embargo las prioridades pueden volver a ser reasignadas posteriormente para realizar ajustes y obtener un mejor sistema de ejecución.

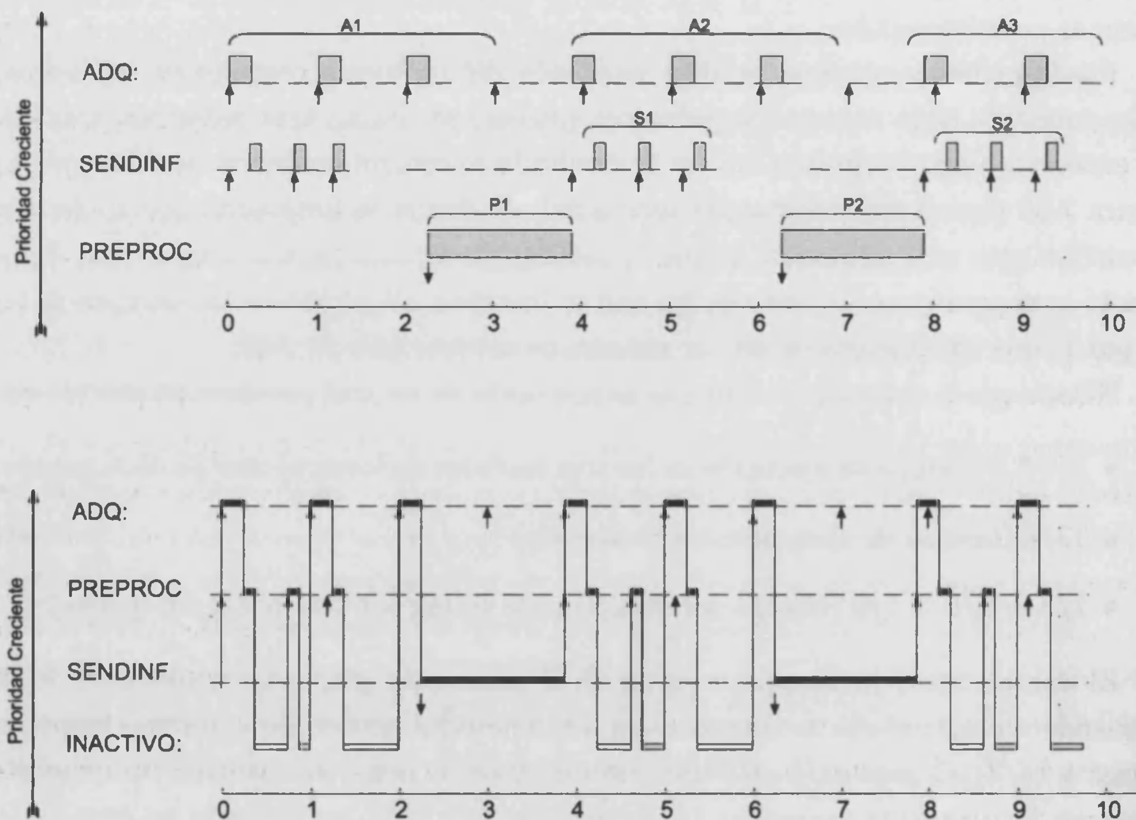


Figura 7.10: Ejecución basada en prioridades.

El proceso **ADQ** dispone de la más alta prioridad dado que se necesita una rápida respuesta a las ráfagas de señales de interrupción del conversor, para la adquisición

de los datos. El segundo nivel de prioridad se concede a **SENDINF** para favorecer el bombeo de los bytes que forman el mensaje al control. Este proceso atiende a eventos que le indican la posibilidad de enviar un byte siempre que queda información por enviar. Finalmente el último nivel de prioridad se asigna a **PREPROC**.

La figura 7.10a muestra la ejecución basada en prioridades de los procesos, considerados desde el punto de vista del sistema. La planificación aplicada en la figura 7.10a es una de las formas más simples, se basa en una planificación no expulsiva con derecho preferente basada en prioridades. Como veremos esta planificación no expulsa al proceso que está ejecutándose, pero cuando finaliza elige para su ejecución aquél de los preparados que mayor prioridad tenga.

Como puede verse en 7.10a, en A1, la primera interrupción del conversor se da en el mismo instante en el que se produce el evento que señala el envío de un byte del mensaje del resultado anterior. Como el servicio de la interrupción del conversor tiene mayor prioridad se ejecutará éste por entero antes de ejecutar el envío del byte. Nótese como esto produce un retardo entre el instante en el que se señala el evento de envío del byte y la ejecución de su envío. Cuando la tercera interrupción de la pasada A1 se ejecuta, puede verse como se produce un evento (flecha hacia abajo) que señala que ya se puede realizar el procesado de estas muestras, al no existir pendiente otro proceso de prioridad superior, se ejecuta **PREPROC**. Es importante notar como la duración del procesado de la información es tan grande que durante su ejecución se indica una interrupción del conversor, aunque el servicio de la interrupción no podrá realizarse hasta que P1 finalice. Al finalizar P1 se produce un evento indicando que el resultado del procesado está preparado y ya se puede enviar el primer byte del mensaje, S1. Pero como existió anteriormente una interrupción del conversor, de mayor prioridad, el envío del primer byte tendrá que esperar a que finalice la ejecución de **ADQ**, en A2. Nótese en este punto, como el retraso en la ejecución del proceso **ADQ**, que sirve a la interrupción, ha sido grande (retraso entre la flecha que indica la interrupción y el momento del comienzo de la ejecución de **ADQ**). Entonces, mientras **ADQ** se está ejecutando, sobreviene una nueva interrupción del conversor que se confundirá con la interrupción en curso [1], por lo que el efecto global es la pérdida de una interrupción. La figura 7.10a ilustra el progreso de la ejecución del sistema, con básicamente los mismos efectos descritos.

[1]En el TMS320C26, las interrupciones externas son activas tanto por flanco como por nivel. Por ello, un pulso en la señal de interrupción dispara un servicio de interrupción. Si se selecciona interrupción por nivel y éste no se deshabilita, la ISR seguirá reentrando indefinidamente. Por esta razón, siempre que una interrupción sea activada por nivel, la ISR debe articular algún mecanismo por el cual restaure el nivel de la interrupción para desactivarla. Típicamente esto se produce al final de la ISR de servicio. Si antes de este momento se vuelve a generar una o más interrupciones, éstas se confundirán con la actual, provocando al sistema una pérdida de interrupciones.

En torno a esta segunda aproximación podemos concluir, véase figura 7.10a, que el proceso se realiza en cuatro unidades de tiempo. Sabiendo que para cada pasada se necesitan 22/8 de tiempo de CPU, puede deducirse que se pierden 10/8 de CPU en los cuales la CPU está inactiva. Sin embargo, la introducción de una forma tremendamente simple de planificación permite realizar el mismo proceso en un 66% del tiempo respecto a la realización secuencial. Este esquema de planificación, pese a su simplicidad, mejora notablemente la eficiencia y aunque el sistema pierde una de cada cuatro muestras, se disminuye la pérdida de muestras respecto a la realización secuencial y aumenta la eficiencia del sistema.

La figura 7.10b indica una forma alternativa de visualizar el funcionamiento de un sistema basado en prioridades, se llama ***diagrama de prioridad versus tiempo de proceso*** y se utiliza para la traza de la ejecución de los procesos en sistemas basados en prioridades, y para la depuración de sistemas operativos.

De esta segunda aproximación cabe destacar el efecto perjudicial que supone para los procesos de mayor prioridad, que existan procesos muy largos de prioridad menor que no puedan ser planificados antes de su finalización. Por lo que se habrá de utilizar un tipo de planificación con mayor rendimiento.

Consideremos, para finalizar, una realización más adecuada del sistema utilizando una planificación expulsiva con derecho preferente basada en prioridades e iniciada por eventos. En ésta, cuando se produce un evento, se invoca al planificador, que decide si la CPU debe seguir asignada al proceso que la poseía o éste debe expulsarse para asignar la CPU a un proceso con mayor prioridad.

Las prioridades de los procesos se mantienen. Las figuras 7.11a y 7.11b ilustran la realización del multiproceso mediante el nuevo esquema de planificación. Como puede observarse, si nos fijamos en la realización de **PREPROC**, veremos que el procesado de las muestras es interrumpido dos o más veces. Por ejemplo, en P1 podemos observar como el evento que es la interrupción del conversor, invoca al planificador, que expulsa este proceso por tener menor prioridad y asigna la CPU al de mayor prioridad preparado. Como puede verse, en este caso las interrupciones del conversor, al tener mayor prioridad, consiguen siempre el favor del planificador, por lo que no se observan retrasos entre evento y comienzo de la adquisición ni pérdida de muestra alguna. Por otra parte, **SENDINF** tiene una prioridad intermedia y también interrumpe la ejecución de **PREPROC** (véase P1, P2 y P3) cuando sobreviene el evento. Sólamente existe un retraso entre el evento y el comienzo de **SENDINF** en S2, causado porque éste sobreviene al mismo tiempo que la interrupción del conversor, que al tener mayor prioridad cede la CPU a **ADQ**, y **SENDINF** obtiene la CPU sólo cuando el proceso de mayor prioridad ha sido terminado.

La planificación expulsiva con derecho preferente basada en prioridades, realiza el

sistema de una forma más eficiente. El proceso se realiza ahora en tiempo real, siendo periódico en tres unidades de tiempo. Por lo que si cada pasada se necesita 22/8 de tiempo la CPU, sólo se pierden 2/8 de CPU en los cuales la CPU está inactiva (en la práctica este tiempo se vería reducido por el uso de la CPU por parte del planificador). Esto supone que el proceso se realiza en un 46% del tiempo respecto a la realización secuencial, y que la CPU está asignada el 92% del tiempo. Como puede verse, la elección del método de planificación aumenta la eficiencia del sistema y la productividad de la CPU.

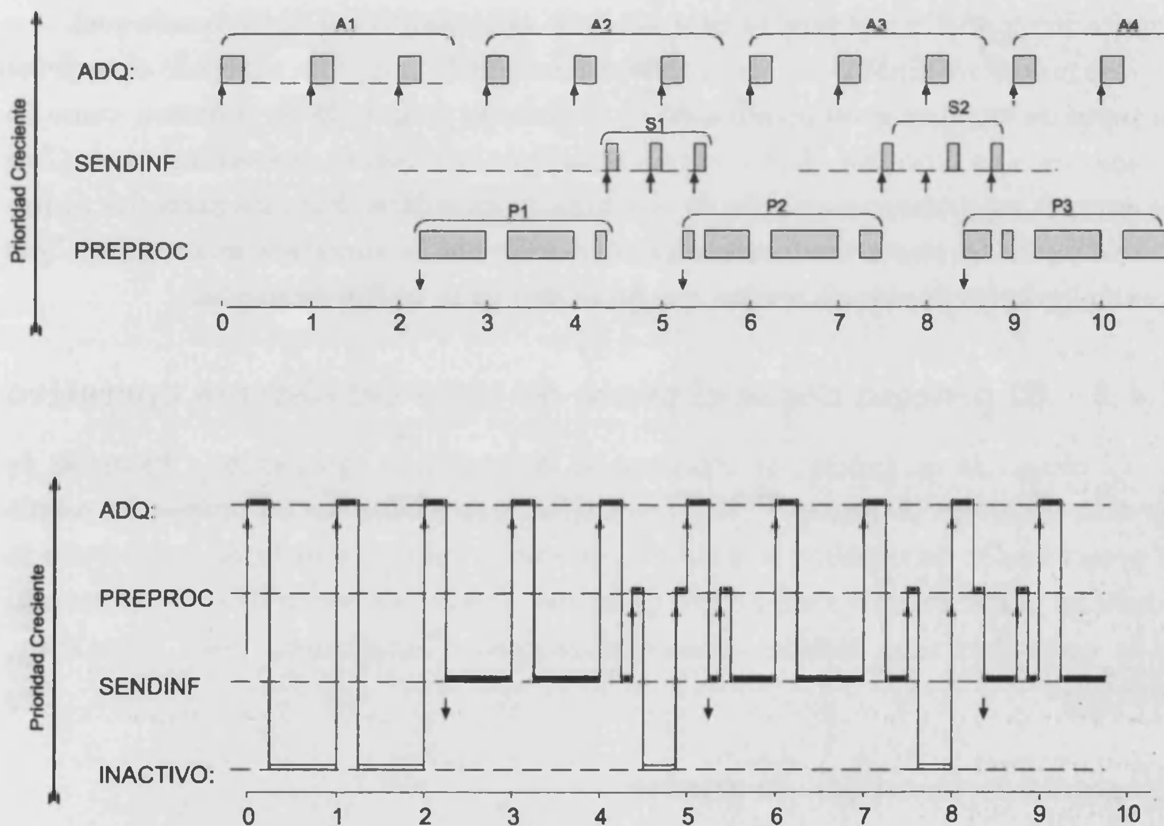


Figura 7.11: Ejecución basada en prioridades con derecho preferente.

La elección del esquema de multiproceso

la operación de multiproceso *puede mejorar de manera espectacular el rendimiento incluso de sistemas pequeños y cerrados.* La potencia de la concurrencia de los procesos está en que se pueden especificar las secuencias de código que se pueden ejecutar simultáneamente, permitiendo que el sistema operativo saque provecho del paralelismo posible en un entorno dado, *lo cual constituye una característica extremadamente*

valiosa en sistemas de tiempo real o de cualquier sistema en el que haya una fuerte necesidad de responder a acontecimientos críticos en cuanto a tiempo.

Otra conclusión que nuestro ejemplo pone de manifiesto es que dentro del multiproceso se ha de seleccionar cuidadosamente el tipo de planificación que el sistema operativo necesita, ésta está en función de las características de las aplicaciones que se han de ejecutar. Esto es muy importante dado que una planificación más compleja no se consigue a cambio de nada. Además de que cuanto mayor uso se haga del planificador tanto más tiempo de CPU y recursos consume, un esquema complejo de multiproceso introduce una complejidad adicional de señales y asignaciones de prioridad, proceso que exige una cuidadosa codificación y aumenta la complejidad de la depuración y análisis de rendimiento antes de su implantación en el mundo real.

Los procesos definidos por los programadores pueden exigir un profundo conocimiento tanto de estrategias de planificación y facilidades de gestión de procesos, como de asignaciones de atributos. Además, en aplicaciones críticas hay que realizar un análisis de tiempos y del comportamiento de la aplicación específica. Por otra parte, los procesos múltiples requieren más depuración de errores que la operación secuencial debido a complejidades de sincronización que no se dan en el código secuencial.

7.4.3 El proceso desde el punto de vista del sistema operativo

El código de un proceso se compone de instrucciones ejecutables y llamadas de servicio al sistema (llamadas al SO). Los atributos asociados con un proceso los asigna el programador de sistemas o el sistema operativo mismo, e incluyen entre otros la prioridad por software y los derechos de acceso. Desde este punto de vista, el proceso es la entidad mínima individualmente planificable (Tanenbaum, 1992; Milenkovic, 1988).

Diagrama de transición de estados

Como sabemos, el planificador es el módulo del sistema operativo que se encarga de asignar la CPU a procesos que estén preparados, basándose en una política o estrategia de planificación y relegando cualquier proceso que pudiera estar haciendo uso de ella sin necesidad de que éste haya finalizado. No obstante, si el proceso expulsado no hubiera finalizado, el sistema operativo se encarga de asignarle posteriormente la CPU para que vaya avanzando en el progreso de su ejecución hasta finalizar.

El proceso va atravesando diferentes estados hasta su finalización. La figura 7.12 proporciona una forma general del diagrama de transición de estados del proceso.

Las cuatro categorías generales de estado del proceso son: *inactivo, preparado, en ejecución y en suspenso.* Así, un proceso creado (dado a conocer al SO) se está

Figura 7.12: Forma general del diagrama de transición de estados.

ejecutando, preparado para su ejecución o dejado en suspenso en espera de que ocurra un acontecimiento o evento.

- El estado de *inactividad* indica procesos que no son conocidos por el SO. En este caso todos los procesos en espera de activación, así como los programas que todavía no han sido presentados al SO, pueden ser considerados como inactivos.

- El proceso *preparado* posee todos los recursos que se necesitan para su ejecución, a excepción de la CPU. Cuando el sistema operativo toma el control de la CPU y lo pasa al planificador, éste puede seleccionar un proceso preparado para su ejecución y asignarle la CPU. Entonces este proceso pasa a estado de ejecución.

- Se dice que un proceso está en estado de *ejecución* cuando posee todos los recursos necesarios para su ejecución, incluida la CPU. En un sistema monoprocesador, de una sola CPU, sólo puede ejecutarse un proceso a la vez, por lo que únicamente puede haber un proceso en ejecución y varios preparados o en cualquier otro estado.

- Se dice que un proceso está en *suspensión* cuando le falta algún recurso aparte de la CPU como, por ejemplo, una señal de sincronización. Normalmente estos procesos son apartados fuera de la competición por conseguir su ejecución (típica

de los procesos preparados), hasta que se elimina la condición que provocó su suspensión.

Sin embargo, no todos los sistemas operativos disponen de este diagrama sino de un subconjunto.

Los descriptores o Bloques de Control de Proceso

Generalmente, el sistema operativo agrupa toda la información que necesita acerca de un proceso determinado en una estructura de datos llamada ***descriptor de proceso*** o ***bloque de control de proceso*** (PCB, "Process control-block"). Alguna de la información del PCB es:

1. *Nombre del proceso (ID).*
2. *Prioridad.*
3. *Estado.*
4. *Otros campos.*

Éste se crea siempre que se crea un proceso, y cuando finaliza su ejecución simplemente se destruye y libera la memoria utilizada.

Para seguir la pista a todos los procesos, un SO mantiene listas de bloques de control de proceso (PCB) clasificadas por el estado actual de los procesos afines. En general, hay una ***lista de procesos preparados,*** que contiene los PCB de todos los procesos preparados, y una ***lista de procesos en suspenso.***

Siempre que hay que realizar la transferencia de la ejecución de un proceso a otro, interviene el sistema operativo para actualizar las listas del sistema. El primer paso es registrar el estado del proceso en ejecución que está a punto de quedar en suspenso o relegado. El estado de un proceso incluye generalmente: contador del programa, indicador de comienzo de pila, palabra de estado del procesador y todos los registros accesibles para los programas. El estado de un contexto, a diferencia del almacenamiento en contexto de interrupciones, tiene que registrar el estado completo, porque el contexto de un servicio de interrupción puede salvar sólo aquellos registros que vayan a ser modificados por la ISR, siempre que se ejecute con las interrupciones deshabilitadas, pero no hay manera de que el SO sepa qué registros particulares podrá utilizar el proceso en ejecución en el futuro.

El cambio de proceso es una operación considerablemente más compleja y con más tareas relativas al coste que el cambio de contexto, y puede llegar a estar bastante involucrado (en sistemas operativos grandes) con planes de contabilidad detallados

y sofisticados esquemas de planificación de recursos. Dada su complejidad y la frecuencia relativamente alta con que se produce, el cambio de proceso puede afectar de manera significativa al rendimiento del sistema operativo. Por este motivo, la velocidad de cambio del proceso tiene que ser alta en sistemas orientados al rendimiento, tales como las aplicaciones en tiempo real. Muchas veces se emplea un esquema de hardware para acelerar el paso de un proceso a otro, consistente en que la CPU disponga de múltiples juegos estructuralmente idénticos de registros. El mínimo son dos bancos de registros, uno para el SO y otro para procesos usuarios. Algunos DSP cuentan, en este sentido, con un par de bancos de registros para realizar cambios rápidos de contexto.

7.4.4 Los procesos EMMOS

Bajo el SO EMMOS, todos los procesos son estáticos, por lo que residen en memoria desde que el cargador secundario los emplaza durante el arranque, permaneciendo siempre residentes para acelerar su ejecución. Cada proceso debe ser, por lo general, lo más escueto posible. Los procesos realizan tareas que comprenden procesado de resultados, atención de peticiones de estado del sistema, modo de funcionamiento, petición de datos, etc.

7.4.4.1 Diagrama de transición de procesos en EMMOS

Los procesos EMMOS se denominan *estáticos* porque poseen una colección de procesos residentes permanentemente en memoria. De esta forma, pierde sentido el concepto de proceso *inactivo* tal como se describe para un diagrama genérico de estados de transición como el de la figura 7.12. En nuestro caso, el SO conoce los atributos de los procesos residentes a través de la tabla de edición de procesos, apéndice H. Así, definiremos como proceso *inactivo* todo aquél que posea entrada en la tabla de edición de procesos. Aquellos procesos que no definen sus atributos en dicha tabla, no podrán ser ejecutados por EMMOS. Como podemos ver, el estado de *inactivo* adquiere otro significado, el de proceso que no está ni preparado ni en ejecución: está disponible.

En nuestro caso el proceso en *ejecución* será, obviamente, aquél que esté preparado y posea la CPU. Dadas las peculiaridades de EMMOS, el concepto de proceso *preparado* también es diferente, ya que, como veremos, se eliminan las listas de procesos preparados para aumentar la eficiencia, y sólo podrá existir, a lo sumo, un proceso preparado de cada prioridad.

El diagrama de transición de estados de EMMOS es el de la figura 7.13. Como podemos ver, no existe el estado de suspensión, dado que los procesos se realizarán, si no son expulsados, enteramente sin necesidad de sincronización con otros, ver sec-

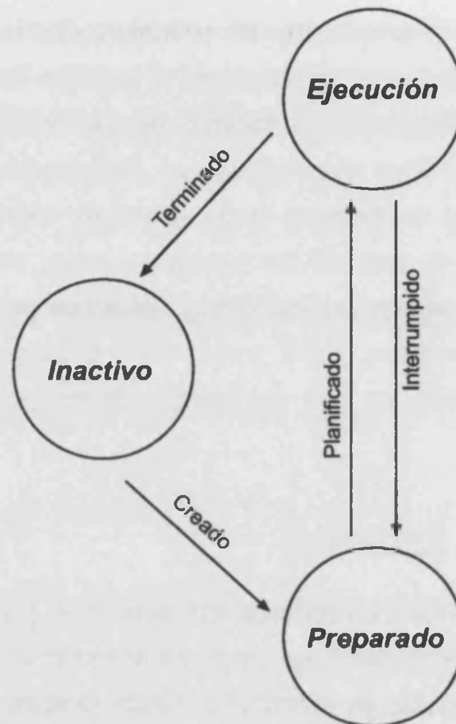


Figura 7.13: Diagrama de transición de estados en EMMOS.

ción 7.7. La sincronización de los procesos viene de la mano de los instantes de llegada de los mensajes, por ello, salvo secciones críticas, no se implementan otros mecanismos de sincronización.

El estado de suspensión se elimina del diagrama de transición de estados.

7.4.4.2 Edición y creación de procesos

Desde el punto de vista de los procesos, EMMOS es un microkernel multitarea que proporciona a los procesos una capa de abstracción del hardware a través de una serie de servicios del sistema. Existen dos tipos de estructuras de datos, las del sistema operativo, que no son accesibles directamente por los procesos pero que pueden ser utilizadas indirectamente por éstos a través de llamadas del sistema (lo cual simplifica la programación) y las estructuras de datos de la aplicación, accesibles directamente por cualquier proceso. Como más adelante veremos, la sincronización de los procesos se proporciona por el SO a través de zonas críticas en las llamadas del sistema, para evitar corromper los contenidos de ciertas estructuras de datos.

7.4.4.2.1 Procedimiento habitual de edición, creación y ejecución de un proceso.

EMMOS sigue un procedimiento de creación de procesos sutilmente diferente del que sería normal en sistemas operativos de escritorio. En éstos, la creación de un proceso se realiza mediante una serie de órdenes tecleadas en un terminal por un usuario que utiliza un sistema interactivo de edición de texto. Así, el usuario introduce y edita desde un terminal, mediante un programa editor de textos, el código fuente de un programa. El archivo producido de esta manera, se procesa en respuesta a una orden del usuario por un *traductor* de lenguaje para producir un archivo objeto. Los errores de sintaxis, si los hay, se corrigen editando el archivo de texto que contiene el código fuente. Tras una compilación satisfactoria, el *enlazador* procesa el archivo del código objeto para producir finalmente un archivo *ejecutable* de programa.

En respuesta a una orden (RUN o similar), el sistema operativo carga en memoria los contenidos del archivo de programa y lo ejecuta. Si éste necesitara una entrada de datos, se le pueden proporcionar interactívamente tecleándolos en el terminal o puede tomarlos directamente de un archivo de datos u otras fuentes. Si se detectan errores en tiempo de ejecución, el programa puede depurarse interactívamente bajo el control de un programa *depurador*.

7.4.4.2.2 Procedimiento de edición y creación de un proceso EMMOS.

EMMOS es un sistema operativo embebido en el subsistema de pesado que puede verse como una "poda" de lo que sería un sistema operativo habitual. Esto es así porque el hecho de funcionar en un entorno tan reducido limita la E/S de datos al CAN. Por otra parte, el abaratamiento de costes redunda en la imposibilidad de utilizar almacenamiento secundario y en el ajuste al máximo de la cantidad de memoria utilizada. Todo esto impide que existan otros servicios más genéricos del sistema operativo, similares a los de sistemas operativos de escritorio.

¿En que redunda entonces la simplicidad del sistema? Ésta proporciona una base para la ejecución de los procesos pero, obviamente, EMMOS no poseerá posibilidades de edición para la programación y creación de procesos (compilación y enlazado). Por ello ha de usarse otra plataforma, por ejemplo el sistema operativo sobre el que opera el control o cualquier otro sistema operativo de escritorio para editar el código, no importa el sistema operativo de la plataforma utilizada. Para realizar la compilación y enlazado ha de utilizarse un sistema operativo MS-DOS, con consola de MS-DOS o emulación de MS-DOS, ya que las herramientas de compilación y enlazado para la familia de procesadores TMS320C2x están disponibles únicamente para este sistema operativo. Así, el código C es compilado para producir un archivo objeto. Si

existen errores de sintaxis en tiempo de compilación, pueden ser corregidos editando de nuevo el archivo de texto que contiene el código fuente, tras lo cual debe volver a recompilarse. Una vez creado el fichero objeto, tras una compilación satisfactoria, el enlazador y los programas de biblioteca procesarán el archivo del código objeto para producir un archivo ejecutable de programa (TI-Sim, 1988; TI-Asm, 1990; TI-C, 1990).

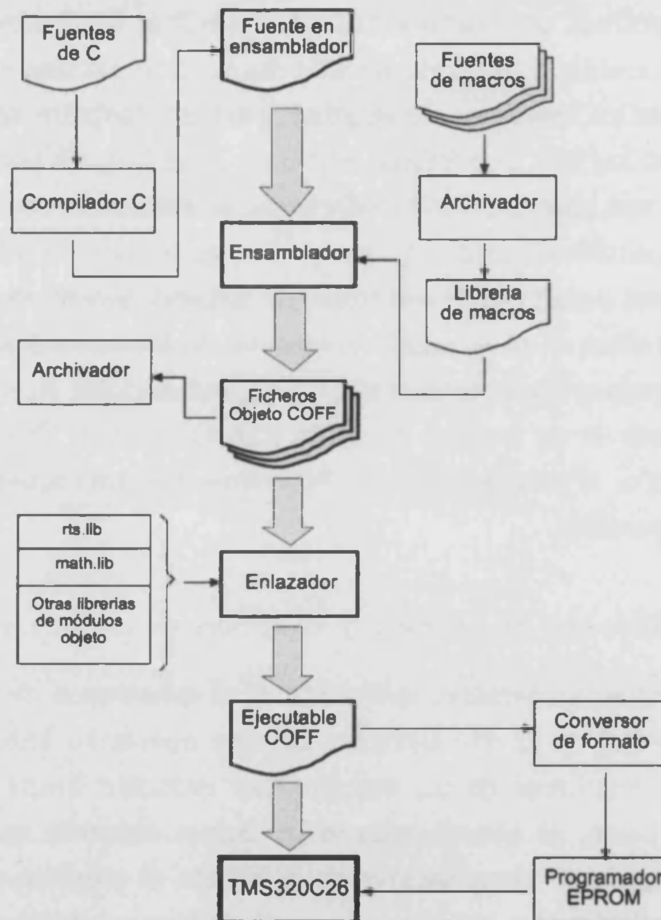


Figura 7.14: Flujo de desarrollo de la aplicación.

Este proceso de compilación se denomina *compilación cruzada*, ya que el código fuente es compilado en una plataforma para ser ejecutado sobre el sistema embebido, cuando de ninguna manera el ejecutable obtenido podría ser ejecutado en el terminal que se usó para su creación, dado que la arquitectura y el juego de instrucciones del procesador *host* es diferente al del 'C26.

El procesador TMS320C26 es soportado por un conjunto completo de herramientas de desarrollo de Texas Instruments que incluyen:

- *Ensamblador*: Traduce los ficheros fuente en lenguaje ensamblador a ficheros

objeto en lenguaje máquina.

- *Compilador de C*: Acepta código fuente en C y produce código fuente en lenguaje ensamblador. El compilador de C tiene 3 partes: el preprocesador, el *parser* y el generador de código.

- *Archivador*: Permite coleccionar un grupo de ficheros en un único fichero archivo (también llamado biblioteca). También permite borrar, añadir, reemplazar y extraer los ficheros miembros de la biblioteca. Una de las aplicaciones más útiles del archivador es construir bibliotecas de módulos objeto, que pueden ser llamadas desde programas en C. También es útil para la creación de bibliotecas personalizadas.

- *Enlazador*: Combina ficheros objeto en un simple módulo ejecutable. Durante la construcción del módulo ejecutable, realiza relocalización y resuelve referencias externas. El enlazador acepta como entrada ficheros objeto relocalizables y miembros objeto de las librerías.

- *Conversor de formato objeto*: Convierte un fichero objeto COFF a formato Intel, Tektronix o TI-tagged que puede ser utilizado por un programador de EPROM.

La figura 7.14 ilustra el flujo del desarrollo de una aplicación. La porción agrupada por el trazo discontinuo indica el camino de desarrollo más común que, como puede verse, comienza por la programación de los procesos en C. Éstos son compilados utilizando las cabeceras "data.h" y "kernel.h", apéndice E, que proporciona información al compilador sobre las estructuras de datos de la aplicación, las llamadas del sistema y las variables públicas del SO (nótese que los procesos programados harán referencia a éstas y el hecho de incluirlas en la cabecera permite al compilador considerar estas referencias como externas, siendo resueltas durante el enlazado).

Una vez obtenido el/los módulos objeto tras la compilación, debe ordenarse al *linker* el enlazado de éstos para formar el ejecutable final. Nótese que:

- El hecho de realizar el desarrollo sobre una plataforma distinta del sistema final implica que ésta no tendrá información sobre la arquitectura y mapas de memoria del sistema destino. En este caso, el enlazador localizaría código y datos a su aire asumiendo que en el sistema destino está disponible todo el rango de direccionamiento de memoria, lo que no es cierto. El enlazador debe, de alguna manera, aceptar información del mapa de memoria del sistema final y algunas instrucciones sobre el orden en que se ha de realizar el enlazado para construir el fichero de salida. Para aportar esta información se utiliza el fichero

de comandos 'dir.cmd', usado por el enlazador para conocer las características del sistema final y la memoria existente tanto en el espacio de direccionamiento de programa como en el de datos. El contenido de este fichero, su significado y la forma de uso se describen en el apéndice D, (TI-C2x, 1993; TI-Asm, 1990).

- Para resolver las referencias a las estructuras de datos del sistema operativo y de la aplicación, el enlazador utiliza el módulo 'sizes.obj'. En éste, el diseñador de sistemas indica, mediante la utilización de secciones (TI-Asm, 1990), la zona de memoria donde cada variable o estructura de datos va a situarse. Así, las estructuras muy usadas y que, por ser críticas, necesiten de un rápido acceso se sitúan en memoria interna, aquéllas menos críticas o que puedan ser penalizadas pueden situarse en memoria externa y aquéllas que no deban volatilizarse ante el apagado del sistema se sitúan en memoria no volátil. Cualesquiera que sean las variables locales que utilice el programador existirán temporalmente en la pila general de memoria interna y caso de utilizar variables globales o no volátiles se gestionan en la sección .bss de memoria externa, véase el apéndice E.

- Al enlazar la aplicación, los procesos contienen llamadas al sistema operativo, por lo que el *linker* debe enlazar los módulos objeto de la aplicación con los módulos objeto del sistema operativo: boot.obj (el cargador secundario), kernel.obj (el sistema operativo) y sizes.obj (gestión de las estructuras públicas y privadas del sistema operativo).

El proceso de enlazado resolverá las referencias cruzadas y tendrá en cuenta la información del fichero de comandos para obtener finalmente un fichero ejecutable en formato COFF. ***Nótese que el programador desarrolla su aplicación basandose en el sistema operativo y las estructuras públicas, y tras el proceso de enlazado queda un sólo ejecutable (sistema operativo + aplicación) que debe ser portado al sistema.***

Carga mediante EPROM

Una vez creado el ejecutable éste ha de ser portado al sistema final para que pueda ser usado para realizar la carga del sistema operativo y la aplicación. El método usado generalmente es la carga del programa a partir de una EPROM.

Para realizar la carga mediante EPROM se han de dar los siguientes pasos:

1. Conversión del fichero ejecutable en formato COFF a un formato inteligible por el grabador de EPROMs utilizado. En nuestro caso se utiliza el formato Intel Intellec.

2. Grabación de la EPROM a partir del ejecutable en formato Intel Intellec.

3. La EPROM, mapeada en la mitad superior del direccionamiento de memoria en memoria global, se coloca en el zócalo del sistema.

4. Se realiza un Reset para proceder a la carga del sistema operativo y la aplicación.

Carga mediante conexión Serie a una CPU.

Éste es el método usado para realizar la carga y ejecución con posibilidades de depuración. Para ello se ha de disponer de una plataforma auxiliar de apoyo, CPU *host*, que posea una consola de MS-DOS o emulación de MS-DOS. Así, si no existe ninguna EPROM en su zócalo y el jumper que habilita la carga serie está colocado, la CPU *host* se comunica a través de un cable serie con el *bootloader* del 'C26 para realizar la carga del sistema operativo y la aplicación en el sistema destino a partir del fichero ejecutable COFF. No obstante, este método también carga un monitor residente en el bloque B0 de la memoria interna del 'C26.

Figura 7.15: Entorno de depuración de la aplicación.

En la plataforma *host* se dispone de un software de depuración (*debugger*) en modo gráfico bajo MS-DOS con el que se puede interaccionar. Éste presenta información, figura 7.15, sobre:

- Contenidos de memoria de datos.
- Programa cargado en el sistema.
- Flags de estado del procesador.
- Registros internos del procesador.
- Registros mapeados en memoria.
- ...

Ya que el *host* y el sistema se conectan en todo momento mediante un cable serie, el debugger permite realizar interacciones con el sistema embebido. Mediante éste podemos realizar órdenes y peticiones como:

- Carga del SO y la aplicación.
- Ejecución.
- Ejecución paso a paso.
- Modificación de cualquier tipo de registros internos y mapeados en memoria.
- Modificación de los flags de estado del DSP.
- Visualización de contenidos de memoria.
- Puntos de ruptura en la ejecución del programa.
- ...

Tanto las peticiones como la visualización se realizan interactivamente en la CPU *host*, que invoca a través de la interrupción hardware /INT2 al monitor residente para ejecutar la acción pedida (siempre que esté colocado el jumper correspondiente a la comunicación serie).

Como vemos, ésta es una forma alternativa a la EPROM de portar el sistema operativo más la aplicación al sistema embebido y que, además, posibilita la depuración de la aplicación. Tales posibilidades de depuración han sido utilizadas para la creación y depuración de EMMOS.

7.4.4.2.3 La creación de los procesos estáticos en memoria

Sea cual sea el procedimiento de carga del sistema operativo y la aplicación, el objetivo es portar el ejecutable SO+aplicación al sistema. Vía serie la carga es más inmediata, y con posibilidades de depuración, pero el sistema ha de estar en todo momento conectado a la CPU *host* mediante cable serie, utilizádose exclusivamente para ello. En general, se utilizará siempre la carga mediante EPROM. De esta forma, tras cada *reset* del sistema, el cargador primario que contiene el 'C26, y es incapaz de cargar la totalidad de la aplicación, cargará poco más que un cargador secundario y una tabla secundaria de vectores de interrupción, tras lo cual cede el control al cargador secundario. El cargador secundario se ocupa de cargar finalmente tanto el sistema operativo como la aplicación a los espacios de memoria correspondientes, véase apéndice F.

Cuando el cargador secundario finaliza la carga, el SO y los procesos de la aplicación quedan emplazados en memoria. Entonces el cargador secundario cede el control a la inicialización del sistema operativo, que realiza:

- Inicialización de los punteros a pila.
- Inicialización de las variables globales.
- Configuración de memoria interna.
- Inicialización de variables y estructuras de datos.
- Habilitación de las interrupciones.
- Programación del timer.
- Inicialización de las comunicaciones CAN.
- Test y programación de los conversores.

Finalizada la inicialización del sistema, se pasa definitivamente el control al proceso nulo, que ocupará la CPU si ningún otro proceso la utiliza. EMMOS aprovecha éste para realizar polling de la cola de mensajes de *background* y servirlos si existe alguno de ellos en la cola, véase apéndice F.

Nótese que *los procesos quedan en memoria estáticamente, es decir, permanecerán inalterados en la misma posición en memoria hasta el apagado del sistema, sin que sean afectados por relocalizaciones, sustituciones, etc. características de la gestión de memoria y planificación a medio plazo, inexistente en el microkernel.* Así, quedan listos para realizar los servicios de las órdenes enviadas vía CAN por el control central de la calibradora. Todos ellos estarán en estado *inactivo* hasta que sean invocados y volverán a dicho estado cuando el proceso finalice.

7.4.4.3 Atributos

El sistema operativo EMMOS no utiliza listas de PCB para mantener propiedades, atributos y contexto de los procesos preparados. La estrategia de planificación seguida actúa más acorde con el reducido tamaño del microkernel y con el contexto de ejecución de la aplicación en estricto tiempo real. Sin embargo aplazaremos la discusión sobre la política de planificación seguida hasta la sección 7.5, donde se verá con más detalle.

Aunque no exista la figura del PCB, los procesos tienen un atributo fundamental, la prioridad. En EMMOS, las prioridades son asignadas por el programador, así como el modo de trabajo en que los procesos pueden ser servidos.

La prioridad

Todos los procesos poseen una prioridad estática (no puede ser modificada en tiempo de ejecución). El planificador concederá la CPU a unos u otros dependiendo de ésta. La figura 7.16 muestra un diagrama que representa esquemáticamente los diferentes niveles de prioridad de EMMOS. En ella pueden diferenciarse dos divisiones:

- Una primera división se establecen entre los niveles de prioridad asociados con una interrupción hardware (los cuatro primeros de arriba a abajo) y los niveles de prioridad software, cuya preferencia es gestionada por el planificador.

- Una segunda división se establece entre los niveles de prioridad atendidos por procesos del sistema operativo, indicados por el relleno gris de los rectángulos, y aquéllos específicos para la aplicación, indicados por los rectángulos en blanco.

La planificación dota a los procesos de una prioridad en la ejecución. Supongamos como ejemplo que se está ejecutando un proceso en el nivel 3, cuando de repente es bombeado un mensaje a la cola de mensajes de nivel 1 (nótese que el proceso de recepción de mensajes es transparente para el proceso que se ejecuta en el nivel 3, ya que se realiza mediante una interrupción hardware de mayor prioridad). Cuando esto sucede y el planificador es invocado, se da cuenta de que un proceso de mayor prioridad requiere la CPU y se la asigna. Cuando acabe la ejecución del proceso en el nivel 1, se restaura automáticamente el contexto del proceso que corría en el nivel 3 y éste retoma la ejecución. Nótese que para el proceso que corre en el nivel 3, la asignación de la CPU a un proceso de nivel 1 ha sido transparente a él como si de una interrupción hardware se tratara.

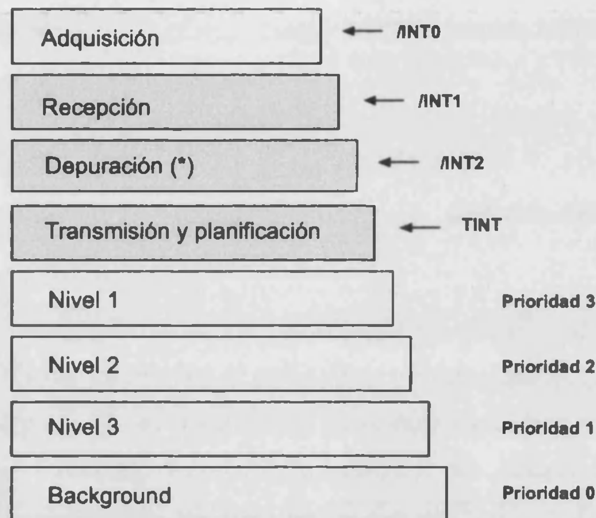


Figura 7.16: Esquema de la jerarquía de los diferentes niveles de prioridad.

La tabla de edición de procesos

Cuando el sistema recibe una orden a través de un mensaje (paquete de bytes formado por una cabecera más un conjunto de datos), el sistema operativo debe encargarse de planificar el servicio de esta orden. Para ello debe ser capaz de obtener, para cada orden, información de la prioridad de su servicio, el puntero al proceso estático de servicio en memoria y si éste ha de ejecutarse en algún modo de funcionamiento particular.

Esta información debe ser introducida por el programador y da lugar a lo que desde el punto de vista del programador se denomina "tabla de edición de procesos". Por ello, cada vez que el programador quiera añadir un proceso, además de programarlo debe indicar sobre la tabla los atributos del mismo.

```
TABLE_BEGIN

  CASE 0x30 PRIORITY 1 ADDRESS ProcessA CEND
  CASE 0x40 PRIORITY 2 ADDRESS ProcessB CEND

TABLE_MIDLE

  MODE0_BEGIN
    CASE 0x330 PRIORITY 1 ADDRESS GetA CEND
    CASE 0x340 PRIORITY 2 ADDRESS MensA CEND
  MODE_END
  MODE1_BEGIN
    CASE 0x350 PRIORITY 1 ADDRESS GetB CEND
    CASE 0x360 PRIORITY 3 ADDRESS StopA CEND
  MODE_END
  MODE2_BEGIN
```

```
      CASE 0x370 PRIORITY 1 ADDRESS PutC CEND
      CASE 0x380 PRIORITY 0 ADDRESS Stop CEND
    MODE_END

  TABLE_END

  VEC_ADQ_IS putmens VEC_ADQ_END
```

Código 7.1 Tabla de edición de procesos.

En la tabla, el programador ha de indicar en la primera mitad (entre TABLE_BEGIN y TABLE_MIDLE) los procesos que sean amodales, es decir, que se pueden ejecutar cualquiera que sea el modo de funcionamiento del sistema. En la mitad inferior (entre TABLE_MIDLE y TABLE_END) se han de indicar aquellos procesos que el programador quiere que sólo se ejecuten si el sistema está en un modo concreto de funcionamiento. Esto se hace incorporando los procesos entre MODEx_BEGIN y MODE_END del modo en que se pretenda hacer posible la ejecución del proceso. Hay diez posibles modos (de MODE0 a MODE9), véase apéndice H.

La declaración de los procesos describe la orden que los invoca, la prioridad que les asigna el programador y la dirección donde se encuentra el código del proceso estático en memoria.

```
CASE (número de orden) PRIORITY (prioridad) ADDRESS (Nombre del proceso que lo sirve) CEND
```

Aunque la dirección del proceso no se especifica directamente, el nombre del proceso es utilizado por el enlazador para obtener esta dirección en su lugar.

Finalmente, también es posible que el programador asocie uno de sus procesos al servicio de la adquisición (proceso de máxima prioridad), a través de la sentencia:

```
VEC_ADQ_IS Process VEC_ADQ_END
```

donde *Process* es el nombre del proceso de servicio de la interrupción. Se asume que dicha función no retorna valores ni acepta argumentos, como se indica en el siguiente prototipo de función:

```
void Process()
```

Nótese como la tabla de edición de procesos sirve para aportar al sistema operativo una información que éste no puede recolectar dinámicamente en tiempo de ejecución: Prioridades, modo de funcionamiento, proceso de servicio de cada orden y proceso servidor de la interrupción de adquisición.

7.4.4.4 La creación de una aplicación bajo EMMOS

Para ilustrar la facilidad que EMMOS proporciona para la creación de aplicaciones, vamos a crear una muy sencilla aplicación sobre EMMOS. El código 7.2 lista el código de dicha aplicación. Como podemos ver, la tabla de edición de procesos indica que el proceso de servicio de la interrupción de conversión es 'intadq'. Éste recoge las muestras de los 10 conversores del sistema, que ya consideramos programados por simplicidad, y las introduce en los buffers circulares de muestras correspondientes para cada línea, siempre y cuando 'booleano' sea falso. Además de la recogida de muestras, se incrementa un contador ('contador') que servirá para contar las veces que se ha atendido el servicio de interrupción.

A continuación, vamos a crear un interfaz con el control, para que éste pueda gestionar fácilmente el funcionamiento del sistema embebido. Para ello creamos los siguientes procesos de servicio:

- **SendContador.** Proceso de servicio que es invocado cada vez que se recibe un mensaje (sin datos) con el número de orden 0x10. Éste envía un mensaje al control (con número de orden 0x208) que comunica el número de veces que se ha atendido el servicio de la interrupción, 'contador'. Como puede verse en la tabla de edición de procesos, se le asigna la mayor de las prioridades.

- **ResetContador.** Proceso de servicio invocado por un mensaje entrante (sin datos) con el número de orden 0x20. Éste sólo inicializa la variable 'contador' a cero. Como puede verse en la tabla de edición de procesos, se le asigna prioridad 2.

- **SendBooleano.** Es invocado cada vez que se recibe un mensaje (sin datos) con el número de orden 0x30. Éste envía un mensaje al control (con número de orden 0x210) que comunica el valor que contiene esta variable. Se le asigna prioridad 1.

- **ChangeBooleano.** Es invocado cada vez que se recibe un mensaje con el número de orden 0x30 (el mensaje incorpora un dato). Éste asigna a 'booleano' el valor que trae consigo el mensaje. Se le asigna prioridad 0, por lo que se ejecutará siempre que no exista otra orden pendiente de servicio.

Todos los procesos descritos son amodales, es decir, pueden ejecutarse en cualquier momento, ya que en este ejemplo no se han definido modos de funcionamiento. Como podemos ver, EMMOS realiza el envío de mensajes, su recepción, la invocación de los procesos de servicio y la planificación de éstos, de forma transparente para la aplicación.

Este sencillo código implementa un interfaz que permite al sistema ejecutar las órdenes del control, realizando multitarea. Cabe remarcar de nuevo que esta aplicación trivial se adjunta a modo de ejemplo con el único propósito de ilustrar la utilización de la tabla de edición de procesos y la sencillez de programación bajo el sistema operativo EMMOS[2]. Las aplicaciones más complejas tienen a su disposición toda una serie de servicios del sistema y estructuras de datos a su disposición, apéndice E.

```
#include "kernel.h"
#include "data.h"

/*________TABLA DE EDICICÓN DE PROCESOS______*/

TABLE_BEGIN
   CASE 0x10 PRIORITY 3 ADDRESS SendContador CEND
   CASE 0x20 PRIORITY 2 ADDRESS ResetContador CEND
   CASE 0x30 PRIORITY 1 ADDRESS SendBooleano CEND
   CASE 0x40 PRIORITY 0 ADDRESS ChangeBooleano CEND
TABLE_MIDLE
   MODE0_BEGIN
   MODE_END
   MODE1_BEGIN
   MODE_END
   MODE2_BEGIN
   MODE_END
TABLE_END

VEC_ADQ_IS intadq VEC_ADQ_END

/*_______________________________________________*/

int genchanoffset=0;
int contador=0;
int booleano=0;

void intadq()
{
   extern int LONGCANAL;
   extern void *puntarray[19];
   int res,i,*inicio,*punt;

   if (!booleano)
   {
      /*Se itera para todas las líneas.*/
      for (i=0;i¡10;i++)
      {
         /*Se toma el dato y se encola en el buffer
         circular de muestras de la línea corresp.*/
         res=recibe(i);
```

[2]El proceso de servicio de la interrupción del conversor debe realizarse siempre con las interrupciones deshabilitadas por ser el servicio de mayor prioridad de todos, no puediendo contener una instrucción tipo asm("EINT").

```
                inicio=(int *)((int)puntarray[7+i]);
                punt=(int *)(((int)inicio)+genchanoffset);
                *punt=res;
            }
            /*Offset para la próxima muestra.*/
            genchanoffset++;
            if (genchanoffset¿=((int)&LONGCANAL)) {genchanoffset=0;}
        }
        /*Se incrementa el contador de muestras recogidas.*/
        contador++;
    }

void SendContador(int linea)
{
    int ade[20];
    ade[0]=0x0001;
    ade[1]=208;
    ade[2]=0x0000;
    ade[3]=0x0000;
    ade[4]=0;
    ade[5]=contador;
    putmens(4,0,ade);
}

void ResetContador(int linea)
{
    contador=0;
}

void SendBooleano(int linea)
{
    int ade[20];
    ade[0]=0x0001;
    ade[1]=210;
    ade[2]=0x0000;
    ade[3]=0x0000;
    ade[4]=0;
    ade[5]=booleano;
    putmens(4,0,ade);
}

void ChangeBooleano(int linea, int dato1)
{
booleano=dato1;
}
```

Código 7.2. Ejemplo de aplicación.

7.4.4.5 La ejecución de los procesos

En los sistemas operativos convencionales, se utiliza la orden RUN para preparar un proceso para su ejecución. Sin embargo, dadas las peculiares características de su entorno de ejecución, EMMOS no posee los módulos de edición e intérprete de

comandos. En éste, una orden viene dada por la recepción de un mensaje. Así, un mensaje entrante invocará un proceso de servicio de la mano del planificador, que asignará a éstos la CPU en función de la prioridad de la orden, permitiendo así realizar multitarea.

7.5 El planificador

La planificación hace referencia a un conjunto de políticas y mecanismos incorporados al sistema operativo, y por los cuales se rige el orden en que se completa el trabajo que hay que realizar. Como ya sabemos, el módulo del SO encargado de la selección de la siguiente tarea a realizar es el *planificador.*

El objetivo primario de la planificación es optimizar el rendimiento del sistema, de acuerdo con los criterios considerados más importantes para su diseño.

7.5.1 Tipos de planificadores

En los sistemas operativos podemos encontrar hasta tres tipos distintos de planificadores. Sólo los sistemas operativos más complejos disponen de los tres tipos de planificación, que se denominan: *planificadores a largo, medio y corto plazo.* Sin embargo, los dos primeros tipos sólo serán descritos brevemente, dado que el sistema final que nos ocupa es un reducido sistema embebido de tiempo real que no necesita del uso y la complejidad añadida de la planificación a largo y a medio plazo.

El planificador a largo plazo

El planificador a largo plazo trabaja, cuando está presente, con una cola de lotes, encargándose de seleccionar el siguiente lote que hay que ejecutar. Los lotes se reservan básicamente para programas de baja o muy baja prioridad, intensivos en recursos (tiempo de la CPU, memoria, dispositivos especiales de E/S) y que pueden ser utilizados para mantener ocupados a los recursos del sistema durante períodos de baja actividad. Un lote contiene todos los datos y órdenes necesarias para su ejecución.

Cuando existe, el planificador a largo plazo también interviene en las transiciones de procesos inactivos a procesos preparados, colocando el PCB del proceso en la lista de procesos preparados para su consideración por el planificador a corto plazo.

El planificador a medio plazo

Un proceso en ejecución puede quedar en suspenso al hacer una petición de datos de E/S o emitir una llamada al sistema. Dado que los procesos en suspenso no pueden

realizar ningún progreso hacia su terminación, hasta que se elimina la condición suspensiva correspondiente, a veces resulta beneficioso sacar dichos procesos de memoria principal con el fin de liberar espacio que podrían ocupar otros procesos. Como el tamaño de la memoria principal es limitado se impone un límite al número de procesos residentes en ella.

En este caso, se plantea la liberación del espacio de memoria principal ocupado por el proceso en suspenso almacenándose una imagen en memoria secundaria. A este mecanismo se le denomina ***intercambio de memoria***, y se dice que el proceso *se descarga de la memoria al almacenamiento auxiliar.*

El planificador a medio plazo, cuando existe, se encarga de atender a los procesos descargados al almacenamiento auxiliar. Una vez que se elimina la condición que suspendió el proceso, el planificador a medio plazo se encarga de intentar asignarle una cantidad de memoria principal, cargarlo de nuevo en memoria y prepararlo.

El planificador a corto plazo

Éste es el planificador fundamental. Existe en todos los sistemas operativos multiproceso como parte fundamental. Su misión es asignar la CPU entre los procesos preparados que residen en memoria. Su objetivo principal es maximizar el rendimiento del sistema de acuerdo con el conjunto de criterios elegido. Al tener a su cargo las transiciones de estado de los procesos preparados a procesos en ejecución, el planificador a corto plazo tiene que ser invocado una vez por cada cambio de proceso, para elegir el proceso siguiente que hay que ejecutar.

En la práctica, el planificador a corto plazo es invocado cuando un evento, interno o externo, cambia el estado global del sistema. Como cualquier cambio ocurrido puede haber dado lugar a la suspensión del proceso en ejecución o de uno o más procesos preparados, se debe pasar el control al planificador a corto plazo para determinar si efectivamente se han producido dichos cambios y, en caso afirmativo, elegir el proceso siguiente que hay que ejecutar.

Algunos de los eventos que pueden dan lugar a la replanificación son:

- Señales de reloj (interrupciones temporizadas basadas en el tiempo).
- Interrupciones.
- La mayoría de las llamadas operacionales al SO.
- Transmisión y recepción de señales de sincronización.
- ...

En general, siempre que se produce uno de estos eventos, el SO invoca al planificador a corto plazo para determinar si se debe planificar otro proceso para su ejecución.

También la creación de un proceso o la reanudación, que añaden un proceso a la lista de procesos preparados, invoca al planificador para verificar si la nueva entrada debe convertirse también en el proceso en ejecución. Suspender un proceso, un cambio dinámico de la prioridad de un proceso, terminar un proceso en ejecución o una finalización anormal de éste, son también eventos que hacen necesaria la invocación del planificador para la elección de un nuevo proceso para su ejecución.

7.5.2 Criterios de planificación y rendimiento

Esta sección trata de exponer algunas medidas del rendimiento y criterios de optimización que se utilizan para maximizar el rendimiento de los planificadores del sistema. Sin embargo, las medidas y criterios utilizados suelen cambiar de un tipo de sistema operativo a otro. La relación de las que se encuentra con más frecuencia en sistemas operativos de propósito general incluye:

- U*tilización de la CPU*: Es la fracción media de tiempo durante la que está ocupada la CPU. La ocupación hace referencia al tiempo en que la CPU no funciona en vacío. Éste es el tiempo invertido tanto en la ejecución de los procesos como en el SO. Como alternativa se puede considerar "utilización útil" sólamente y excluir así el tiempo invertido en la ejecución del SO. En cualquier caso, para obtener un buen rendimiento, la idea es mantener la CPU tan ocupada como sea posible.
- *Productividad*: Se refiere a la cantidad de trabajo realizada por unidad de tiempo. Una manera de expresar la productividad es por medio del número de trabajos usuarios ejecutados en una unidad de tiempo. Cuanto mayor sea el número, más trabajo se hace aparentemente por el sistema. Sin embargo, la productividad así definida es difícil de usar para realizar comparaciones porque depende de las características y las necesidades de recursos de los procesos que se consideran. Para realizar comparaciones sobre el rendimiento de distintos algoritmos de planificación, se debe presentar una carga idéntica de trabajo.
- *Tiempo de espera*: Esencialmente es el tiempo que pasa un proceso esperando la asignación de recursos debido a la pugna con otros trabajos. En otras palabras, es la penalización impuesta por compartir recursos con otros.
- ...

Sin embargo, nuestro interés no es genérico, sino que incide directamente sobre los sistemas operativos de tiempo real, y en éstos algunas de las medidas y criterios utilizados suelen ser particulares de este tipo de sistemas. Para las aplicaciones que

deben ejecutarse bajo estricto tiempo real no sólo importan los resultados computados, sino también el instante en que se producen. Por ello cada aplicación tiene unos límites máximos absolutos que deben ser satisfechos para su correcto funcionamiento. Antes de continuar con los criterios de rendimiento, comentaremos algunos de los conceptos asociados a sistemas de estricto tiempo real (Quinnell, 1995).

Tiempo de respuesta El *tiempo de respuesta* se utiliza con frecuencia tanto en sistemas de tiempo compartido como de tiempo real. Sin embargo, las definiciones y las constantes de tiempo que intervienen en los dos sistemas son bastante distintas. En los sistemas de tiempo compartido, se puede definir como el tiempo que transcurre desde el momento en que se da entrada al último carácter de una línea de orden de lanzamiento de un programa hasta que aparece el primer resultado en el terminal. A esto sé le llama usualmente el *tiempo de respuesta del terminal.*

En los sistemas de tiempo real, el *tiempo de respuesta a un evento* es el tiempo que transcurre desde el momento en que se señala el evento hasta que se ejecuta la primera instrucción de su rutina de servicio, figura 7.17. Es obvio, que un aumento indiscriminado de este parámetro puede provocar pérdida de eventos, con el consiguiente fallo de la aplicación.

Latencia de Interrupción Cuando un proceso está en ejecución y sucede una interrupción, el servicio de la interrupción se debería realizar rápidamente. Sin embargo, dado que algunas operaciones del sistema se ejecutan con las interrupciones deshabilitadas para no ser interrumpidas, el peor de los tiempos de respuesta ocurre al suceder el evento cuando el sistema está activo. Por esto, debe ser considerado el mayor intervalo temporal de ejecución no interrumpible dentro de una llamada del sistema. Este tiempo se denomina *latencia de interrupción.*

El tiempo de respuesta de una interrupción depende del manejo realizado de las rutinas de servicio de interrupción (ISR): Generalmente, una interrupción dispara una rutina de servicio y esto deshabilita automáticamente todas las interrupciones. Este 'primer nivel' de interrupciones debe mantenerse tan corto como sea posible para no poner en peligro los tiempos de respuesta de otras fuentes de interrupción. Por ello, en este 'primer nivel', el dato suele ser colocado en un buffer para ser procesado en un nivel superior, reduciendo considerablemente el tiempo de respuesta de otras fuentes de interrupción respecto al caso en el que el servicio de interrupción se sirve con las interrupciones deshabilitadas. Cuando se dan varias fuentes de interrupción con frecuencias divergentes, el

sistema debe estar preparado para manejar el caso en que todas ellas ocurran simultáneamente.

El uso de procesadores con dos o más bancos de registros para conmutación de contexto, uno de ellos para el contexto del SO y el resto para los procesos, permite la obtención de mejores tiempos de respuesta. Estos deben, en general, estar reservados para las interrupciones más críticas.

Retardo entrada - salida Dependiendo de la aplicación, existe un límite máximo del retardo entre el momento del muestreo y el momento en el que los resultados estén disponibles. Este retardo no debería ser superado nunca para asegurar el correcto funcionamiento de la aplicación. Esto es patente en sistemas donde hay lazos de realimentación como en control o sistemas de audio profesional, donde el retardo permisible entre entrada y salida es del orden de una o dos muestras, es decir, de unas pocas decenas de microsegundos.

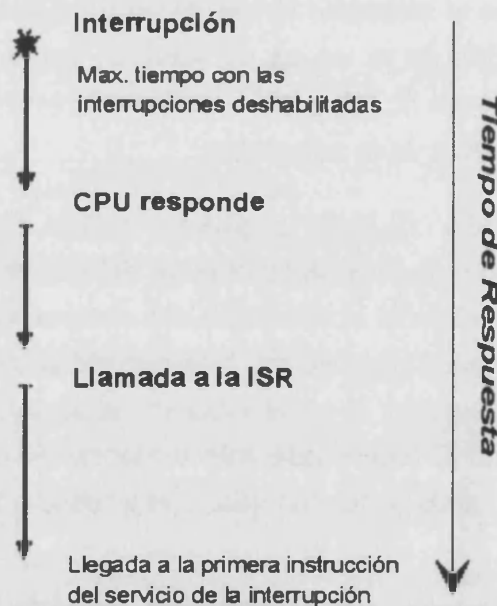


Figura 7.17: Tiempo de respuesta.

Como vemos, el sistema operativo utilizado va a jugar un papel fundamental para el correcto funcionamiento de la aplicación, lo cual justifica plenamente la necesidad de sistemas operativos muy especializados.

Las magnitudes empleadas por los diseñadores de RTOS (Sistemas operativos de tiempo real, *Real Time Operating Systems*) para evaluar las prestaciones de sus productos, son referencias temporales como:

- **Latencia de interrupción**: Máximo intervalo temporal en el que el código correspondiente al sistema se ejecuta con las interrupciones deshabilitadas.
- **Latencia de planificación**: Mide el máximo tiempo en que una tarea expulsa a otra de menor prioridad.
- **Reanudación por interrupción**: Tiempo entre la ocurrencia de una interrupción que despierta una tarea y el instante en que ésta comienza a ejecutarse.
- **Conmutación de contexto**: Tiempo necesario para realizar una conmutación de contexto.

Sin embargo, no son éstas las únicas referencias usadas. Cabe la posibilidad de usar otras, como por ejemplo: tiempo de conmutación entre tareas con la misma prioridad tras un evento de suspensión de una de ellas, tiempo necesario para poner un semáforo, etc. En cualquier caso estas referencias pretenden medir las prestaciones del RTOS. En general, se deben proporcionan medidas conservadoras tomando siempre el peor de los casos y/o describiendo cómo ha sido tomada la medida.

Cabe remarcar que los criterios tomados dependerán en gran medida del tipo de sistema operativo que se utilice. Por ejemplo, la productividad y la utilización de componentes son los objetivos primarios en un sistema por lotes, mientras que los sistemas multiusuario de uso compartido están dominados por preocupaciones acerca del tiempo de respuesta de los terminales, y los sistemas operativos de tiempo real se los diseña para tener la posibilidad de atender de manera sensible una multitud de eventos externos.

7.5.3 Tipos de planificadores

Puede establecerse una importante división entre esquemas de planificación según se realice sustitución o no. Así, podemos hablar de disciplinas de planificación con y sin derecho preferente. Al aplicarlo a la planificación a corto plazo, la *no sustitución* implica que el proceso en ejecución conserva el disfrute de los recursos asignados, incluida la CPU hasta que dicho proceso en ejecución cede voluntariamente el control al SO. Esto es cierto incluso cuando existan eventos de prioridad superior. En otras palabras, al proceso en ejecución no se le puede forzar a renunciar a la posesión de la CPU cuando un proceso de prioridad más alta queda preparado para su ejecución. No obstante, cuando el proceso en ejecución queda en suspenso, como resultado de su propia acción, se puede planificar otro proceso ya preparado.

Por otro lado, la *planificación con derecho preferente* puede sustituir en cualquier momento un proceso en ejecución por otro de prioridad más elevada, para lo cual se

ha de activar el planificador siempre que se detecta un evento que cambia el estado del sistema. Como es obvio, este tipo de planificación hace necesaria una ejecución más frecuente del planificador (Tanenbaum, 1992; Milenkovic, 1988).

En la literatura también puede encontrarse la denominación *planificación expulsiva* para hacer referencia a la *planificación con derecho preferente,* y *planificación no expulsiva* para hacer referencia a la *planificación sin derecho preferente.*

Planificación de servicio por orden de llegada (FCFS, «First-Come-First-Served»)

Ésta es, con diferencia, la disciplina de planificación más simple. La carga de trabajo se procesa tal como llega, sin sustitución por derecho preferente. La implantación del planificador FCFS es bastante directa y su ejecución conlleva pocos gastos generales. Como se puede esperar, este tipo de planificación también resulta, usualmente, en un bajo rendimiento.

En particular, los procesos cortos pueden sufrir retrasos considerables cuando existen uno o más procesos largos en el sistema.

Sólo se invoca al sistema operativo para procesar las transiciones de estado y la conmutación (cambio) de procesos. La planificación FCFS elimina la noción e importancia de las prioridades de los procesos: el instante de llegada es el único criterio de planificación.

Planificación por el orden del menor tiempo restante (SRTN, «Shortest Remaining Time Next»)

La planificación por el orden del menor tiempo restante (SRTN) es una esquema de planificación que elige el siguiente proceso a ejecutar, en virtud de que el tiempo de ejecución que le resta sea el más corto, figura 7.18. La planificación SRTN se puede implantar en las variedades con derecho preferente y sin él. En uno u otro caso, siempre que se invoca la planificación SRTN, examina la cola correspondiente (de trabajos o procesos por lotes o preparados) para hallar el proceso o trabajo que permita el menor tiempo restante de ejecución. La diferencia entre los dos casos reside en las condiciones que conducen a la invocación del planificador y, por consiguiente, la frecuencia de su ejecución. Sin derecho preferente, al planificador SRTN se le invoca siempre que se completa un trabajo o el proceso en ejecución cede el control al SO. En la versión con sustitución por derecho preferente, que sólo tiene sentido para la planificación a corto plazo, siempre que se produce un evento que hace que un nuevo proceso quede preparado, se invoca al planificadorpara para comparar el tiempo restante de ejecución que corresponde al proceso en ejecución, con el tiempo

que se necesita para completar la ejecución del nuevo proceso.

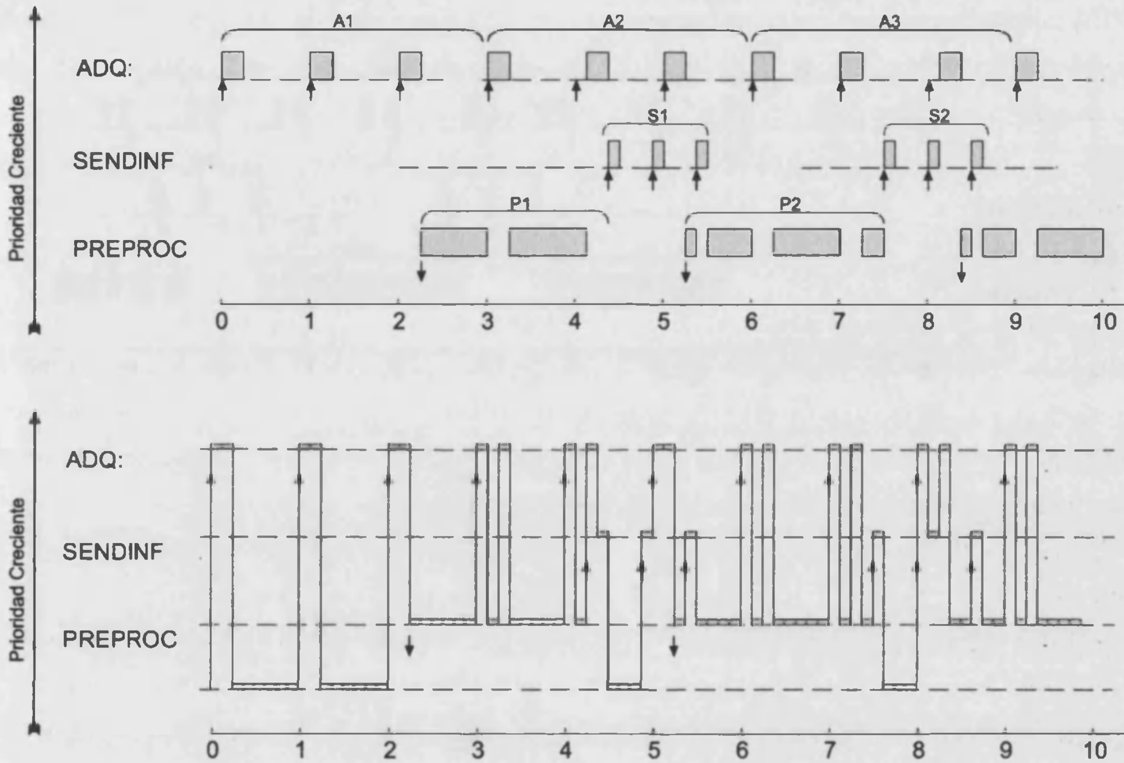
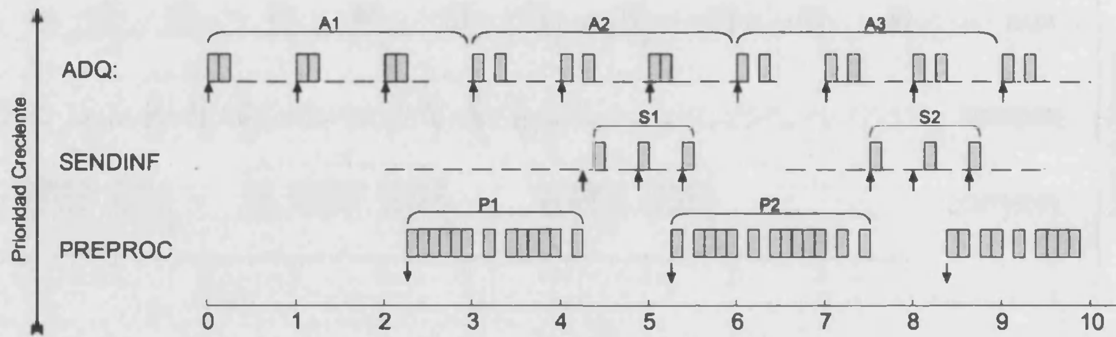


Figura 7.18: a) Planificación SRTN. b) Diagrama *prioridad versus tiempo de proceso*.

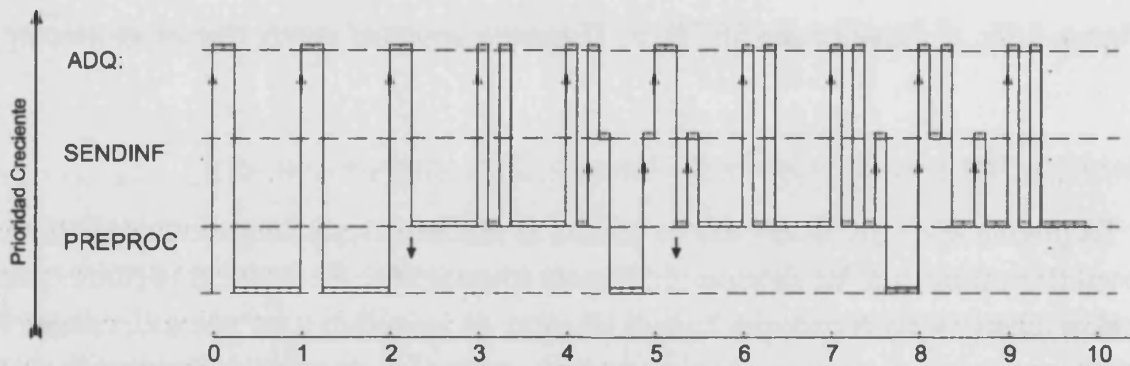
Planificación por fracciones de tiempo (RR, circuito cíclico)

La planificación por fracciones de tiempo es una estrategia muy usada en entornos interactivos como son los sistemas de tiempo compartido. En estos, el requisito principal es ofrecer a los terminales buenos tiempos de respuesta y, en general, compartir equitativamente los recursos del sistema entre todos los usuarios. Obviamente estos objetivos sólo pueden llevarse a cabo mediante esquemas de planificación con derecho preferente. Uno de los más populares es el de *división del tiempo en fracciones*, al que también se denomina *circuito cíclico* (RR '*Round-Robin*').

La planificación por fracciones de tiempo se basa en la división del tiempo de la CPU en fracciones, que se asignan a los procesos peticionarios. Ningún proceso podrá ejecutarse durante más de una fracción de tiempo si existen otros esperando en la cola de procesos preparados. Si un proceso necesita más de una fracción dé tiempo para completar su ejecución después de agotar su fracción de tiempo, se le coloca al final de la cola de trabajos preparados para que espere a la asignación siguiente. Al



a)



b)

Figura 7.19: a) Planificación Round Robin para T=0.125 unidades. b) Diagrama *prioridad versus tiempo de proceso.*

contrario, si el proceso en ejecución cede el control al SO antes de llegar al final de su tiempo asignado, se declara un evento y se planifica para su ejecución el siguiente proceso preparado. Así, el tiempo de la CPU se asigna a los procesos sobre la base de una prioridad rotativa, recibiendo cada uno de ellos el l/N del tiempo de la CPU, con N el número de procesos preparados, figura 7.18.

La implantación de la planificación de circuito cíclico requiere el soporte de un temporizador de intervalos que invoca al planificador. Éste se limita a almacenar el contexto del proceso en ejecución, transferirlo al final de la cola de procesos preparados y poner en marcha un nuevo proceso. Si un proceso cede el control al SO antes de finalizar su tiempo, se inicia el temporizador con el fin de dar al nuevo proceso en ejecución la fracción de tiempo completa.

La planificación RR depende de una juiciosa elección de la fracción de tiempo para la obtención de un adecuado rendimiento, ya que una fracción de tiempo demasiado corta da por resultado excesivas tareas relativas al cambio de contexto, mientras que una fracción de tiempo demasiado larga degenera de planificación RR a planificación FCFS, ya que los procesos ceden el control al SO en lugar de ser sustituidos por otros con derecho preferente por el temporizador de intervalos. La duración de la fracción de tiempo es un parámetro regulable del sistema con posibilidad de modificación dinámica.

Planificación con derecho preferente basado en la prioridad (iniciada por eventos)

Anteriormente ya se presentó la planificación con derecho preferente basada en la prioridad, figura 7.11. Este tipo de planificación permite a los programadores influir en el orden en el que el planificador da servicio a los acontecimientos externos mediante la asignación de prioridades a los procesos. Esta planificación, denominada también *planificación expulsiva basada en la prioridad,* se caracteriza por sus excelentes y predecibles tiempos de respuesta a eventos de alta prioridad, baja latencia de interrupciones y elevado ancho de banda de E/S.

En principio, a cada proceso se le asigna un nivel de prioridad y, siempre que se produce un evento significativo, el planificador elige para ejecución el proceso preparado que tiene la máxima prioridad. Las prioridades pueden ser estáticas y dinámicas. En cualquiera de los dos casos, los valores iniciales los asigna el usuario o el sistema en el momento de la creación de los procesos. El nivel de prioridad puede considerarse como una cifra global sobre la base de las características del proceso, las necesidades de recursos y el comportamiento del proceso en cuanto al tiempo de ejecución.

7.5.4 La planificación EMMOS

Anteriormente ya se introdujo que la planificación utilizada por EMMOS carece de la típica lista de descriptores o bloques de control de proceso para los procesos preparados. EMMOS acelera el proceso de almacenamiento del contexto, guardandolo en la pila, como más adelante veremos.

La planificación se basa en el temporizador hardware. El servicio del timer se activa periódicamente e invoca al planificador si existen procesos de mayor prioridad a los que asignar la CPU. Los procesos finalizados ponen el contador del timer a un valor bajo para que el planificador sea invocado inmediatamente tras la habilitación de las interrupciones.

7.5.4.1 Política o estrategia de planificación.

La figura 7.16 ilustra la jerarquía de prioridades, tanto software como hardware. Las interrupciones hardware no necesitan ser planificadas, ya que la invocación jerárquica de sus procesos de servicio se realiza mediante mecanismos hardware. Las interrupciones hardware del ***timer***, **/int1** e **/int2** (que realizan la gestión de planificación y transmisión, recepción y depuración) son privadas del sistema operativo. Esto es, los procesos de servicio están predefinidos y son inalterables por el programador. Estos son ejecutados en su totalidad con las interrupciones deshabilitadas, por lo que en estos niveles no existe planificación expulsiva por procesos de mayor prioridad. Sin embargo, no es mucha la latencia que esto introduce, por lo que no afectará sustancialmente al funcionamiento en tiempo real de la aplicación.

Los procesos no son llamados por órdenes escritas en un editor de comandos, sino por órdenes recibidas por el sistema en forma de mensajes a través de CAN. Así pues, cada mensaje CAN recibido empaqueta una orden, que necesita ser servida, y los datos que ésta necesita para su ejecución.

EMMOS dispone de una cola de mensajes para cada prioridad software. Así, existen cuatro colas de mensajes: cola de mensajes de nivel 1, cola de mensajes de nivel 2, cola de mensajes de nivel 3 y cola de mensajes de *background*. En el modo de recepción por defecto, cuando se recibe una orden en forma de mensaje, ésta es bombeada a la cola de mensajes correspondiente a la prioridad de su servicio. La variable pública del sistema operativo 'fintprior' dispone de un bit para cada una de estas colas de mensajes, que se activa cuando es apilado un mensaje y se desactiva cuando es desapilado el último mensaje de la cola.

Las colas de mensajes son usadas para contener órdenes de la misma prioridad. Así, en cualquier momento puede haber una, ninguna o varias órdenes en cada una de las colas. Ésta va a ser una idea clave para la eliminación de las listas de procesos

preparados. Así, cuando se invoca el planificador, éste sólo necesitará observar los flags de las colas de mensajes de las colas de prioridad mayor que la del proceso en ejecución para saber si se necesita una planificación. Sin embargo, cuando existan varios mensajes en una cola, no existirá multiproceso entre ellos. Entre mensajes de la misma prioridad se establece una planificación FCFS, donde el primero en llegar será el primero que será servido.

Podríamos definir el tipo de planificación que EMMOS realiza como una ***planificación expulsiva (con derecho preferente) basada en prioridades con ejecución FCFS de los procesos de la misma prioridad.***

7.5.4.2 Ejemplo de planificación

Para ilustar la idea de la planificación EMMOS pongamos un ejemplo en el que la CPU está asignada a un proceso de nivel 3 como respuesta a una orden de esta prioridad que se apiló en la cola de prioridades de nivel 3. Asumamos que no existe ningún otro mensaje en el resto de las colas de mensajes de prioridad. Inicialmente, las invocaciones periódicas del planificador no observan necesidad de planificar este proceso para asignar la CPU a otro, por lo que el proceso de nivel 3 seguirá ocupando la CPU y progresando hacia su finalización. Supongamos que se recibe el último byte de un mensaje entrante y el servicio de recepción bombea éste a la cola de mensajes de nivel 2, de mayor prioridad que el proceso que ocupa la CPU. Sin embargo, figura 7.20, puede verse como la CPU continua en posesión del proceso de nivel 3 hasta la llegada de la próxima interrupción del timer, que invoca al planificador. Éste determina que la CPU ha de ser asignada al proceso de prioridad 2 y se la asigna. Supongamos ahora que se recibe el último byte de otro mensaje entrante y el servicio de recepción bombea éste a la cola de mensajes de nivel 1, de nuevo de mayor prioridad que el proceso que ocupa la CPU. Nótese de nuevo como la CPU no le será asignada a éste hasta la próxima invocación del planificador. En el nivel 1, la CPU ya no puede ser arrebatada por planificación, aunque sí por cualquiera de las interrupciones hardware soportadas. El proceso progresará hacia su finalización, restaurando tras ella el proceso de nivel 2 anterior. Asumiendo que no se produce ninguna recepción de mensajes, el proceso de nivel 2 finalizará y restaurará el contexto del primigenio proceso de nivel 3, que de nuevo podrá ejecutarse hasta su finalización. Cuando ya no existen mensajes que atender en las colas de mensajes, la CPU será finalmente tomada por el proceso nulo en espera de cualquier otra orden.

El diagrama prioridad versus tiempo de proceso de la figura 7.20 ilustra el ejemplo anterior. No obstante, nótese que la recepción y apilamiento de un mensaje, que en esta figura se indica como realizado por un único servicio de interrupción, lo que es una

simplificación del caso real. Una recepción de mensaje, realmente presentaría varios segmentos de ejecución de la rutina de recepción, ya que el mensaje debe ser formado byte a byte hasta completarlo. Este comportamiento es descrito por la figura 7.21, donde puede observarse que cuando el servicio de recepción capta el último byte, ocupa más tiempo de CPU porque también bombea el mensaje entrante a la cola de mensajes correspondiente.

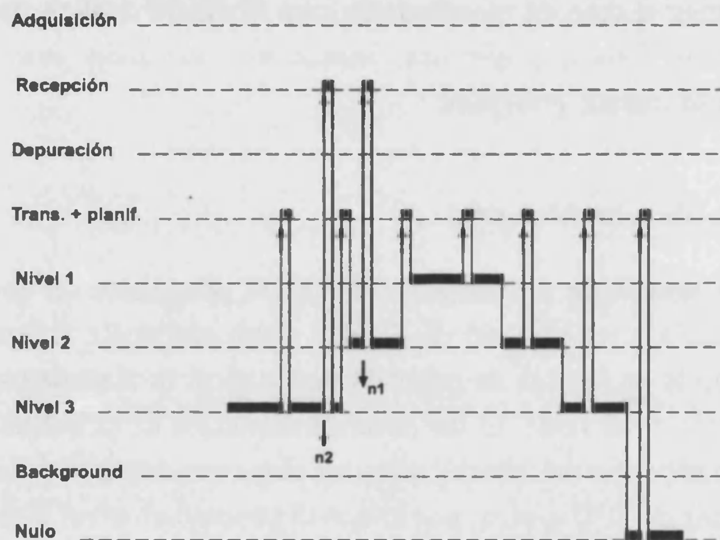


Figura 7.20: Traza de la ejecución del ejemplo de planificación.

Por otra parte, el tiempo de proceso de los servicios de recepción y del timer (transmisión y planificación) no son proporcionales al tiempo real en la figura 7.20. Nótese que el tiempo de proceso del servicio del timer es tan breve comparado con el tiempo entre disparos, que si se hubiera representado en tiempo proporcional aparecerían prácticamente en forma de rayas verticales. Para obtener más información de los tiempos de ejecución de los procesos críticos véase el apéndice I.

Cuando existe más de un mensaje en la cola de mensajes de un nivel de prioridad, todos ellos serán ejecutados según el orden de llegada por un mecanismo que responde a planificación FCFS. Así, la planificación expulsiva sólo se aplica a procesos con prioridades distintas, por lo que podríamos decir que el planificador asigna la CPU para la ejecución de todos los procesos, según el orden de llegada, de la cola de mensajes más prioritaria.

Por otra parte, nótese como, al terminar un proceso, éste restaurá automáticamente el de menor prioridad que le cedió la CPU. Este mecanismo evita la utilización de PCB ya que los procesos son breves y la planificación tradicional es bastante más costosa comparada con la mayoría de los procesos planteados aquí. Además, el uso de PCBs requeriría en ocasiones más tiempo para planificar algunos procesos que para

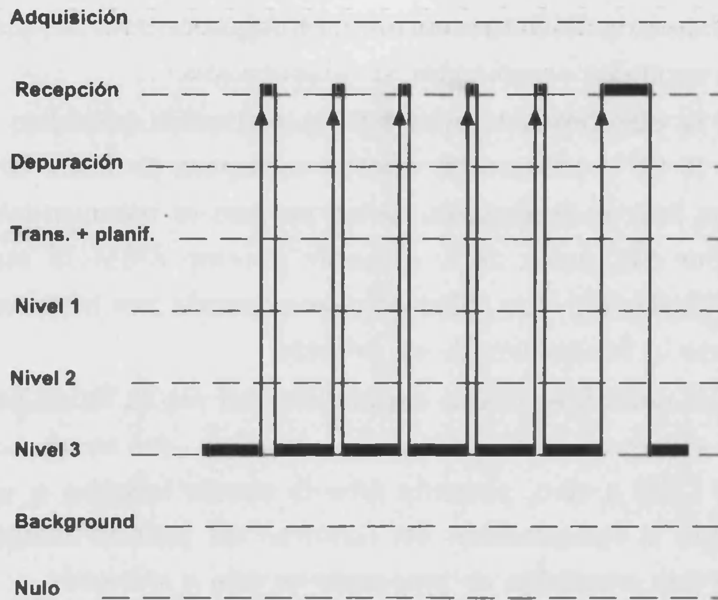


Figura 7.21: Traza de la recepción de un mensaje de 6 bytes.

ejecutarlos. Por esta razón prescindimos de las listas de procesos preparados para cambiar la estrategia de planificación.

7.5.4.3 La ISR del temporizador: Transmisión y planificación

En nuestro caso, la estrategia de planificación debe minimizar el tiempo de planificación y tratar de simplificarla, evitando recurrir a tipos complejos que requieran frecuentes invocaciones o consuman mucho tiempo de CPU. La planificación elegida para EMMOS se basa en la interrupción del *timer* para dotar a los procesos de una prioridad en la ejecución. Así, el *timer* se utiliza como temporizador para la invocación del planificador.

La frecuencia de invocación del planificador EMMOS puede ser gestionada con los servicios públicos del sistema operativo 'WriteTimer' y 'ReadTimer', con la que es posible leer y escribir el valor de carga al *timer*, usado para definir el tiempo entre invocaciones. Los prototipos de estas funciones son 'void WriteTimer(int prd)' y 'void ReadTimer(int *prd)', y actúan directamente sobre el registro PRD mapeado en memoria. Debe tenerse especial cuidado con el valor de este parámetro, ya que puede afectar significativamente a las prestaciones del sistema y la aplicación. El valor por defecto es $0x200$, por lo que el tiempo entre planificaciones es, por defecto, $512\mu s$. Si este valor fuera muy bajo se incrementaría el número de interrupciones y el servicio podría llegar a colapsar el sistema, acaparando la CPU, si por el contrario fuera muy alto pondría en peligro el multiproceso y las características de tiempo real

de la aplicación. Pese a todo, este parámetro se deja a criterio del programador para poder ser variado según las necesidades de la aplicación.

La política de la planificación se basa en la activación periódica del planificador para expulsar de la CPU procesos si existen en espera procesos con mayor prioridad. Sin embargo, tras la finalización de un proceso es recomendable una llamada al planificador. Por ello, antes de finalizar un proceso EMMOS asigna el valor de cuenta del timer, situándolo a un valor bajo, asegurando una intervención inmediata del planificador tras la finalización de un proceso.

Sin embargo, los episodios críticos del planificador son la forma en que éste almacena un contexto, generando la sutil transición de un proceso en *ejecución* a *preparado* en pila, asigna la CPU a otro, pasando éste de estado *inactivo* a *ejecución*, y permite posteriormente la recuperación del contexto del proceso relegado de la CPU, generando en éste una transición de *preparado* en pila a *ejecución*.

Almacenamiento de contexto

La invocación del planificador parte de una interrupción y, como tal, ésta deja guardado todo el contexto íntegramente en pila. El servicio de la interrupción del timer examina inicialmente si existen bytes por enviar a la placa de comunicaciones, si los hay llama a la rutina de transmisión. Haya o no haya que realizar este envío, inmediatamente después se mira si debe invocarse una planificación del proceso que posee actualmente la CPU. Para ello, el planificador inspecciona 'fintprior' para ver si existen flags activos correspondientes a colas de mensajes con una prioridad mayor que la del 'workinglevel' actual. De esta forma, el planificador conoce si hay esperando mensajes de mayor prioridad que el que ocupa la CPU.

Si el resultado del examen de 'fintprior' es negativo, la interrupción del *timer* finaliza devolviendo la CPU al proceso en ejecución. Sin embargo, si el resultado es positivo se invoca el planificador para relegar éste de la CPU. Antes de la planificación del proceso realiza una segunda comprobación: utiliza 'GetCaracterísticas' para obtener la dirección del proceso servidor de la orden que empaqueta el mensaje, si ésta devolviera una dirección nula, querría decir que la orden no puede ser servida bajo el modo de funcionamiento actual, o que simplemente no existe un proceso estático en memoria del servicio para esta orden. Sea cual sea la causa, si la dirección retornada es cero, el mensaje se desapila, y se finaliza el servicio de interrupción devolviendo el control al proceso en ejecución sin haberlo planificado.

Si, por el contrario, figura 7.22, se encuentra la dirección del servicio de la orden recibida, se apila el valor actual de la variable 'workinglevel' (0:background, 1:nivel 3, 2:nivel 2, 3: nivel 1), que pasará a formar parte del contexto, y que nos dice

la prioridad del proceso que ocupaba la CPU en el momento de la activación de la interrupción (el proceso que pasa ahora a estado preparado)[3].

Asignación de la CPU a otro proceso

Para asignar la CPU al proceso de mayor prioridad, la variable 'workinglevel' del proceso relegado se apila y se asigna a ésta el valor de la nueva prioridad. De esta forma, se establece un mecanismo para definir que prioridades pueden planificar el proceso entrante, sólo aquellos procesos con prioridad mayor que el 'workinglevel' actual podrán relegar a éste de la CPU. Nótese que el proceso de asignación de la CPU a otro proceso de mayor prioridad, equivale a realizar la transición de éste de estado *inactivo* a *ejecución.*

El proceso de servicio de la orden que se debe atender, se realiza mediante una llamada de tipo C. Antes de la llamada se apilan todos los datos empaquetados por el mensaje más el número de línea según un prototipo de función:

```
void function(linea, datol, dato2, dato3, dato4,...)
```

Este prototipo de función debe corresponderse exactamente con el proceso de servicio programado, enumerando en estricto orden como argumentos los datos enviados a través del mensaje. Tras apilar los argumentos se habilitan las interrupciones para posibilitar el multiproceso durante la ejecución del proceso de servicio. Entonces se realiza la llamada del proceso. Los argumentos ya se encuentran situados en pila y el proceso de servicio debe hacer uso de ellos según el prototipo de función indicado.

El servicio de interrupción del *timer* se escribe en ensamblador en aquellas partes, como la llamada del proceso y la restauración, para obtener mayor rapidez y flexibilidad que la que el C proporciona. Además, se ejecuta con las interrupciones deshabilitadas hasta que se realiza la llamada para asignar la CPU al proceso de mayor prioridad, tras el apilamiento de los argumentos de éste.

Restauración del contexto

Antes de realizar la llamada al proceso de servicio se activan las interrupciones y se establece como 'workinglevel' el del nivel de prioridad que se va a servir. Cuando finaliza el proceso de servicio y éste retorna de su realización:

a. Se deshabilitan las interrupciones para que el cambio de proceso se realice por completo antes de ser habilitadas.

[3] El valor 'workinglevel'=0 se aplica indistintamente tanto al proceso nulo como a cualquier proceso en background.

b. Se desapilan las variables apiladas, argumentos del proceso de servicio.

c. Se restaura el *'workinglevel'* anterior.

d. Se restaura el estado de la pila al estado anterior a la entrada en el planificador.

e. Se restaura el contexto de la interrupción.

Cuando finalmente se restaura el contexto de la interrupción, se restaura el contexto del proceso que corría antes de que el de mayor prioridad le arrebatara la CPU. Nótese que el mecanismo de restauración indicado equivale a una transición de proceso *preparado* en pila a *ejecución*, una vez finalizado el proceso de mayor prioridad.

Cuando la CPU no tiene procesos que servir, ejecuta el proceso nulo, que inspecciona y sirve continuamente la cola de procesos en background, de manera similar a una planificación a largo plazo con una cola de procesos por lotes. Caso de no existir ningún proceso que servirse en background realiza un bucle de espera que ocupará la CPU hasta que exista algún proceso que servir. Cuando la CPU ocupa el proceso nulo o un proceso de background el valor de 'workinglevel' es 0, el mismo valor al que esta variable se inicializa por defecto.

7.5.5 Otras consideraciones

Cada vez que se activa el timer se busca una posible planificación. Si procede la asignación de la CPU a otro proceso de mayor prioridad, se realiza una llamada a éste, relegando de la CPU al proceso de menor prioridad. El planificador se ejecuta con las interrupciones deshabilitadas, pero se habilitan en el momento de la llamada al proceso de servicio de mayor prioridad y vuelven a deshabilitarse cuando finaliza, antes de la restauración del contexto. Nótese que esta planificación simula el hecho de que la aparición de un mensaje en una cola de prioridad más alta interrumpa por software, y con derecho preferente, al de una prioridad más baja, que está en ejecución.

El servicio de interrupción que planificó un proceso finalizará cuando el proceso finalice, y la restauración de contexto restaurará el proceso software de menor prioridad que se ejecutaba anteriormente.

La planificación EMMOS de un proceso puede verse también como interrupciones anidadas donde sólo pueden interrumpir los procesos de mayor prioridad. Siempre que se ejecuta un proceso con niveles de prioridad 1 a 3, se está ejecutando sobre un servicio de interrupción del timer anidado, es decir, no se originan diferentes interrupciones. La variable 'workinglevel' es fundamental, proporcionando una cota de los niveles de

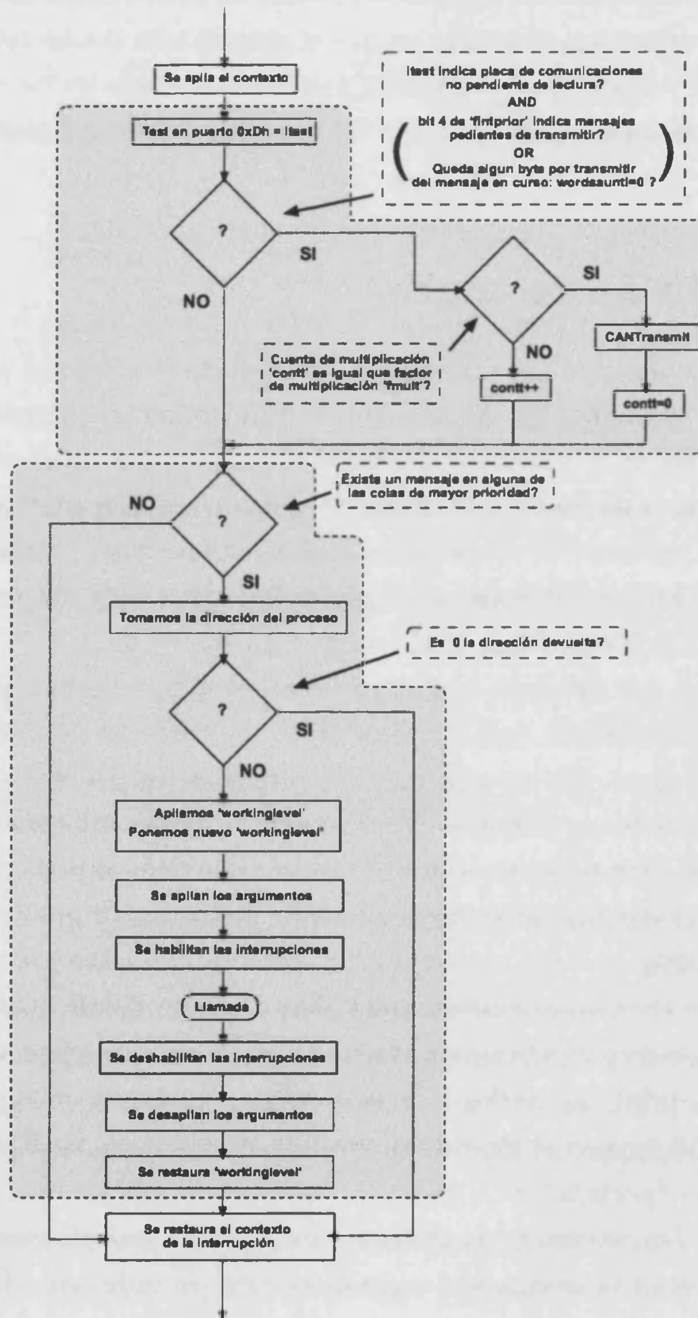


Figura 7.22: Diagrama de flujo del servicio del *timer*.

prioridad que pueden interrumpir, por lo que describe la prioridad del proceso en ejecución en los diagramas 'prioridad - tiempo proceso'.

Los diagramas 'prioridad - tiempo proceso' indican el cambio de prioridad desde el momento en que se realiza la llamada al proceso de nueva asignación hasta que éste cesa, omitiendo el tiempo de ejecución en que el planificador ocupa la CPU. Esto es así porque, dada la brevedad del servicio, éste puede ser obviado en las representaciones: El proceso de atención de la ISR del timer sin necesidad de planificación o transmisión dura $18.4\mu s$.

7.6 Gestión de memoria

En un sistema operativo de escritorio, la gestión de la memoria principal consiste principalmente en la asignación de la memoria física a los requerimientos de los procesos. Las demandas de la memoria provienen de la creación de nuevos procesos y de la expansión dinámica de los ya existentes. Normalmente la memoria se libera tras la terminación de un proceso. Por otra parte, el sistema operativo puede elegir retirar de memoria principal los procesos temporalmente inactivos o en suspensión para dejar espacio que necesitan los procesos preparados y en ejecución.

Sin embargo, en los sistemas en tiempo real, el gestor de memoria está comparativamente menos solicitado que en otros tipos de sistemas de multiprogramación. La razón principal para ello es que muchos procesos residen permanentemente en memoria para proporcionar tiempos de respuesta rápidos. Al contrario que tiempo compartido, la población de los procesos en sistemas de tiempo real está prácticamente estática y hay relativamente poco movimiento de programas entre el almacenamiento primario y secundario.

EMMOS, es un sistema operativo embebido de tiempo real, que debe ejecutarse en un entorno reducido y relativamente cerrado, sin almacenamiento secundario; esto redunda en que la totalidad de los procesos residan estáticos en memoria principal. Por ello puede considerarse el módulo de gestión de memoria en EMMOS como una parte "atrofiada" o "podada".

Generalmente, los procesos en sistemas de tiempo real tienden a cooperar estrechamente, soportando separación y compartición de memoria. Para ello se usan normalmente procesadores de carácter general que admiten paginación y separación de espacios de direcciones e implementan vía hardware utilidades para que el chequeo de la protección se pueda realizar sin penalización del rendimiento (Intel, 1989). Estos mecanismos genéricos gestionan la separación y compartición de memoria en procesadores genéricos, pero no existen en los procesadores digitales de señal. Estos últimos poseen alta velocidad en la realización de primitivas típicas del procesado digital de

señal, pero no soportan la protección de páginas y segmentos. Así, un proceso puede acceder a todo el espacio de direccionamiento de datos. Sin embargo, esto no es problemático en la práctica, porque en EMMOS los procesos sólo usarán aquellos datos, estructuras y variables, del sistema operativo que le sean declarados mediante las correspondientes cabeceras (datos públicos del SO, apéndice E), las variables locales (variables temporales alojadas en pila) y las variables globales de la aplicación (alojadas en la sección .bss de memoria, apéndices D y E).

7.7 Sincronización

Dada la naturaleza de EMMOS, que debe procesar las órdenes enviadas desde el control siendo cada proceso independiente, se han eliminado mecanismos genéricos de sincronización entre procesos como son: semáforos, monitores, *mutexes*, etc. Sin embargo, EMMOS conserva un simple y eficiente mecanismo de sincronización que evita interferencias no recuperables entre procesos: las *zonas críticas.*

Una parte de las llamadas del sistema utiliza zonas críticas para asegurar que un determinado segmento de código pueda ser ejecutado íntegramente y evitar conflictos no recuperables, como es el caso, por ejemplo, del desapilamiento o apilamiento de mensajes. Supóngase que ocurre una planificación a mitad de un desapilamiento, si el nuevo proceso pretendiera desapilar un mensaje, se encontraría con un desapilamiento en marcha que corrompe el estado de la cola, por lo que siempre debe asegurarse que si uno de los procesos críticos que realiza el sistema operativo comienza, también debe finalizar antes de cualquier transferencia de control a otro proceso distinto del que está en marcha.

Un segmento de código se convierte en una sección crítica cuando se protege con una habilitación y deshabilitación de interrupciones antes y después de dicha sección. Así, podemos evitar cualquier tipo de intervención por parte de un proceso de mayor prioridad. Pero EMMOS prescinde del uso de DINT/EINT, ya que podría comprometer la estabilidad del sistema alterando el estado de las interrupciones anterior a la zona crítica. La deshabilitación de las interrupciones se basa en el enmascaramiento de los flags de la máscara de interrupción. Para ello, previamente se almacena la máscara de interrupciones y se enmascaran todos los flags, estableciendo una zona crítica hasta la restauración. Aunque esto no es necesario en aquellos casos en que se realice una llamada del sistema con las interrupciones deshabilitadas, como en el caso de los procesos de recepción o transmisión, la deshabilitación mediante máscara evita la utilización de DINT/EINT, que habilitaría las interrupciones tras la zona crítica sea cual sea el estado anterior.

Debe tenerse en cuenta que una componente fundamental de sincronización de la

aplicación la introduce el control, que mediante el envío de mensajes no sólo transmite información sino *eventos de sincronización* como son los mensajes de sincronismo, que señalan la finalización del paso de una taza por el patín de la célula de pesada, o los de señalización de cambio de velocidad de la cinta.

7.8 Resumen

Con un tamaño de tan sólo 7KW, EMMOS es un microkernel embebido, diseñado específicamente para sistemas reducidos con una única vía de comunicación con el exterior, los mensajes CAN. Este núcleo es grabado en memoria EPROM para evitar la necesidad de almacenamiento secundario. EMMOS realiza una ***planificación expulsiva con derecho preferente basada en prioridades con ejecución FCFS de los procesos de la misma prioridad.*** Utiliza cuatro prioridades software. Posee una latencia de interrupción típica de 120μs y un diseño que lo habilita para ser usado como sistema de estricto tiempo real.

EMMOS posibilita que cualquier aplicación pueda hacer uso de los mecanismos hardware mediante una capa de software, las llamadas del sistema y unas estructuras públicas. La utilización del *microkernel* facilita el desarrollo y depuración de la aplicación, sobre todo cuando las tareas DSP se incrementan, empleándose para asignar los recursos del sistema y permitir multitarea (Castelli, 1995). Las aplicaciones bajo EMMOS son muy sencillas de programar, sección 7.4.

La compilación de la aplicación es una compilación cruzada por realizarse sobre otra plataforma, que puede ser portada de dos formas diferentes al sistema final, mediante cable serie cuando se desee realizar labores de depuración, o mediante EPROM en funcionamiento normal. En cualquier caso se requiere un enlazado de la aplicación con el SO para formar el ejecutable que será portado.

Capítulo 8

La Aplicación

8.1 Introducción

El módulo de pesado es, dentro de la calibradora, uno de los nodos inteligentes del control distribuido. La aplicación es el software que le proporciona dicha "inteligencia". Ésta se basa en el SO, aprovechando los servicios del sistema y la gestión del hardware que éste proporciona, para realizar una función principal: La obtención de los pesos de las piezas que son transportadas sobre la célula de pesada a grandes velocidades.

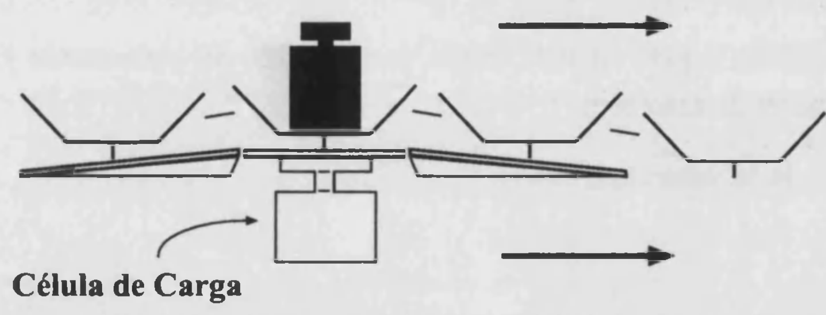


Figura 8.1: Esquema del paso de una taza sobre la célula de carga.

La figura 8.1 ilustra el proceso por el cual las tazas, llenas o vacías, pasan sobre la célula de carga. Las tazas son piezas móviles que, unidas a la cadena de arrastre, poseen un par de grados de libertad de movimiento para evitar influencias externas en el resultado del proceso de pesado. En la figura 8.1 vemos cómo la cadena de arrastre se mueve, con una velocidad constante, transportando las piezas a calibrar en sus tazas. En una zona, éstas llegan a un patín de pesado, que eleva las tazas dejando

descansar todo su peso sobre el patín mientras que la cadena de arrastre sólo interviene dotando a ésta de componente horizontal de movimiento, que no influenciará a la magnitud transducida.

El patín eleva gradual y suavemente la taza llegando a una zona plana que posee, antes de su final, un par de discontinuidades. Esto es, un segmento independiente de la zona plana del patín que se encuentra unido al vástago de la célula de carga, permitiendo obtener una señal, al paso de la taza sobre el patín, que procesada proporcionará el valor escalar del peso de la fruta que la taza contenía.

Tras el paso de la taza sobre la célula de carga, el patín vuelve a bajar la taza, que volverá a ser soportada por la cadena de arrastre camino de la etapa de salida del sistema.

La aplicación realiza el procesado de la señal obtenida de la célula de carga durante el paso de la taza. Esta señal es adquirida con una *frecuencia de muestreo de 300Hz y una resolución de 16 bits.* La corriente de datos adquirida se utiliza para obtener el valor del peso (Calpe, 1996). Sin embargo existen ciertas complejidades como:

- La señal adquirida, con el peso sobre la célula de carga, no es constante, se comporta como un sistema de segundo orden, en el que la taza está tan poco tiempo sobre la zona de pesado, que no llega al tiempo de asentamiento de la señal.
- El peso obtenido depende también del peso de la taza en concreto que lo soporta.
- El peso de la taza puede cambiar durante el funcionamiento.
- La adquisición capta, además, ruido, componente del rozamiento y resonancias mecánicas de la máquina.
- Depende de la velocidad.
- ...

A pesar de todo, la máquina debe dar un valor muy preciso del peso que transporta la taza, y para ello es necesario: una buena calibración; ajuste y programación de los conversores; tarado de las tazas vacías y control dinámico de su peso, etc. Para todo ello, la aplicación juega un papel fundamental.

La aplicación es la encargada de realizar todas las tareas relativas al procesado de la señal, teniendo en cuenta varias variables como se ha descrito anteriormente, para obtener los resultados en tiempo real, y comunicarlos al control. No obstante, el DSP no sólo realiza tareas de procesado, sino también de control, a nivel de definición de modos de funcionamiento, tareas de ajuste y gestión de los conversores, configuración,

etc. por ello las exigencias que el control requiere del módulo de pesado justifica la utilización de un sistema operativo de estricto tiempo real para soportar la aplicación.

8.2 Estructura

Para estructurar el funcionamiento de la aplicación se definen los denominados *modos de funcionamiento*, esquematizados en la figura 8.2. En ésta pueden verse los modos de funcionamiento del sistema estructurados en forma de árbol, donde para pasar de uno a otro se ha de navegar por el árbol de la misma manera que ocurre en la navegación por una árbol de directorios.

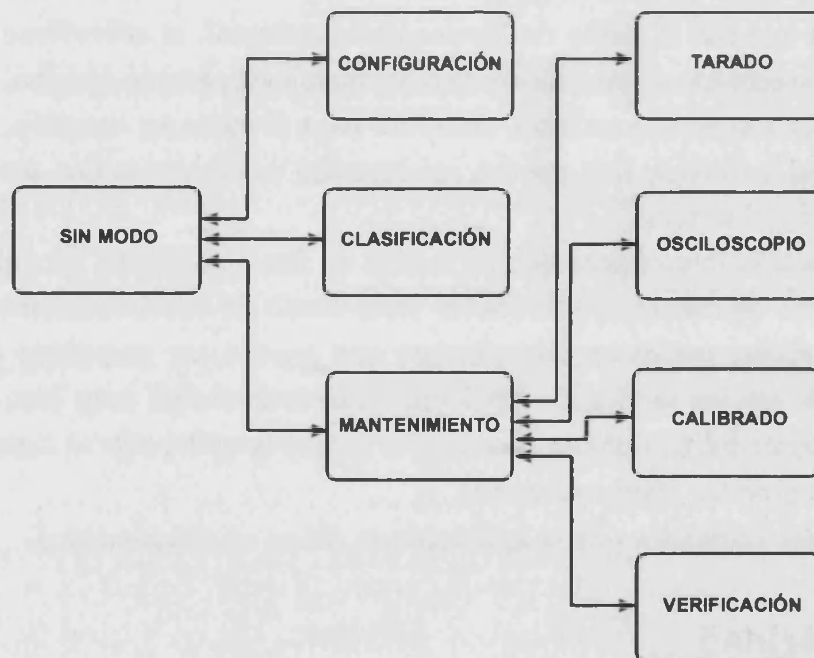


Figura 8.2: Árbol de modos de funcionamiento.

Pero, ¿por qué hay modos? Si miramos la arquitectura del software de servicio de las órdenes, veremos que hay ciertas órdenes que se atienden sea cual sea el modo actual de la pesadora (órdenes asíncronas). Sin embargo, la mayoría de las órdenes son síncronas, es decir, sólo se atienden dentro de su modo característico. Si la pesadora no se encuentra en dicho modo se ignoran totalmente los servicios a dichas órdenes.

El modo 0 es un modo de funcionamiento en que la pesadora está "en punto muerto". También existe un modo denominado 'Sin Modo' que es un modo en el que la pesadora está en punto muerto y en el que sólo aceptará órdenes asíncronas

y pasos a cualquiera de los siguientes tres modos: Clasificación, Configuración, o Mantenimiento.

El control es el que cambia los modos de funcionamiento del sistema, determinando así el modo de funcionamiento del módulo para que eéte actúe acorde con el funcionamiento global de la máquina.

Como se recordará del estudio de la tabla de edición de procesos (capítulo 7), cada modo dispone de una serie de mensajes que pueden ser atendidos en ese modo de funcionamiento, ignorando los mensajes que se definen en otros modos de funcionamiento. Por ejemplo, en configuración la tarjeta no atiende mensajes propios de mantenimiento, etc. Esto se hace para evitar conflictos de funcionamiento y evitar ordenes que pudieran resultar conflictivas, evitar servicios de mensajes atrasados, etc. Como ya sabemos, las órdenes que cada modo de funcionamiento puede atender se presentan en la tabla de edición de procesos .

Cualquiera que sea el modo de funcionamiento actual, si sobreviene un mensaje definido para su servicio en otro modo de funcionamiento, éste se desecha. Esto ocurre con todos los mensajes que no estén definidos para el modo en cuestión, a excepción de los mensajes *amodales*, que pueden ser servidos cualesquiera que sea el modo de funcionamiento del sistema.

Como vemos, la implementación de modos de funcionamiento permite establecer una distribución de mensajes válidos en cada modo de funcionamiento, siendo los mensajes amodales, mensajes atemporales, que pueden ser atendidos en cualquier momento. Además los modos de funcionamiento estructuran muy bien los estadios de funcionamiento del módulo de pesado, por lo que la aplicación se describirá según una clasificación de los distintos modos.

Los mensajes utilizados por la aplicación se listan en el apéndice J.

8.2.1 Prioridad

EMMOS proporciona a la aplicación facilidades multitarea, basándose en las prioridades de los procesos de servicio. Se puede establecer (véase apéndice H) una jerarquía de prioridades entre los procesos de la aplicación para realizar una rendimiento eficiente de la CPU. Algunos ejemplos de ello son:

- El proceso de ajuste de los conversores es muy largo y posee prioridad de nivel 2. El final del modo de ajuste posee prioridad de nivel 1 para habilitar los mecanismos que produzcan el aborto del ajuste ante esperas prolongadas.
- El procesado que desencadena el mensaje de llegada de un sincronismo tiene prioridad de nivel 1, frente al nivel 2 del mensaje de finalización del modo

para asegurar que la finalización del modo se realiza cuando se han enviado los últimos pesos calculados.

- Existe en la aplicación un servicio en *background* que revisa las tazas cuando el retarado dinámico de alguna baja dé un umbral, que podría significar que la taza está rota o que simplemente ha sido arrancada de la cadena de arrastre. Este proceso no es en absoluto prioritario, por ello se realiza en *background*.

- El proceso de servicio de la solicitud de taras en el modo de clasificación posibilita que en este modo el sistema pueda enviar las taras, o pesos de las tazas, con prioridad de nivel 3, aprovechando los tiempos de CPU no usados por procesos prioritarios de este modo.

- ...

Además, el multiproceso y el sistema operativo facilitan tanto la adaptación a cualquier exigencia futura, como la realización de nuevas versiones, proceso que sería más complicado en las programaciones monolíticas (?). Por ejemplo, cuando se implementen en tiempo real de manera industrial los algoritmos obtenidos en el capítulo siguiente.

8.3 Modos de funcionamiento

8.3.1 Modo configuración

Modo de funcionamiento al que se accede desde el modo 0 (sin modo). En éste se configuran los parámetros fundamentales de la pesadora: número de tazas, número de lineas y distancia al origen por defecto. Este modo corresponde al modo 1. Las siguientes órdenes sólo corresponden a este modo de funcionamiento:

CONFIGURACIÓN

Esta orden sólo se atiende en modo 0 (sin modo). Simplemente cambia el modo de funcionamiento a CONFIGURACION (1).

Rx	CONFIGURACION **Orden: 101**	0101 1111	1111111 1111111	00000

LINEAS_SISTEMA

Este mensaje envía el número de líneas del sistema (dato 1) tanto en NVRam (y se valida su flag testigo), como en la copia local. Este valor se corresponde con el número de líneas que compone la calibradora. El número de líneas es de 1 a 10, de lo contrario se lanza un error.

Rx	LINEAS_SISTEMA **Orden: 112**	0101 1111	1111111 1111111	00000	Número de líneas

TAZAS_SISTEMA

Este mensaje envía el número de tazas que compone cada línea de la calibradora (datos 1 y 2) tanto en NVRAM (y se valida su flag testigo), como en la copia local. Todas las líneas deben tener el mismo número de tazas. El número de tazas ha de ser de 1 a 1400, de lo contrario se lanza un error.

Rx	TAZAS_SISTEMA **Orden: 110**	0101 1111	1111111 1111111	00000	Número de tazas por línea (hasta 1400)

_DISTANCIA_ORIGEN_PESADO

Este mensaje guarda la distancia al origen de la pesadora (datos 1 y 2) tanto en NVRAM (y se valida su flag testigo), como en la copia local. Esto es, el encoder envía un sincronismo cada vez que un punto concreto de una taza pasa por una determinada posición. Con ese mensaje también se envía el índice de la taza, pero de la taza que pasa por delante del encoder. Pero claro, por la célula de pesada estará pasando una taza de índice anterior, por lo que hay que sumarle un offset. Este offset que hay que sumarle a cada uno de los índices que trae consigo el mensaje de sincronismo es la distancia al origen de la pesadora.

Este valor se resta a cada índice entrante en el mensaje de sincronismo, para obtener el índice correcto de la taza que pasa sobre la célula de pesada.

Rx	DISTANCIA_ORIGEN _PESADO **Orden: 140**	0101 1111	1111111 1111111	00000	Distancia en tazas al punto de origen

FIN_CONFIGURACIÓN

La orden FIN_CONFIGURACIÓN sólo se atiende estando en modo 1 (CONFIGURACIÓN). Se encarga de cambiar el modo actual al modo a 0 (Sin Modo).

Rx	FIN_ CONFIGURACION **Orden: 199**	0101 1111	0000000 1111111	00000

8.3.2 Tarado

El tarado es el procedimiento por el cual serán pesadas en vacío las tazas de la calibradora. Las taras, o pesos en vacío de las tazas, se guardan en NVRAM. La NVRAMm tiene una capacidad de 32KB, por lo que los valores de las taras han de almacenarse en los LSB bytes de los campos de una estructura que diferencia parte alta y parte baja:

```
struct Bibyte
{
   int MSB;
   int LSB;
};
```

Como hay hasta 1400 taras posibles por linea, con un máximo de 10 lineas, el espacio de NVRAM reservado para las taras es 28KB. Nótese que esto significa la práctica totalidad de la NVRam para este cometido.

Sin embargo, como los valores del peso de las tazas, que por supuesto se almacena en puntos del conversor, es siempre muy bajo, no almacena una palabra de 16 bits sino sólamente 12 bits. La figura 8.3 ilustra la topología de los 16 bits de cada tara.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ntaras		erbits		Valor tara despazada 2 bits a la derecha (tara/4)											

Figura 8.3: Topología de los 16 bits de cada tara.

Inicialmente, cuando se inicia la tara, 'erbits' (*bits de error*) y 'ntaras'(*número de taras*) son puestos a cero, mientras que 'valor'=0xFFF. Cuando, durante la primera vuelta de la cadena de arrastre, comienza a obtenerse el valor del peso de las tazas (nótese que para realizar el proceso de tarado es necesario que todas las tazas del

sistema estén vacías) la primera vuelta sencillamente toma el valor de la taza en vacío en puntos del conversor y lo introduce íntegramente en el campo 'valor'. Esta primera vez que se tara la taza se pone 'nvueltas' a 1 y 'erbits' continúa a cero para todas las tazas que se taren por primera vez.

Cuando una taza se tara por segunda vez, o un mayor numero de veces, el procedimiento es idéntico. A partir de la segunda pesada, el número de taras ('ntaras') se incrementa. Posteriormente se halla el nuevo valor de la tara, que tendrá en cuenta el valor anterior para que sucesivas vueltas hagan converger la sucesión al resultado. Para ello se utiliza un coeficiente de convergencia en tarado que indica cómo convergerá la sucesión hacia el resultado final. La expresión por la cual se obtiene el siguiente valor de la tara es:

$$valor = viejovalor + (nuevovalor - viejovalor) * coeftara$$

donde 'viejovalor' es el valor que contenía la tara, resultado de la vuelta anterior, 'coeftara' es el coeficiente de convergencia (de 0 a 1) que indica cómo se tiene en cuenta el nuevo valor para el nuevo cálculo de la tara, y finalmente 'nuevovalor' es el valor leído (en puntos del conversor) para el peso de la taza en esta última vuelta. Así, si 'coeftara'=0.5 simplemente estamos promediando el valor de la tara, de otra forma haremos un promedio pesado para encontrar el valor de la nueva tara. De esta forma puede indicarse la forma en que la tara convergerá a su valor definitivo.

Cada vez que se vuelve a tarar la taza, el campo número de taras se incrementa. Pero como éste sólo consta de dos bits, únicamente puede indicar hasta 3 vueltas. A partir de la cuarta se satura a 3 indicando que la taza se ha tarado 3 o más veces, ya no volviendo a modificarse más este valor por más vueltas que se le dé al tarado.

Por otra parte, los dos bits del campo bis de error indican las diferencias encontradas entre diferentes tarados. Sabemos ya que durante la primera vuelta 'erbits' queda siempre a 0, puesto que en ésta no existe un valor anterior con el que establecer una comparación. Los bits de error indican que en vueltas posteriores han existido variaciones sospechosas en el valor obtenido del tarado de la taza. A partir de la segunda vuelta, y siendo 'dif' el módulo de la diferencia entre el 'antiguovalor' de la tara y el 'valor' pesado ahora :

- Si $dif > viejovalor/4$ a *Erroneo*, y 'erbits' se pone a 11*b*
- Si $dif > viejovalor/8$ a *Dudoso* y 'erbits' se pone a 01*b*
- Si $dif < viejovalor/8$ a *OK*, y 'erbits se pone a 00*b*

Fijémonos que de esta forma pueden verse las dispersiones que han sufrido los valores pesados de la taza vacía en unidades célula y utilizar este campo para observar

la fiabilidad del valor hallado durante el proceso de tarado para cada una de las tazas, identificar tazas rotas, etc. Nótese también que si en el tarado de una taza se produce *dudoso* ('erbits' a 01b), aunque en posteriores vueltas vaya todo bien, los bits de error siempre se mantendrán a nivel de dudoso y sólo se modificarán caso de que se produzca una diferencia que los ponga a 11b (Error). En este último caso permanecerán a partir de ese momento siempre a 11b. En definitiva, si en cualquier vuelta ocurre un error, aunque en la siguiente se realice correctamente siempre quedará en 'erbits' el mayor nivel de error obtenido, por lo que en 'erbits' siempre tendremos el error máximo que ha ocurrido durante el tarado de cada taza.

El proceso de tarado se realiza cada vez que le llega al sistema un sincronismo, indicando que la taza está en la posición adecuada para pesar, e indirectamente, el número de taza que está sobre la célula de pesada. Más adelante se estudiarán con más detalle los sincronismos.

Nótese como, de esta forma, el número de vueltas durante el procedimiento de tarado no tiene porqué ser fijo. A mayor número de vueltas mejora el procedimiento de tarado de las tazas del sistema.

ORDEN_TARAR

ORDEN_TARAR sólo se atiende estando en modo MANTENIMIENTO y se encarga de poner el modo actual a 4 (TARADO) y realizar las inicializaciones correspondientes.

Rx	ORDEN_TARAR **Orden: 1300**	0101 1111	0000000	00000

El tarado comienza inicializando las estructuras de todas las tazas del sistema, asignando a todas un mismo valor 0x0FFF. Esto es, el campo bits de error a 0, el campo número de vueltas a 0, el campo valor de la tara a 0xFFF. Posteriormente saca del *standby* a los conversores, y comprueba el número de líneas útiles del sistema (aquéllas en las que el conversor responde adecuadamente y existe conectada una célula de carga), si hay al menos una línea útil programa los conversores y activa la interrupción de conversión para que comience la adquisición, y con ello el tarado. Si no hay ninguna línea útil que tarar, envía un mensaje de error, pone los conversores de nuevo en *standby* y sale del modo. En este último caso, es el mensaje de error el que notifica al control que se ha abandonado el tarado por falta de líneas útiles.

ORDEN_SOLICITUD_TARAS

El proceso de servicio de este mensaje envía las taras de todas las tazas de todas las líneas del sistema. Sin embargo, ***antes de llamar al proceso de servicio que invoca este mensaje, se ha de tener la precaución de haber realizado un tarado.*** De lo contrario no tendrá sentido la información enviada.

Es un mensaje asíncrono especial, ya que puede atenderse en cualquier modo de funcionamiento, a excepción del de tarado, pues no tiene sentido pedir las taras mientras se están obteniendo.

Rx	ORDEN_SOLICITUD _TARAS **Orden: 1306**	0101 1111	0000000	00000

Como se verá en breve, las taras se envían al control de tres en tres, encapsuladas en el mensaje de salida DATOS_TARAS. Sin embargo, nótese que el número de taras, y por lo tanto de mensajes DATOS_TARAS a enviar, es muy alto (caso de 10 líneas y 1400 tazas, se enviarían más de 4600 mensajes), lo cual desbordaría rápidamente la cola de mensajes de transmisión, que apenas puede contener más de una veintena. Por otra parte, si se está realizando a la vez otra actividad que necesite de la transmisión de mensajes, ésta tiene prioridad sobre el proceso de servicio del envío de las tazas, que no deberá colapsar la cola de mensajes de transmisión para evitar interferencias con aquél.

Para cumplir estos requisitos, el proceso de servicio se ejecuta en nivel 3. Si éste encuentra que el modo actual es CLASIFICACIÓN, VERIFICACIÓN u osciloscopio, pues TARADO le está vetado, realiza un bucle de temporización entre mensajes enviados para asegurar que, trabajando a plena carga, el servicio de transmisión tenga tiempo de enviar los mensajes propios del modo actual y los datos de las taras sin colapsar la cola de transmisión. Si el modo de funcionamiento es otro, el proceso envía mensajes hasta que llena la cola de transmisión, a partir de ese momento realiza una temporización y lo intenta un poco más tarde. Se procede de esta forma hasta haber enviado todas las taras. La ventaja de la ejecución en nivel 3 es que este proceso no interfiere en la ejecución habitual del modo de funcionamiento vigente.

Para indicar la finalización del envío de las taras se envía un mensaje DATOS_TARAS con los siguientes parámetros: $0xFF$ | $0xFF$ | $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x00$. Denotando con el índice de taza $0xFFFF$, que se han enviado ya todas las taras.

DATOS_TARAS

Este mensaje indica la línea (de 1 a 10) y el índice de la taza a la que corresponde la primera tara que se envía. En cada mensaje se envían tres taras, todas corresponden a la misma línea y tienen índices de taza consecutivos (a partir del indice indicado que es el de la primera tara). Si el mensaje procesado es el último, puede que la última o las dos últimas taras que se envían no correspondan a tazas existentes en la cinta, esto se marca con ceros en su valor. Cada una de las taras lleva los llamados 'bits de fiabilidad' (los ya mencionados 'erbits' y 'nvueltas') y el valor de la tara ('valor'), que ya se describieron anteriormente.

Tx	DATOS_TARAS Orden: 1308	Índice línea	Índice taza tarada	Fiabilidad + TARA 1		Fiabilidad + TARA 2		Fiabilidad + TARA 3		Fiabilidad + TARA 4	
				LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

Grado de fiabilidad:

- **ntaras** (bit 15-bit14) Número de veces que se ha tarado la taza en cuestión.
- **erbit** (bit13-bit12) Error detectado tarando la taza.

Tara : (Bits 11 a 0) Son el los bits 1 a 12 del valor en puntos de la tara.

ORDEN_INTERRUMPIR_TARADO

El procedimiento de tarado puede ser parado en cualquier momento mediante esta orden. Para ello, se cambia el modo actual a MANTENIMIENTO, se desactiva la interrupción de conversión y se ponen en *standby* los conversores.

Rx	ORDEN_INTERRUMPIR_TARADO Orden: 1305	0101 1111	0000000	00000

8.3.3 Calibración

El propósito del modo CALIBRACIÓN es la cooperación con el control para la obtención de una serie de cuádricas de interpolación de puntos a gramos, que además pueden ser diferentes para cada línea y cada velocidad.

La calibración se puede realizar a varias velocidades. La NVRAM de la tarjeta de pesado almacena los tres coeficientes ($y = l \cdot x^2 + m \cdot x + n$) que han de ser usados

para la transformación de puntos a gramos a distintas velocidades: 0 (en parado), 1, 2, ..., 20 (frutas/segundo). En total son 21 las ternas de coeficientes almacenadas para cada línea. Estas ternas son almacenadas en NVRAM (utilizando sólo los bytes LSB) a través de una estructura:

```
struct Bibyte
{
    int MSB;
    int LSB;
};
struct Bibyte NvmCal[21][10],NVnCal[21][10],NVlCal[21][10];
```

En modo CALIBRACIÓN, siempre que llega un sincronismo en el que se detecta que la taza pasa llena se envía al control un mensaje con el valor digital de la pesada (en puntos del conversor). Así, el control recibe el valor digital del peso de la pieza que ocupa la taza (que en este caso generalmente será un peso patrón) para que éste pueda ser tenido en cuenta durante la obtención de los coeficientes. Es el control el que mediante mensajes a través del interfaz, indica los pasos a realizar durante el proceso de calibración.

Una vez finalizado el procedimiento de calibración y obtenidos los coeficientes, éstos le serán enviados al módulo de pesado a través del mensaje ENVIO_PARAMETRO. Posteriormente se puede verificar su actualización en la pesadora mediante los mensajes PETICIÓN_PARAMETRO y ENTREGA_PARAMETRO.

En otros modos, la aplicación utilizará estos coeficientes para obtener el peso en gramos a partir del peso en unidades del conversor. En esta interpolación se utilizará la terna de coeficientes indicada por la velocidad actual de la cadena de arrastre, siendo distintas las ternas de coeficientes que se utilizan para cada línea.

El modo CALIBRACIÓN consta de los siguientes mensajes:

ORDEN_CALIBRAR_LINEA

Esta orden sólo se atiende en modo MANTENIMIENTO, de lo contrario el mensaje será desechado. En los datos se pasa un parámetro que consiste en una máscara de bits que especifica que líneas se pretenden calibrar (bit 0 $\longrightarrow$ linea 1, ..., bit 9 $\longrightarrow$ linea 10).

<table>
<tr><td rowspan="2">Rx</td><td rowspan="2">ORDEN_CALIBRAR_LINEA
Orden: 1320</td><td rowspan="2">0101
1111</td><td rowspan="2">0000000</td><td rowspan="2">00000</td><td colspan="2">Máscara de líneas a calibrar. LSBit=Línea 1</td></tr>
<tr><td>LSB</td><td>MSB</td></tr>
</table>

Esta orden pone el modo actual a 6 (CALIBRACIÓN), toma las líneas útiles que hay (aquéllas en las que el conversor funciona correctamente y hay conectada una célula de carga) y procede a calibrar aquellas líneas útiles cuya calibración se ha solicitado. Para ello realiza las inicializaciones oportunas, saca a los conversores del *standby* y desenmascara la interrupción. Si, por el contrario, ninguna de las líneas que se pide calibrar son útiles, envía un mensaje de error, mantiene los conversores en *standby* y la interrupción enmascarada, y sale (sin salir de modo 6) sin iniciar la calibración. En este último caso, es el mensaje de error el que notifica al control que es imposible la calibración de línea alguna .

PESA_CALIBRE_UNIDADES

Este mensaje notifica al control el peso, en unidades célula, de la pieza (que en este caso generalmente será un peso patrón) que ocupaba la última taza no vacía de la línea que se indica. Este mensaje se lanza siempre que llega un sincronismo y se diagnostica que la taza que pasa sobre la célula de pesada no está vacía. Para más detalle, se estudiarán más adelante el proceso de los sincronismos.

Tx	PESA_CALIBRE_ UNIDADES **Orden: 1326**	0000	0000000	Linea	Índice de taza calibrada		Peso en unidades célula de la taza	
					LSB	MSB	LSB	MSB

ORDEN_FIN_CALIBRADO

Esta orden finaliza el modo de calibración. No tiene parámetros. Para ello, se cambia el modo actual a MANTENIMIENTO, se enmascara la interrupción de conversión y se ponen en *standby* los conversores.

Rx	ORDEN_FIN-CALIBRADO **Orden: 1325**	0101 1111	0000000	00000

8.3.4 Ajuste de cero

El ajuste de cero es el modo de funcionamiento 8. Éste *es un modo de funcionamiento metaestable* porque se sale automáticamente del modo tras realizar el ajuste.

La realización del ajuste de cero es esencial para el funcionamiento correcto del pesado. Analiza el comportamiento de cada conversor del sistema para obtener para

cada uno de ellos un valor de offset adecuado con el que el peso obtenido, en unidades célula, y en ausencia de taza ni peso sobre la célula de carga, sea un valor de 0 puntos. Por ello es imprescindible que el desarrollo del ajuste de cero se realice siempre con el patín vacío, quitando las tazas que pudieran estar sobre él. En otras palabras, *durante el ajuste no debe haber nada sobre la célula de pesada.*

Cada vez que se entra en este modo, se indican las líneas que se deben ajustar, siendo el proceso de obtención del offset para cada conversor el siguiente:

- El 'C26 realiza 16 ajustes de cero para cada conversor, tomando el offset obtenido en cada uno de ellos. Después se obtiene la media, el máximo y el mínimo. Se usa la media como offset de ajuste de cero. Para validar el ajuste se comprueba que $maximo - minimo < (media/128)$.
- Una vez obtenidos los valores de offset promedio y validado este resultado, se almacenan estos valores en NVRAM para tenerlos disponibles en posteriores encendidos de la máquina.
- De este análisis se obtiene un valor de offset para cada conversor, que será almacenado en NVRAM para evitar la repetición del proceso cada puesta en marcha de la máquina. Sin embargo, este proceso puede ser largo, por lo que el ajuste, muy intensivo computacionalmente, se ejecuta en el nivel 2. De esta forma, el proceso puede ser abortado mediante una mensaje de finalización previa ORDEN_FIN_ASY_AJUSTE_CERO. De otra forma, cuando el ajuste finaliza, se envía un mensaje al control indicando la finalización del modo metaestable y las líneas que se han podido ajustar.

Los mensajes del modo AJUSTE DE CERO son los siguientes.

ORDEN_AJUSTE_CERO

Esta orden sólo se atiende en modo MANTENIMIENTO (3). Al comenzar su servicio, pone el modo de funcionamiento a 8 (AJUSTE DE CERO), estado metaestable del que automáticamente saldrá. La petición de ajuste de cero incluye un máscara de bits (bit 0 ⟶ línea 0, ..., bit ⟶ línea 9) de las líneas de las que se solicita el ajuste. El proceso de servicio saca de *standby* los conversores, se obtienen las líneas útiles del sistema y si para al menos una de ellas, se ha pedido el ajuste, el proceso comienza. En éste, se llama a la función 'AjusteDeCero', que realiza los pasos siguientes:

- Se inicializan los conversores.
- Se aplican los pasos siguientes únicamente a las lineas a ajustar que son útiles.

- Realiza una configuración del conversor con los parámetros de trabajo.
- Se realiza una *Full Scale Calibration.*
- Se temporiza la espera para la finalización del comando anterior. Esta temporización depende la frecuencia de muestreo que se tome, ya que se basa en la adquisición interna de un número de muestras.
- Se realizan 16 ajustes de cero. De cada ajuste de cero se obtiene un valor de offset. Al finalizar las 16 iteraciones se realiza la media del valor de offset obtenido y se considera éste el ajuste de cero. Este valor obtenido como ajuste de cero se guarda en NVRAM como el offset del conversor y será usado cada vez que se programe el conversor de dicha línea. Este proceso se repite para cada una de las líneas que se ha pedido ajustar y son útiles.

Una vez finalizados los ajustes, se desenmascara la interrupción de conversión. Esto se hace sencillamente porque los ajustes generan una bajada en la señal de Ready al final de la 'Full Scale Calibration', que activa la interrupción de conversión, por lo que al final de la función que realiza el ajuste tenemos que aunque en ese momento la señal de Ready no se encuentra activa por nivel, ha sido activada por flanco anteriormente, quedando mantenida. Para eliminar este *latch*, habilitamos al final del proceso de servicio dicha interrupción. Cuando la interrupción llame a su proceso de servicio, éste detectará que el modo es 8 y se ejecutará un segmento de código que terminará el modo automáticamente (de ahí que sea un modo metaestable). Se elimina la interrupción que estaba pendiente y se termina el modo.

<table>
<tr><td rowspan="2">**Rx**</td><td rowspan="2">ORDEN_AJUSTE_CERO
Orden: 1310</td><td rowspan="2">0101
1111</td><td rowspan="2">0000000</td><td rowspan="2">00000</td><td colspan="2">Líneas a ajustar.
LSBit=Línea 1</td></tr>
<tr><td>LSB</td><td>MSB</td></tr>
</table>

Si las líneas que se pretenden ajustar no coincidieran con ninguna de las líneas útiles, se envía un mensaje de error, ponen los conversores en *standby* y automáticamente pasa de nuevo al modo 3 (MANTENIMIENTO).

ORDEN_FIN_ASY_AJUSTE_CERO

El proceso de servicio de esta orden, de nivel 1, habilita los mecanismos para la finalización prematura del ajuste de cero. Éste es, en ocasiones, necesario para que el control pueda terminar el modo para realizar otras acciones, dado que el tiempo de ejecución del ajuste de cero puede ser largo.

Cuando se fuerza la finalización del ajuste de cero, ésta sucede de la forma normal, enviando finálmente el mensaje ORDEN_FIN_AJUSTE_CERO e indicando cuáles de las líneas pedidas se pudo ajustar con éxito hasta el momento de la aborción del proceso.

Rx	ORDEN_FIN_ASY_ AJUSTE_CERO **Orden: 1313**	0101 1111	0000000	00000

ORDEN_FIN_AJUSTE_CERO

Este mensaje se envía cuando finaliza automáticamente el modo de de ajuste de cero, retornando al modo MANTENIMIENTO. Así, se notifica al control la finalización del ajuste de cero en curso. Antes de ello se desenmascara la interrupción de conversión y se ponen a *standby* los conversores. Se envían dos bytes de datos que forman una máscara de bits de las líneas que se han ajustado durante este proceso.

Tx	ORDEN_FIN_ AJUSTE_CERO **Orden: 1315**	0000	0000000	00000	Líneas a ajustar. LSBit=Línea 1	
					LSB	MSB

8.3.5 Verificación

Este modo se usa para verificar el funcionamiento del módulo de pesado. Con cada llegada de un sincronismo se envía un mensaje VERIFICACIÓN_TAZA para todas las tazas de las diferentes líneas con el mismo índice, indicando: índice de la taza, valor digital de la pesada, valor de la tara anterior y el peso en gramos de la pieza. Como puede verse, los datos enviados son tanto los datos obtenidos como los resultados alcanzados a partir de aquéllos. De esta forma, el control puede verificar el funcionamiento del módulo en su transducción a gramos de los datos obtenidos.

El proceso de verificación se realiza cada vez que le llega al sistema un sincronismo, indicando cual es la posición en la que se debe tomar el peso de la pieza. Posteriormente se estudiarán con más detalle.

Los mensajes del modo AJUSTE DE CERO son:

ORDEN_VERIFICACION

Esta orden sólo se atiende en modo MANTENIMIENTO (3). El proceso de servicio pone el modo actual a VERIFICACIÓN (7), saca de *standby* a los conversores,

toma aquellas líneas útiles que se han pedido verificar (de las que se han pedido verificar, sólo las líneas útiles) y, si al menos existe una, comienza la verificación. Caso de que exista alguna, desenmascara la interrupción de conversión para que comience la adquisición. Si no hay ninguna línea útil, envía un mensaje de error y pone de nuevo el *standby*.

Rx	ORDEN_VERIFICACIÓN **Orden: 104**	0101 1111	0000000	00000	Máscara de líneas a verificar. LSBit=Línea 1	
					LSB	MSB

VERIFICACION_TAZA

Este mensaje se envía cada llegada de un sincronismo cuando el modo de la pesadora es VERIFICACIÓN. Con cada llegada de un sincronismo se envía un mensaje VERIFICACIÓN_TAZA indicando línea e índice de la taza, valor digital de la pesada, de la tara anterior y el peso en gramos de la pieza.

Tx	VERIFICACIÓN_TAZA **Orden: 1205**	Índice línea	Índice de taza pesada		Valor digital de la pesada		Valor digital tara anterior		Peso pieza en gramos	
			LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

ORDEN_FIN_VERIFICACION

Esta orden finaliza el modo de verificación. No tiene parámetros. Para ello, se cambia el modo actual a MANTENIMIENTO, se enmascara la interrupción de conversión y se ponen en *standby* los conversores.

Rx	ORDEN_FIN_VERIFICACIÓN **Orden: 194**	0101 1111	0000000	00000

8.3.6 Osciloscopio

En este modo, la aplicación permite la visualización de las muestras adquiridas, a modo de osciloscopio. De la misma forma que todo osciloscopio, la visualización permite:

- Regular la base de tiempos indicando el número de muestras que se deben perder, 'numlostsampscope', entre dos selecciones consecutivas.

- Selección de la línea a monitorizar.

Otra capacidad del modo osciloscopio es la selección de la señal que se pretende visualizar. Las opciones son dos: a) la corriente de muestras adquiridas, o b) la salida del filtrado de dicha corriente. Cualquiera que sea la señal seleccionada, se aprovecha que en cada mensaje se pueden enviar hasta ocho bytes de datos para transmitir en cada mensaje paquetes de cuatro muestras, aprovechando al máximo la capacidad de transmisión de información en el mensaje DATOS_OSCILOSCOPIO.

Cada una de las palabras enviadas consta de dos campos, figura 8.4:

- El valor desplazado dos bits a la derecha, es decir, los 14 bits más significativos de la muestra (en los bits 13 a 0 de la palabra enviada).
- Un código de sincronismo que ocupa dos bits (los bits 15 y 14) y cuyo significado es el siguiente :

Bit15	Bit 14	Significado
1	1	Entre la muestra anterior y ésta ha llegado un sincronismo.
0	0	No ha llegado sincronismo entre la muestra anterior y ésta.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Código sincro-nismo		Muestra adquirida por el conversor despazada 2 bits a la derecha (tara/4)													

Figura 8.4: Campos delmensaje OSCILOSCOPIO.

El proceso de servicio de la interrupción es el encargado de dejar perder el número de muestras indicado y coleccionar los valores seleccionados en paquetes de cuatro, enviándolos cuando están completos. Como vemos, la información enviada también indica si entre una muestra y la anterior ha llegado un sincronismo. Así, además de la corriente de datos solicitada, se envían como información adicional los instantes en los que han llegado sincronismos (aspecto que es de gran utilidad, pudiendo ser representados como lineas verticales sobre la visualización de la corriente de datos enviada).

Los mensajes del modo OSCILOSCOPIO son:

OSCILOSCOPIO

La orden OSCILOSCOPIO sólo se atiende en modo MANTENIMIENTO (3). Dicha petición se realiza para una sóla línea, con una base de tiempos determinada. El proceso de servicio pone el modo actual a 5 (OSCILOSCOPIO), saca de *standby*

a los conversores e inspecciona si la línea que se pretende visualizar es una línea útil (aquéllas en las que el conversor funciona correctamente y hay conectada una célula de carga). Si al menos existe una, inicializa las variables y estructuras pertinentes, desenmascara la interrupción y comienza la operación del modo. Si la línea seleccionada no es una línea útil, se envía un mensaje de error y la pone de nuevo en *standby*.

Rx	OSCILOSCOPIO **Orden: 1317**	0101 1111	0000000	Linea	Muestras que pierde entre dos envíos	Muestras (0) Salida filtro (1)

DATOS_OSCILOSCOPIO

Este mensaje envía un paquete de cuatro muestras. Como se ilustra en la figura 8.4, su valor está desplazado dos bits a la derecha, por lo que ocupa 14 bits. Además se envían dos bits de información sobre los sincronismos. El proceso de servicio de la interrupción de los conversores es el encargado de ir saltando muestras, recopilándolas y enviando estos mensajes.

Desde el momento en que se activa el modo osciloscopio hasta que se desactiva, el envío de estos paquetes de muestras es continuo, permitiendo al control la visualización de los eventos internos del módulo de pesado.

Tx	DATOS_ OSCILOSCOPIO **Orden: 1319**	Índice línea	Marca sincronismo+valor digital	Marca sincronismo+valor digital	Marca sincronismo+valor digital	Marca sincronismo+valor digital

FIN_OSCILOSCOPIO

Esta orden finaliza el modo osciloscopio. Para ello, cambia el modo actual a MANTENIMIENTO, enmascara la interrupción de conversión y ponen en *standby* los conversores.

Rx	FIN_ OSCILOSCOPIO **Orden: 1318**	0101 1111	0000000	00000

8.3.7 Clasificación

Cuando el módulo de pesado se encuentra en el modo CLASIFICACIÓN, con cada llegada de un sincronismo se enviará un mensaje PESO_FRUTA indicando el peso de

las piezas que ocupan las tazas sobre la célula de carga. En este mensaje se envían dos pesos *en gramos:* el de la taza cuyo índice y línea se indica, y el de la taza del mismo índice pero de la línea siguiente. Asimismo, para cada uno de estos existe un byte de estado que describe:

1. Pesada OK.
2. Taza vacía.

Como vemos, el proceso de clasificación se realiza cada vez que llega un sincronismo, indicando cual es la posición en la que se debe tomar el peso de la pieza. Posteriormente se estudiarán con más detalle los sincronismos.

Los mensajes en CLASIFICACIÓN son:

INICIO_CLASIFICACIÓN

La orden INICIO_CLASIFICACIÓN sólo se atiende estando en modo 0. El proceso de servicio pone el modo actual a 2 (CLASIFICACIÓN), saca de *standby* a los conversores, toma las líneas útiles que hay y, si al menos una de las líneas del sistema es una línea útil, se desenmascara la interrupción de conversión y comienza el proceso de pesado. Si no hubiera ninguna línea útil se envía un mensaje de error y pone de nuevo en *standby* los conversores.

Rx	INICIO_ CLASIFICACIÓN **Orden: 415**	0101 1111	0000000 1111111	00000

PESO_FRUTA

Este es el mensaje mediante el cual se envía la información sobre el peso *en gramos* de las piezas que ocupan las tazas. En éste se envían un par de pesos, el correspondiente a la taza e índices indicados, y la taza de mismo índice de la línea siguiente. Éste es el mensaje de notificación de pesos durante el funcionamiento habitual de la máquina (CLASIFICACIÓN). Como puede verse, los bytes 7 y 8 notifican el estado de las tazas:

0. No contiene peso.
1. Pesada OK.
2. Taza vacía.

Tx	PESO_FRUTA **Orden: 1200**	Índice línea	Índice de taza pesada		Peso 1 (línea indicada)		Peso 2 (línea +1)		Estados de ambas tazas
			LSB	MSB	LSB	MSB	LSB	MSB	

FIN_CLASIFICACIÓN

Esta orden finaliza el modo CLASIFICACIÓN. Para ello, cambia el modo actual a modo 0, enmascara la interrupción de conversión y pone en *standby* los conversores.

Rx	FIN_CLASIFICACIÓN **Orden: 405**	0101 1111	0000000 1111111	00000

8.3.8 Proceso de los sincronismos

Un sincronismo es un mensaje CAN prioritario enviado por el *encoder* y previamente regulado para que sea realizado un envío cada vez que una taza pase por un cierto punto sobre la célula de carga (por ejemplo, éste puede regularse para ser envado, por ejemplo, cada vez que el centro de la taza esté situado a mitad del patinillo sobre la célula de carga, etc.).

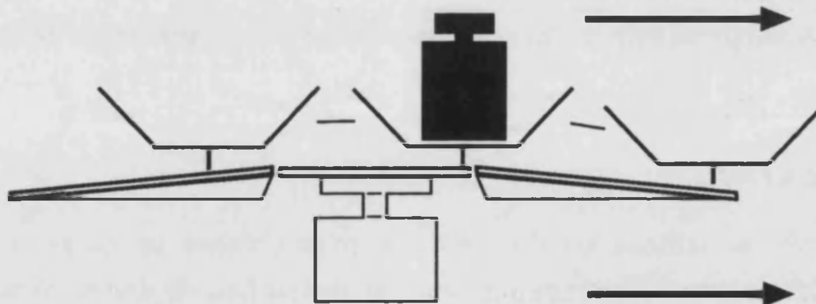


Figura 8.5: Momento en el que la taza empieza a salir sobre la célula de carga.

Así pues, los sincronismos indican el momento en el que procesar el resultado. Cada sincronismo incluye indirectamente el índice de la taza sobre la célula de carga, pudiéndose obtener éste sumando un offset (constante para cada calibradora) al índice que trae consigo el sincronismo. Este offset, denominado ***distancia al origen***, da cuenta de la distancia en tazas de la tarjeta de encoder al módulo de pesado.

La llegada de un sincronismo viene dada por la llegada de una orden:

La indicación de sincronismo es un mensaje asíncrono que siempre es atendido. Sin embargo, el servicio del sincronismo sólo procesa resultados en los siguientes modos de

Rx	SINCRONISMO_ PESADO **Orden: 40**	0101	1111111	00000	Índice de taza actual	
					LSB	MSB

funcionamiento: CLASIFICACIÓN, VERIFICACIÓN, CALIBRACIÓN y TARADO. Por defecto, lo que hace, cualquiera que sea el modo, es sumar la distancia al origen al índice de la taza que trae el sincronismo consigo, y comprobar que el resultado no sale de los índices válidos (a modo de direccionamiento circular, ya que en este caso la cinta habría dado una vuelta), en cuyo caso resitua el índice y guarda este valor para posibilitar un análisis de sincronismos perdidos a la llegada del siguiente.

Cuando comienza un modo de funcionamiento que utiliza sincronismos (CLASIFICACIÓN, VERIFICACIÓN, CALIBRACIÓN o TARADO), se dejan perder los primeros para dar tiempo a los buffers de muestras, intermedios y de salida a ser rellenados. El número de sincronismos que se dejan perder viene definido por la constante INNERSYNCHROS, que puede ser fácilmente cambiada con sólo recompilar el código de la aplicación.

Cuando un modo de funcionamiento que utiliza sincronismos comienza, se inicia un contador a cero, que será incrementado con cada sincronismo que se deje pasar hasta que éste se hace igual a la constante citada. A partir de dicho momento todos los sincronismos son atendidos. Se utiliza un valor INNERSYNCHROS=2 para dejar perder el primer par de sincronismos.

El proceso a partir de este momento es particular para cada modo de funcionamiento:

MODO TARADO

En este modo se utilizan los sincronismos para obtener las taras, o pesos de las tazas vacías, del sistema. El número de vueltas que se han de dar en el procedimiento de tarado no es fijo, pudiéndose obtener mayor precisión en el resultado incrementando el número de vueltas. Los valores obtenidos se almacenan en NVRAM para preservarlos entre diferentes puestas en marcha.

MODO CLASIFICACIÓN

Este modo tiene como objetivo enviar al control los pesos, en gramos, de las piezas contenidas en las tazas que pasan sobre la célula de carga, y marcar las tazas que pasan vacías.

A grandes rasgos, los pasos que se dan en el proceso del sincronismo en este modo son:

- **Retarado dinámico (si la taza viene vacía).** Ocurre cuando detecta que la taza viene vacía (es decir, cuando el valor en puntos de la diferencia entre el valor obtenido y el de la taza vacía es menor que la mitad de la taza vacía). Si la taza está vacía puede utilizarse el valor obtenido y el peso almacenado para obtener una nueva tara. Nótese que este proceso tiene mucho sentido, ya que el peso de la taza puede cambiar dinámicamente durante el funcionamiento por acumulación de suciedad, etc. En este punto se produce un reajuste del valor del peso de la taza vacía, calculándose de la misma forma que en tarado, pero con un coeficiente de convergencia distinto, pesando más el antiguo valor que el nuevo.

- **Pesado.** Cuando la taza no llega vacía, se obtiene la diferencia entre el valor obtenido y la tara. Ésta se pasa a gramos teniendo en cuenta la línea a la que pertenece la taza y la velocidad a la que se ha tomado el valor, y finalmente se marca dicho valor como 'Pesada OK'. Cuando la taza llega vacía se marca el valor enviado como ' Taza vacía', y se envía el valor en unidades del conversor de la nueva tara recalculada para dicha taza.

- **Envío del mensaje.** Se envía el mensaje PESO_FRUTA.

MODO VERIFICACIÓN

En este modo de funcionamiento, con cada llegada de un sincronismo se envía, para cada una de las tazas situadas sobre la célula de carga, un mensaje indicando: línea e índice de la taza, valor digital de la pesada, valor digital de la tara anterior y peso en gramos de la pieza.

Los pasos que se dan en el proceso del sincronismo en este modo son:

- **Retarado dinámico (si la taza viene vacía).** Ocurre cuando detecta que la taza viene vacía. En este caso puede utilizarse el valor obtenido y el peso almacenado de esta taza para obtener un nuevo valor del peso (en unidades del conversor) de la taza. Se utiliza el mismo coeficiente de convergencia que en el modo CLASIFICACIÓN.

  Este modo posee retarado dinámico para simular el comportamiento del modo CLASIFICACIÓN.

- **Pesado.** Cuando la taza no viene vacía, entonces se obtiene el peso en gramos y se marca la taza como 'Pesada OK'.

  Cuando la taza llega vacía se marca el valor enviado como ' Taza vacía'.

- **Envío del mensaje.** Se envía el mensaje VERIFICACION_TAZA.

MODO CALIBRACIÓN

El propósito del modo CALIBRACIÓN es cooperar con el control para obtener una curva de interpolación de puntos a gramos, que puede ser diferente para cada velocidad y línea. En este modo, se envía un mensaje con el valor digital de la pesada siempre que a la llegada de un sincronismo se reconoce taza no vacía. Así, se envía el valor digital de lo que pesa. En presencia de taza vacía no se envía mensaje.

Un esquema de la operación en este modo es el siguiente:

- Si la taza está vacía:
  - No ocurre Retarado dinámico.
  - Se marca la taza como vacía.
  - Se omite el envío del mensaje.
- Si la taza no está vacía:
  - Se envía el mensaje PESA_CALIBRE_UNIDADES con el valor digital de la pesada.

8.3.9 Petición y entrega de parámetros

En ocasiones puede ser interesante, por motivos de depuración e interactividad entre el control y la aplicación, acceder a una serie de parámetros de la aplicación tanto de memoria principal como de memoria no volátil. Por ejemplo, resulta interesante para el control acceder a la tara de una taza en concreto sin tener que pedirlas todas, inspeccionar el modo en que se encuentra el módulo de pesado, cambiar el modo de depuración basado en mensajes, conocer cuáles son las líneas útiles del sistema, etc. con sólo acceder a dicha variable en el módulo de pesado. Para realizar esta labor se ha creado una serie de mensajes cuyos servicios permiten al control escribir y leer parámetros de la aplicación (no sólo los parámetros que están guardados en NVRAM sino también los que en ese momento están activos en memoria). Estos parámetros se listan en la Tabla J.1 del apéndice J.

Como dos de los parámetros a los que se pretende acceder (offset del ajuste de cero y offset nivel de cero) son de 24 bits, el campo 'Dato' consta de 3 bytes (tres bytes en orden inverso, primero el menos significativo).

ESCRITURA DE PARÁMETROS

Cuando el control pretende modificar algún parámetro de la aplicación del módulo de pesado, se utiliza la orden ENVIO_PARAMETRO. Ésta debe incluir el nuevo valor

del parámetro y el código del parámetro, que junto con los dos datos auxiliares (véase Tabla J.1 del apéndice J) determinan unívocamente el parámetro que se pretende modificar. El uso incontrolado de esta orden en medio de la operación normal de la calibradora puede corromper el sistema y provocar malos funcionamientos, por lo que se recomienda un uso controlado. Como vemos, la ventaja de esta orden reside en su potencia y flexibilidad.

El código del parámetro es siempre necesario e identifica el parámetro a modificar. La línea también es un índice para parámetros que dependen de la línea (como offset de ajuste de cero, del cual hay uno para cada canal). Los datos auxiliares se usan para los coeficientes de calibración, ya que éstos dependen también de la velocidad y del tipo de coeficiente (pueden ser coeficientes l, m y n donde $gramos = l \cdot x^2 + m \cdot x + n$). Ver Tabla I para la relación de códigos de cada parámetro y sus índices.

Rx	ENVÍO_PARÁMETRO **Orden:**	Índice línea	Índice de taza pesada			Código del parámetro pedido	Dato auxiliar 1	Dato auxiliar 2
			LSB	MSB	MMSB			

LECTURA DE PARÁMETROS

Para leer una variable que esté utilizando la aplicación o guardada en la NVRAM, se usa el mensaje de petición PETICIÓN_PARÁMETRO, donde se debe indicar el código de la variable que se pretende leer y los datos auxiliares si los necesita (Tabla J.1 del apéndice J). Cuando se escribe un parámetro en NVRAM se valida el flag testigo correspondiente al parámetro escrito.

Rx	PETICIÓN_PARÁMETRO **Orden:**	0101	0000000	Índice línea	Código del parámetro pedido	Dato auxiliar 1	Dato auxiliar 2

Cada mensaje de petición es respondido con una contestación del dato pedido a través del mensaje ENTREGA_PARÁMETRO. Para comprobar que el dato recibido corresponde a la petición realizada, el control dispone adjunto en el mensaje del código del dato, y los parámetros con los que se realizó la petición.

Tx	ENTREGA_PARÁMETRO **Orden:**	Índice línea	Índice de taza pesada			Código del parámetro pedido	Dato auxiliar 1	Dato auxiliar 2
			LSB	MSB	MMSB			

8.3.10 Mensajes asíncronos

Como ya sabemos, los mensajes asíncronos son aquéllos que pueden ser servidos en cualquier modo de funcionamiento. En este punto ya hemos visto la misión de algunos de ellos como: ENVIO_PARAMETRO, PETICION_PARAMETRO, ENTREGA_PARAMETRO, ORDEN_FIN_ASY_AJUSTE_CERO y SINCRONISMO_PESADO. Sin embargo, existen otros muchos con objetivos no menos importantes.

ERROR_TARJETA_PESADO

La aplicación utiliza este mensaje para enviar al control los códigos de los errores que se van produciendo. Los tipos de error y sus fuentes están expresados en la Tabla J.2 del apéndice J. El control puede hacerse eco de ellos para resolver los problemas notificados, o simplemente tenerlos en cuenta para conocer el estado global del sistema.

Tx	ERROR_TARJETA_PESADO **Orden: 85**	0000	0000000	0000	Número Error

DEBUG_MESSAGE

A través de este mensaje se envían al control notificaciones de depuración. Estos sirven para indicar estados o variables de la aplicación y han sido muy usados para la depuración de la aplicación durante el desarrollo. Para el uso de estos mensajes, es necesario la habilitación del modo de depuración basado en mensajes a través de los parámetros 'feedbackmessage' y 'sendmodeafterservice', ambas con carácter booleano.

Tx	DEBUG_MESSAGE **Orden: 86**	0000	0000000	0000	Sentido	Byte1	Byte 2

INICIO_LINEA

Esta orden es enviada por el *encoder* cada vez que la calibradora ha dado una vuelta.

Rx	INICIO_LÍNEA **Orden: 50**	1111	1111111	00000

ERROR_PERDIDA_SINC_MESSAGE

Este mensaje es utilizado cada vez que llega un mensaje INICIO_LINEA para enviar al control notificación de los sincronismos que se han perdido durante la vuelta anterior de la cinta de la calibradora. Esta es una forma de informar al control sobre los sincronismos que se han perdido durante la última vuelta, aunque si el modo de funcionamiento es CLASIFICACIÓN, VERIFICACIÓN o CALIBRACIÓN también se articulan otros mecanismos para informar no sólo de cuantos sincronismos ha perdido sino de cuales son los sincronismos que se pierden, mientras van sucediendo.

Tx	ERROR_PERDIDA_ SINC_MESSAGE **Orden: 208**	0000	0000000	00000	Número Sincronismos perdidos

ERROR_PERDIDA_GRAVE_SINC_MESSAGE

Notificación ante un gran número de sincronismos perdidos.

Tx	ERROR_PERDIDA_ GRAVE_SINC_ MESSAGE **Orden: 204**	0000	0000000	00000	Número Sincronismos perdidos

VELOCIDAD

A la llegada de este mensaje, la aplicación actualiza el parámetro 'velocidad' (variable global que describe la velocidad de la máquina) con el valor que trae el primer byte de datos del mensaje. Esta velocidad se da en frutas por segundo.

Rx	VELOCIDAD **Orden: 1405**	1111	1111111	00000	Tazas/ segundo

8.3.11 Mantenimiento

Al modo mantenimiento se accede únicamente desde el modo 0 (sin modo). Incluye todos los modos de funcionamiento que corresponden a las tareas propias de mantenimiento como ver líneas a través de un osciloscopio, calibrar, ajustar el cero de las líneas, etc.

MANTENIMIENTO

La orden MANTENIMIENTO sólo se atiende estando en modo 0 (sin modo). Pone el modo de funcionamiento actual a 3 (MANTENIMIENTO).

Rx	MANTENIMIENTO **Orden: 105**	0101 1111	0000000 1111111	00000

FIN_MANTENIMIENTO

El modo mantenimiento se finaliza mediante la orden FIN_MANTENIMIENTO. Ésta sólo se atiende en modo 3 (MANTENIMIENTO). Pone el modo actual a 0 (sin modo).

Rx	FIN_ MANTENIMIENTO **Orden: 195**	0101 1111	0000000 1111111	00000

8.3.12 Sin modo

El modo de funcionamiento denominado 'Sin Modo' es un modo en el que puede decirse que el módulo de pesado está "en punto muerto". Este modo no tiene un cometido específico y sólo aceptará órdenes asíncronas y pasos a cualquiera de los siguientes tres modos: Clasificación, Configuración o Mantenimiento.

Capítulo 9

Resultados

El procesado de la señal implementado puede dividirse en dos partes: por una parte el acondicionamiento de la señal o preprocesado y, por otra, el algoritmo de estimación del peso.

La aplicación de los algoritmos propuestos sigue manteniendo la necesidad de sincronismos. El sincronismo es un evento indicador de la situación concreta de una taza sobre la zona de pesado. Este mecanismo se utiliza tradicionalmente en sistemas de pesado para determinar un punto como referencia temporal para procesar el peso de la pieza sobre la célula de carga.

Se propondrá un algoritmo para sustituir la necesidad de hardware externo de generación de sincronismos (encoder, etc.) por un par de sensores de aceleración y la aplicación de procesado digital. No obstante, la generación de sincronismos de esta forma solo es adecuada para bajas velocidades de la cadena de arrastre.

Posteriormente se analizan los resultados obtenidos con el método propuesto y se describen las prestaciones de los algoritmos implementados en la calibradora MAX-SORTER, basados en un sincronismo inteligente que introduce un promediado hacia atrás en función de la velocidad. Aunque, por razones de privacidad no pueden describirse los algoritmos implementados, es útil ofrecer una estadística de sus prestaciones.

9.1 Selección del método de preprocesado

Para elegir un método de preprocesado de entre los planteados en el capítulo 4, se realiza una elección razonada basada en criterios objetivos. Ésta no se basará en el ensayo sobre un solo tramo de adquisición, sino sobre un conjunto de tramos

adquiridos en las mismas condiciones, y agrupados como pruebas distintas de un mismo experimento. Para ello se ha realizado varios experimentos, con distintos pesos a una velocidad de 20 frutas/segundo. Cada uno de éstos consiste en el marcado de una taza n que se utiliza para portar siempre el mismo peso p en vueltas diferentes de la cinta de arrastre.

Tras la realización de las adquisiciones, se procesa cada conjunto de tramos solapados obtenidos en cada experiencia mediante los distintos algoritmos planteados. Finalmente se realizará una comparativa basada en los siguientes criterios:

a) **Grado de eliminación de las oscilaciones de las zonas peso.** Se mide como diferencias de pico a pico en las *zonas peso* de la señal preprocesada.

b) **Dispersión de los tramos respecto de un patrón promedio.** Aplicado al conjunto de tramos de cada experimento, podemos observar en qué medida siguen un patrón promedio en la zona peso. Esto se calcula marcando manualmente el *inicio* y *final* de una zona peso. Entonces, para cada tramo, se puede calcular τ_i,magnitud que define el promedio del valor absoluto de la diferencia con el patrón promedio del tramo i, que no tiene porqué ser el valor del peso. Ésta se calcula mediante la expresión:

$$\tau_i = \frac{1}{(final - inicio + 1)} \sum_{n=inicio}^{final} abs(x_i(n) - \bar{x}(n)) \tag{9.1}$$

El valor obtenido para cada experimento será dp, o lo que es lo mismo, el promedio de τ_i :

$$dp = \frac{1}{i} \sum_{i} \tau_i \tag{9.2}$$

c) **Convergencia tras la transición.** Básicamente consiste en que, tras la transición de paso de una taza con peso a una taza vacía o con otro peso, el peso de la segunda taza pueda ser determinado con independencia del peso de la taza anterior. Este criterio ha sido introducido básicamente para comprobar el comportamiento de los sistemas adaptativos utilizados, ya que éstos dependen de la memoria del sistema.

9.1.1 Criterio 1: Presencia de oscilaciones en la zona peso

La selección del algoritmo de preprocesado resultará en la elección de aquél que mejor acondicione la señal para la aplicación de un algoritmo de obtención del peso.

El estudio se realiza para la velocidad máxima de 20 frutas/segundo, por ser el peor de los casos en cuanto a: *a)* oscilaciones en la zona peso, *b)* menor número de muestras disponibles, y *c)* con transiciones más acusadas en proporción a la extensión de las zonas peso.

Una primera observación del comportamiento de los filtros adaptativos se ilustra en la figura 9.1. Ésta se obtiene para una pesa de 197 gr a 20 frutas/segundo, mostrando una comparación del resultado del preprocesado de los sistemas adaptativos. En negro se representa el resultado de la variante del LMS con 10 iteraciones de Fletcher-Reeves en el espacio intermuestral y en rojo la variante del momento. Ambas pueden compararse con el resultado de la deconvolución del modelo ARMA, en azul. Nótese que esto representa el preproceso del conjunto de tramos obtenidos, de ahí que cada algoritmo venga representado por un conjunto de tramos representados con el mismo color.

La figura 9.1 muestra que el preproceso de los sistemas adaptativos mantiene grandes oscilaciones y que se puede realizar un criba inicial, eliminando de la comparativa estos dos sistemas adaptativos. La oscilación es muy grande tanto en la zona peso de la taza llena como en las tazas siguientes que pasan vacías. El algoritmo basado en la deconvolución mediante el modelo ARMA es mucho más suave en las zonas peso con menor transición.

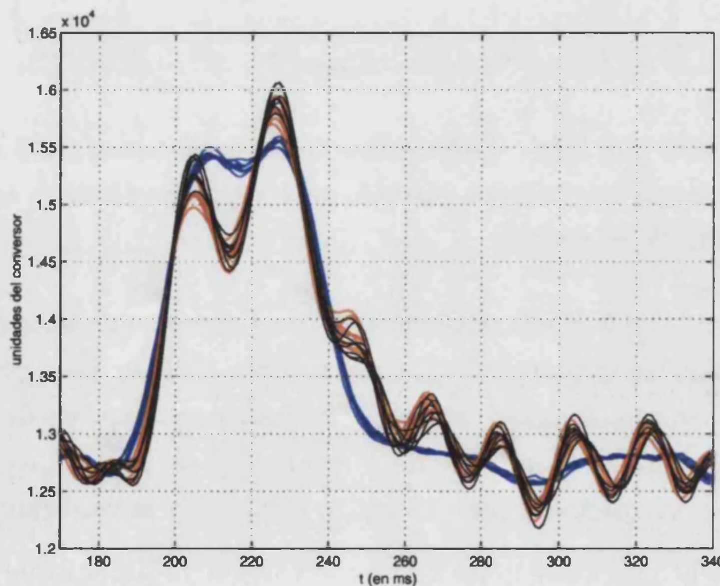


Figura 9.1: Comparación del Fletcher-Reeves (negro), la variante del momento (rojo), y la deconvolución con el modelo ARMA (azul).

Respecto a los demás algoritmos, se han realizado una serie de experimentos, utilizando pesos diferentes, para estudiar la oscilación de pico a pico. Como ejemplo,

las figuras 9.2 y 9.3 ilustran respectivamente como los distintos algoritmos (deconvolución del modelo ARMA, en azul; promediado de orden 20, en rojo; ALMS en verde; MLMS en negro) actúan sobre las zonas peso de las tazas llenas, con 180 y 40gr respectivamente. Nótese que cada algoritmo equivale a una curva, a diferencia de la figura 9.1 donde cada algoritmo es representado por un conjunto de éstas. La curva representada es la correspondiente al promediado de todos los tramos solapados y preprocesados, denominada "patrón promedio", estas figuras se han representado de esta forma para que la ilustración gane en claridad.

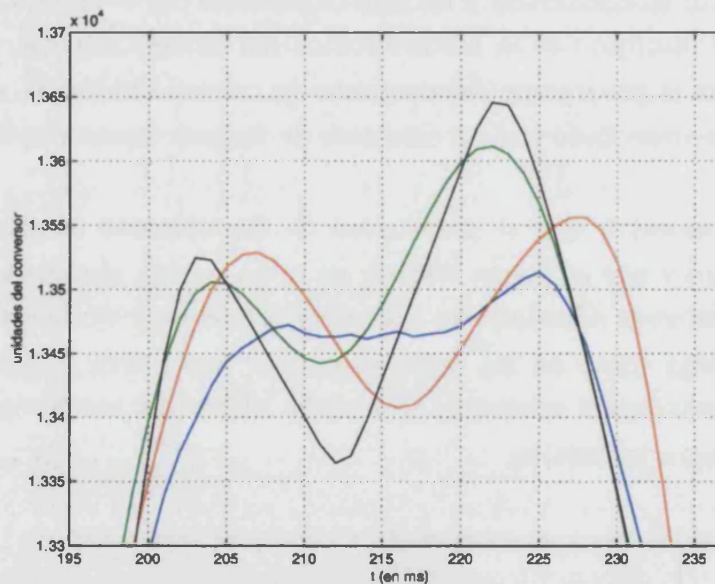


Figura 9.2: Taza llena con 50gr. Comparativa de las oscilaciones en la zona peso de los algoritmos de preproceso comparados (ARMA, en azul; promediado de orden 20, en rojo; ALMS en verde; MLMS en negro).

En las figuras 9.2 y 9.3 puede observarse el buen comportamiento del algoritmo de preprocesado basado en la deconvolución del modelo ARMA. Esto ocurre para todas las velocidades y cualquier rango de pesos, incluso para taza vacía. Para expresar esto de forma cuantitativa se ha medido la oscilación de pico a pico promedio para cada algoritmo, en un rango de pesos de 40 a 200gr, con la calibradora funcionando a 20 frutos/segundo.

Obviamente, cuanto mayor es el peso mayor es la oscilación. Sin embargo, podemos representar conjuntamente los resultados obtenidos, aplicando porcentajes respecto al algoritmo que posee mayor oscilación, con cada peso. Esto se ilustra en la figura 9.4, donde el rango de oscilación de pico a pico obtenido para un rango de experimentos con pesos de entre 40 y 200gr. Como puede verse, la oscilación máxima siempre es la del algoritmo MLMS.

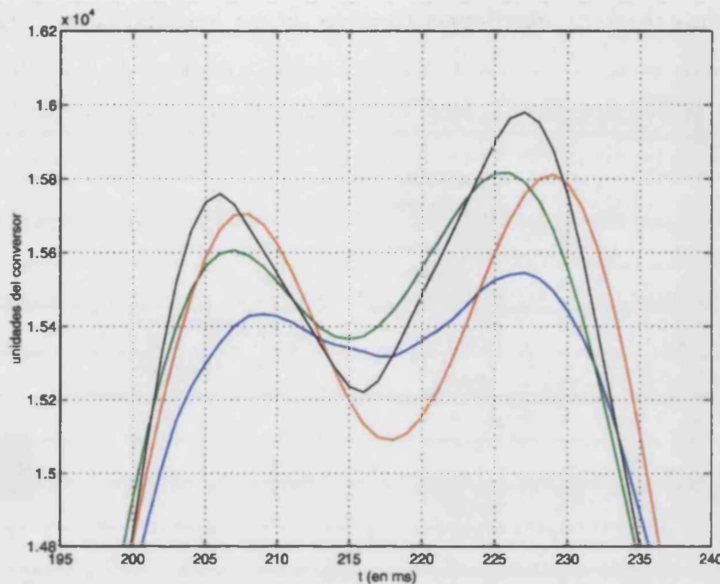


Figura 9.3: Taza llena con 200gr. Comparativa de las oscilaciones en la zona peso de los algoritmos de preproceso comparados (ARMA, en azul; promediado de orden 20, en rojo; ALMS en verde; MLMS en negro).

Como vemos, el algoritmo MLMS posee siempre un máximo de oscilación, de ahí que ocupe una línea en 100%. Le sigue el promediado de orden 20, con unas oscilaciones del 50 al 91% respecto a la del MLMS en los experimentos con pesos en el rango de 40 a 200gr. Por otra parte, el algoritmo ALMS ocupa una franja del 45 al 60% de dicha oscilación, y por último el algoritmo basado en la deconvolución del modelo ARMA es el que menos oscilación posee, en el rango del 16 al 31%. Según este criterio este último es el algoritmo más adecuado.

9.1.2 Criterio 2: Dispersión respecto de un patrón promedio

A la hora de aplicar el segundo criterio, se utilizan las expresiones 9.1 y 9.2. En este caso existe gran similitud entre los valores obtenidos para la dispersión relativa en los diferentes experimentos con pesos distintos (en el rango de 40 a 200gr). Por ello, puede utilizarse el valor promedio de la dispersión relativa, dp en la expresión 9.2, de cada algoritmo para representar gráficamente los datos en que se basa el segundo criterio de selección (Srinath, 1996).

En este caso, vemos que los algoritmos que más fielmente reproducen el patrón promedio son el sistema adaptativo ALMS y la deconvolución del modelo ARMA.

Dado que éstos son considerablemente superiores al sistema adaptativo MLMS y al promediado de orden 20, los eliminaremos para la consideración del tercer criterio.

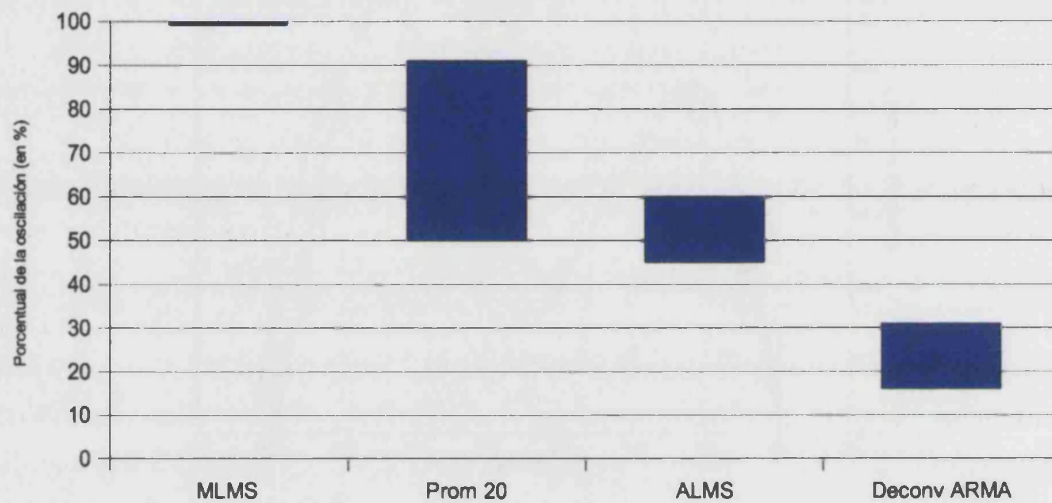


Figura 9.4: Diagrama de magnitudes de oscilación de pico a pico en la zona peso relativas a la máxima oscilación de pico a pico.

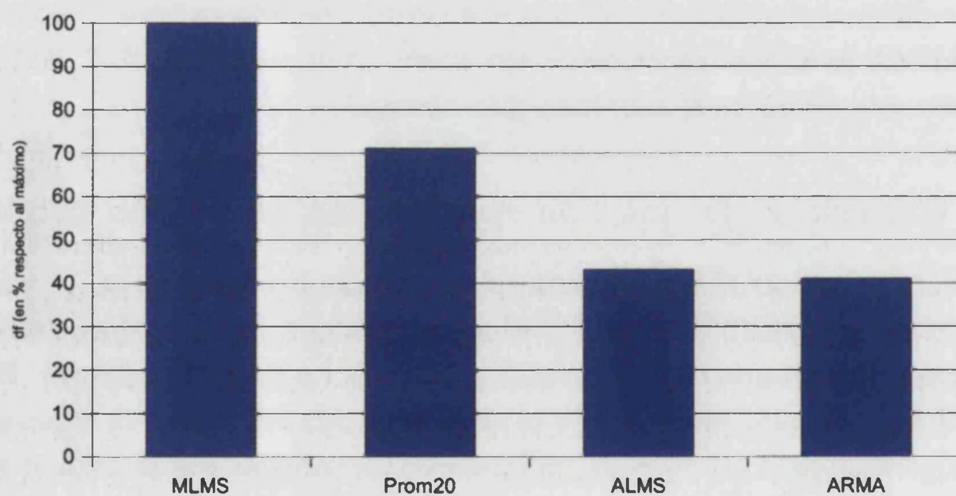


Figura 9.5: Dispersión promedio para cada algoritmo respecto del patrón promedio.

A partir de este momento continuamos únicamente con la deconvolución del modelo ARMA y el ALMS.

9.1.3 Criterio 3: Convergencia del comportamiento tras la transición

Este es un punto importante, introducido básicamente para comprobar el comportamiento de los sistemas adaptativos utilizados, ya que éstos dependen de la memoria del sistema.

En este tercer criterio, vamos a centrarnos en el peor de los casos, considerando una pieza de 200gr con la calibradora funcionando a 20 frutas/segundo. En estas condiciones se ha tomado un conjunto de tramos correspondientes a vueltas de la cinta en las que la taza n lleva el citado peso de 200gr, y vueltas en las que dicha taza pasa también vacía. No obstante, en las figuras que se citan en esta sección sólo aparecen los patrones promedio.

En la figura 9.6, la curva verde corresponde al patrón promedio de los tramos que llevan el peso de 200gr en la taza n, procesados con el ALMS. La curva roja corresponde al patrón promedio de los tramos con la taza n vacía, también procesados con el ALMS. De acuerdo con esto, se puede apreciar cómo, tras dos tazas vacías, la curva verde comienza la transición de la taza n, mientras que la curva roja continúa con la taza n vacía. Nótese como antes de dicha transición, ambas coinciden aún, pudiendo encontrar pequeñas diferencias entre ambos patrones, únicamente atribuibles a ruido aleatorio.

Las curvas azul y negra son análogas a las anteriormente descritas, pero el algoritmo de preprocesado utilizado ha sido, en ambas, la deconvolución basada en el modelo ARMA. Así, la curva azul corresponde al patrón promedio de los tramos que llevan el peso de 200gr en la taza n, y la curva negra al patrón promedio de los tramos con la taza n vacía. También en este caso se puede apreciar como ambas curvas coinciden, antes de comenzar la transición, excepto en lo correspondiente a una base de ruido.

La figura 9.7 muestra el comportamiento de los patrones promedio descritos durante y tras la transición. Como puede verse, para el caso del patrón azul (procesado con la deconvolución del modelo ARMA) la transición se extiende hasta más de la mitad de la zona peso de la taza $n+1$. En este caso, a partir del momento en el que la señal alcanza el nivel de la zona peso de la taza $n+1$, el comportamiento de este patrón se asimila al del patrón negro.

Por otro lado, nótese como el patrón verde, preprocesado con ALMS, no se asimila al rojo tras la transición, aunque presente una tendencia a hacerlo, cosa que conseguirá varias tazas después. Podemos tildar éste como un "efecto memoria" que aparece

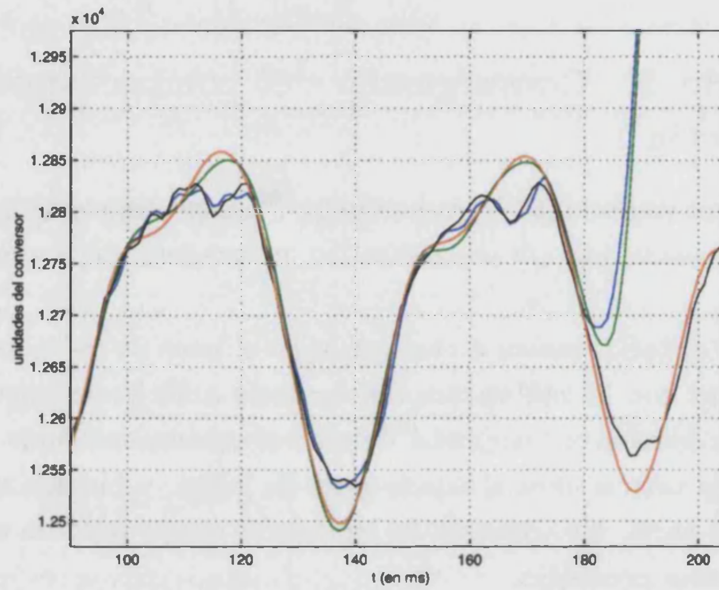


Figura 9.6: Comportamiento de los patrones promedio antes de iniciar la transición.

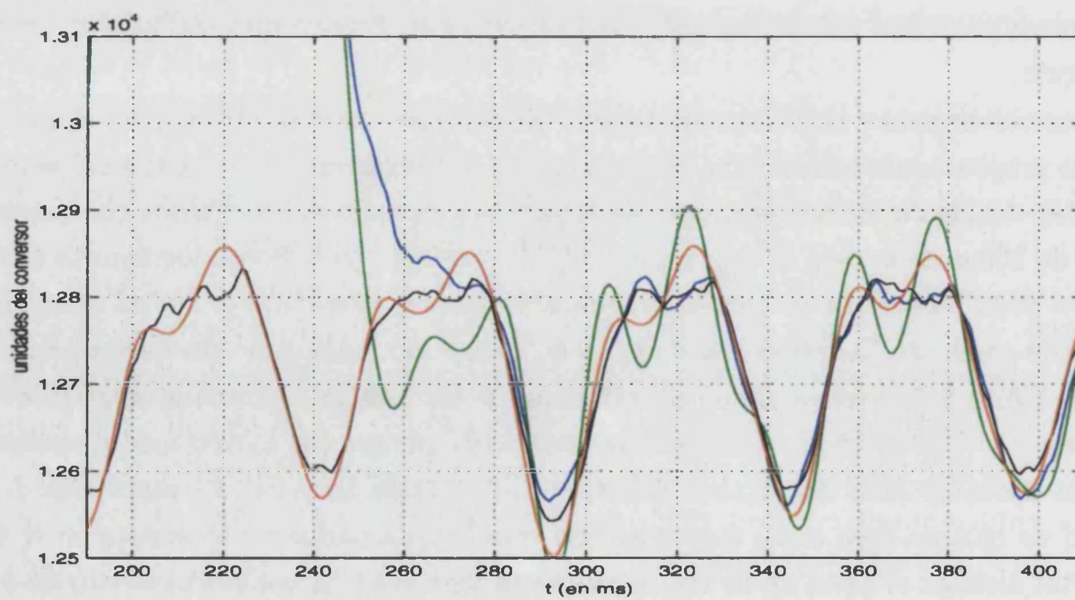


Figura 9.7: Comportamiento de los patrones promedio durante y tras la transición.

por haber sufrido una gran transición, y que obviamente no compensa la rápida transición que este algoritmo alcanza. La aplicación del algoritmo de obtención del peso, sección 9.2, a la taza *n+1* nos dice que el patrón verde proporciona para la taza *n+1* un peso 3.5gr menor que el obtenido para la misma taza con el patrón rojo. Esta diferencia de peso se basa únicamente en el hecho de que en la taza n hubiera o no pesa, y aunque esta diferencia se hace menor cuando disminuye la transición, no es tolerable, máxime cuando la aplicación del algoritmo de obtención del peso a la taza *n+2* proporciona una diferencia de 0.5gr. Nótese como este "efecto memoria" afecta a varias tazas tras la transición.

Esta tendencia es producida por una transición anterior, y va disminuyendo con el paso de las tazas hasta estabilizarse, coincidiendo finalmente los patrones promedio con y sin taza (verde y rojo). Obviamente, este efecto memoria es inadmisible, ya que la obtención del valor real del peso dependería no sólo de la velocidad de la cinta y del peso en la taza procesada, sino también de los pesos que ocuparon tazas anteriores. Una posible alternativa para la disminución de este efecto puede ser la de disminuir el orden del promediado que utiliza el ALMS, sin embargo esto deteriora las prestaciones tendiendo a las obtenidas para el MLMS.

En este punto, podemos establecer que el algoritmo más adecuado para el preproceso de la señal es la deconvolución basada en el modelo ARMA. Además de las buenas prestaciones obtenidas para los criterios 1 y 2, el proceso es lineal y no presenta el efecto memoria anteriormente descrito.

9.2 Algoritmo para la estimación del peso

Básicamente, el problema de la estimación del peso que ocupa una taza se basa en el cálculo de la diferencia del valor obtenido para la taza llena y para la misma taza vacía. El valor del peso de una taza vacía suele denominarse *tara* y su obtención se realiza a través del procedimiento de tarado mediante varias vueltas de la cinta a una velocidad lenta, con lo cual ésta es calculada con una buena precisión. Por otra parte, el valor del peso de la taza llena se ha de realizar con la máquina funcionando en clasificación. Este valor se obtiene típicamente del promediado de un segmento de puntos de la zona peso de la taza.

El problema radica en seleccionar cuál es el segmento de puntos a promediar. Nótese que la longitud de este intervalo dependerá de:

- **La velocidad**. La longitud de la zona peso se acorta con la velocidad y tanto la longitud del segmento a promediar.

- **La transición entre los pesos de tazas consecutivas.** Las grandes transiciones restan puntos a la longitud de este segmento a promediar.

El algoritmo de obtención de peso utilizado en el módulo de pesado de la calibradora MAXSORTER se basa en un sincronismo inteligente con promediado variable dependiente de la velocidad. Las prestaciones de éste son suficientes, indicadas en las estadísticas realizadas incluidas en la sección 9.4.2, para alcanzar su objetivo: una desviación estándar en la precisión de un gramo a una velocidad de 15 frutas/segundo. Sin embargo, aquí se plantea un algoritmo "inteligente" que adapta automáticamente el segmento a promediar independientemente de la velocidad y la transición.

El algoritmo planteado se ha probado *off-line*, de esta forma se extrae información equivalente a los sincronismos mediante procesado digital de la señal. Una implementación en tiempo real también puede basarse en el uso de sincronismos, para marcar los extremos del segmento.

Antes de continuar con la descripción del algoritmo, cabe notar que existe una limitación empírica, impuesta para su utilización, consistente en que la diferencia del peso de una taza (p_n) a la siguiente (p_{n+1}) no puede ser mayor de 50 gramos ($p_{n+1} - p_n > -50gr$). Esto es así para que la transición de la señal entre dos zonas peso pueda dar lugar a una forma concava, necesaria para que el algoritmo haga evolucionar los extremos del segmento de forma idónea.

Esta limitación es consecuencia directa de la dinámica de la célula de carga, y puede mejorarse ampliamente con la modificación de longitud de la zona de pesado. En el prototipo actualmente disponible, la plataforma de la célula de carga representa el 90% de la distancia entre tazas; es decir, hay unos 65mm de zona de pesado y 7mm de transición entre tazas. Esta relación podría modificarse y hacer que la longitud de la zona de pesado representara un 70% de la distancia entre tazas. Con ello desaparecera la limitación citada y la pérdida de precisión, según simulaciones realizadas, sería de 0.5g.

El algoritmo consiste en la adaptación del segmento a promediar hasta una longitud y posición idónea. Para ello se siguen los siguientes pasos:

- Inicialmente se han de inicializar los extremos del segmento a la mitad del flanco izquierdo y derecho correspondiente a la taza procesada.

- A partir de este momento comienza una iteración por la cual se mueve el extremo izquierdo (derecho) del segmento hacia la derecha (izquierda) siguiendo en función de la primera derivada de la señal, que crecerá (decrecerá) hasta un cruce por cero, que se tomará como extremo provisional izquierdo (derecho) del segmento. Si se produce un máximo (mínimo) antes del cruce por cero se tomará éste en su defecto.

- A continuación se incrementa/decrementa el extremo del segmento que mayor nivel posee hacia su correspondiente flanco hasta que éste adquiera un nivel similar al otro.
- Reajustadas las marcas, únicamente cabe promediar el segmento entre ellas para estimar el valor escalar que será el valor del peso (en unidades del conversor).

Una vez estimado el valor del peso en unidades del conversor, se habrá de restar a la tara almacenada para la taza vacía. El resultado puede entonces considerarse proporcional a su valor en gramos, no obstante lo más adecuado es someterlo a una conversión calculada a partir de una calibración previa de la máquina.

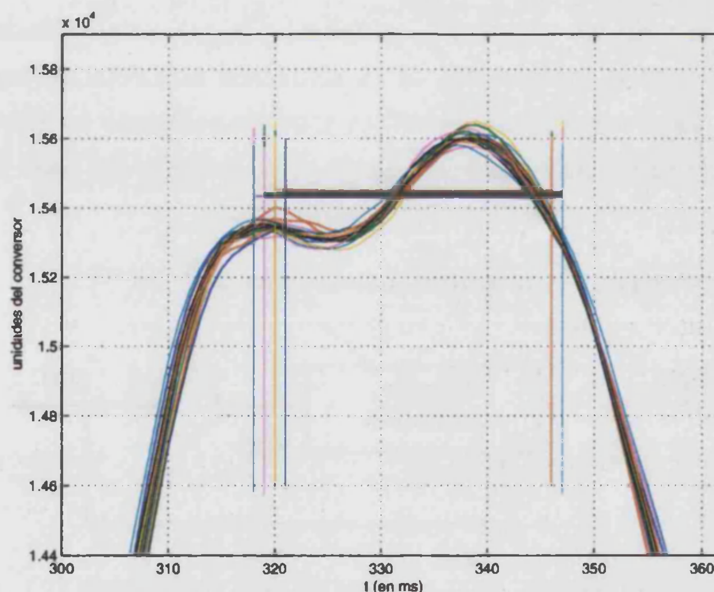


Figura 9.8: Aplicación del algoritmo a un conjunto de casos del mismo experimento con una pesa de 197gr a 15 frutas/segundo.

La figura 9.8 presenta un ejemplo por el cual se realiza la aplicación del algoritmo a diferentes tramos, correspondientes a un peso de 197 gramos obtenidos a una velocidad de 15 frutas/segundo. En éste se puede ver para cada uno de los extremos de los segmentos, el nivel promedio obtenido. Nótese que la desviación estándar respecto del valor medio es baja, 7 unidades del conversores, lo cual equivale a aproximadamente $\frac{1}{2}$ gramo.

9.3 Uso de los acelerómetros

El planteamiento inicial del uso de los sensores de aceleración fue obtener entradas auxiliares que permitieran eliminar, mediante técnicas adaptativas, las vibraciones y

oscilaciones que la señal de la célula de carga de línea presenta. Este planteamiento no resultó satisfactorio aunque se obtuvieron aplicaciones como la generación de sincronismos, que se describirá a continuación.

Sin embargo, hemos comprobado que los sensores de aceleración son una excelente herramienta para medir, a partir de la vibración, la "salud"de la maquina, lo cual permite, con un sencillo procesado digital, asistir a los montadores en la puesta en marcha y equilibrado inicial de la máquina e informar a los usuarios finales sobre la necesidad de ajustes o de mantenimiento.

9.3.1 Generación inteligente de sincronismos

Para la aplicación de los algoritmos propuestos sigue manteniéndose la necesidad de sincronismos (eventos indicadores de la situación concreta de una taza sobre la zona de pesado). Este mecanismo se utiliza tradicionalmente en sistemas de pesado para deteminar una referencia temporal usada en el proceso del peso de la pieza sobre la célula de carga.

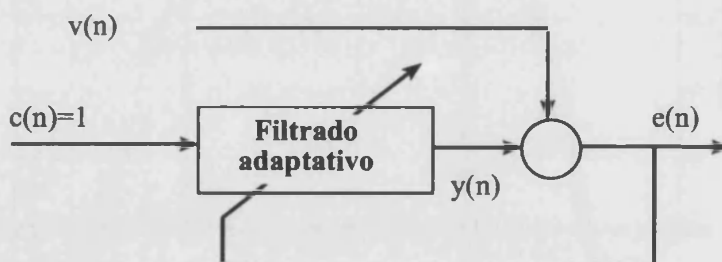


Figura 9.9: Esquema del filtrado adaptativo.

En este punto, proponemos un algoritmo para sustituir la necesidad de hardware externo de generación de sincronismos (encoder, etc.) por un par de sensores de aceleración y la aplicación de procesado digital. Este planteamiento sólo será válido para bajas velocidades de la cadena de arrastre, hasta 9 frutas/segundo.

La generación inteligente del sincronismo considera tanto la vibración del cuerpo de la célula de carga de línea (fijada al chasis de la calibradora), como la vibración del vástago de ésta (cuya parte superior es la zona de pesado).

Es obvio que la vibración de ambas partes de la célula de carga será diferente ya que, aunque recogen componentes comunes, la vibración del vástago comprende, además, la inducida por la entrada de las tazas en la zona de pesado. Esto se ilustra en las figuras 9.10 y 9.11, que indican las vibraciones adquiridas, simultáneamente, del vástago, $v(n)$, y del cuerpo de la célula de carga de línea, $c(n)$. Como puede verse, ambas son claramente distintas y no es posible obtener una información sobre

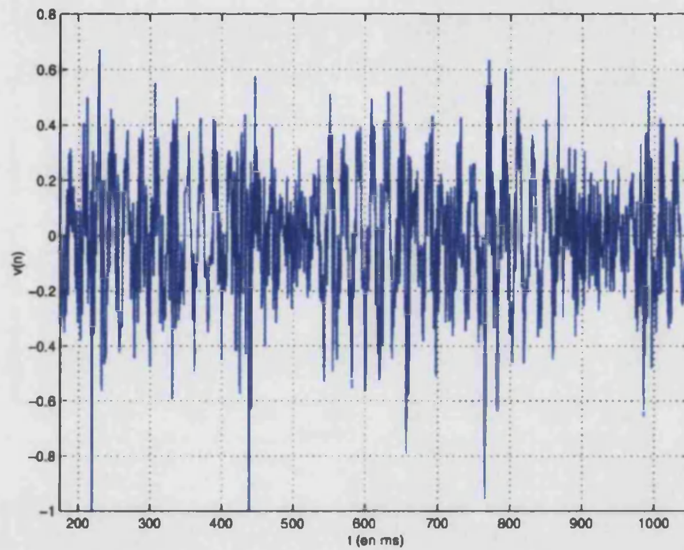


Figura 9.10: Vibración del vástago de la célula de carga.

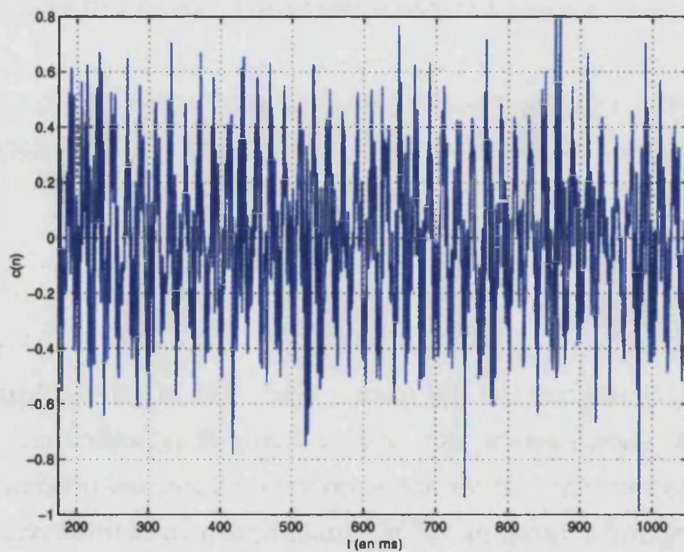


Figura 9.11: Vibración del cuerpo de la célula de carga.

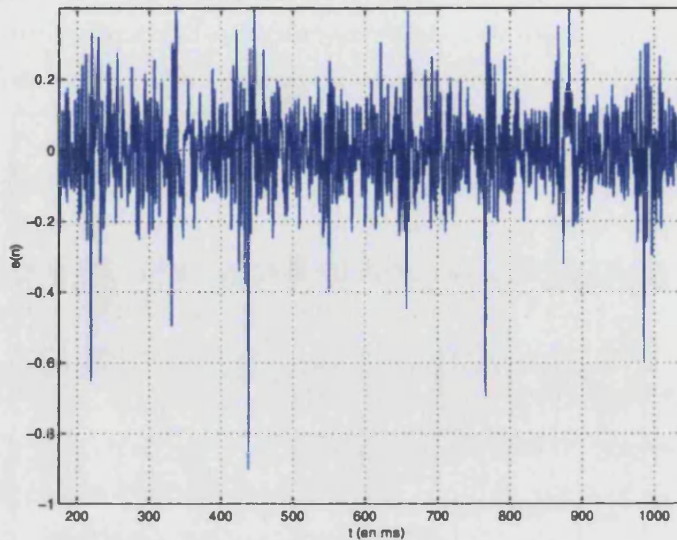


Figura 9.12: Señal de error procedente del filtro adaptativo: Sincronismos.

el momento en que entra una taza a la zona de pesado. Para ello se utiliza un sistema adaptativo LMS básico, figura 9.9, que eliminará las componentes que se correlacionan entre ambos registros, $v(n)$ y $c(n)$. Éste utiliza una función de coste, J, que minimiza el error cuadrático mínimo, expresiones 4.8 a 4.13, con un filtro adaptativo de 100 pesos.

La figura 9.12 muestra los eventos periódicos que indican el paso de la taza a la zona de pesado, es decir, los sincronismos que se han detectado utilizando el algoritmo propuesto.

Sin embargo, este método resulta complejo y el coste es superior al de una fotocélula que, situada sobre el cuerpo de la célula de carga, detectara el paso de cada taza, generando una interrupción hardware.

9.4 Resultados

Como sabemos, la estimación del peso a partir de la señal adquirida de la célula de carga, consta de varios pasos. En primer lugar el acondicionamiento de la señal mediante un preproceso que reduce las grandes oscilaciones originales y posibilita la aplicación, como segundo paso, de un algoritmo para la estimación del peso a partir del tramo de señal preprocesada correspondiente a una taza.

Esta sección trata la aplicación del método descrito sobre la señal adquirida para realizar un estudio de la precisión con la que, a partir de éstos, puede obtenerse el peso a grandes velocidades. Durante este estudio, se introducirá un sencillo procesado que,

aplicado tras el preprocesado y antes de la obtención del peso, optimiza la precisión del peso obtenido como resultado. Un último paso aplicado es la *corrección*, obtenida de una calibración precisa de la máquina.

9.4.1 Evaluación de las prestaciones de los algoritmos propuestos

Esta sección describe los resultados obtenidos con el método propuesto. Finalmente se propone la realización de una minuciosa calibración para mantener la precisión dentro del rango de un gramo a velocidades altas.

Diseño del experimento

La evaluación de la precisión del peso que un método proporciona, pasa por la aplicación reiterada de éste sobre diferentes casos de un mismo experimento que, en condiciones controladas, se repite, permitiendo estudiar su repetitividad y precisión.

Para evaluar las prestaciones del método que nos ocupa se ha diseñado el siguiente experimento:

> *Se pone en funcionamiento la calibradora para una velocidad que permanecerá constante hasta la finalización del experimento. Previamente han sido marcadas una de cada 25 tazas hasta un total de seis. En estas tazas, será depositado, vuelta tras vuelta, siempre el mismo peso sobre la misma taza. El depósito del peso sobre la taza se realiza en la zona de entrada de la calibradora, que dista aproximadamente unos cinco metros de la zona de pesado, distancia suficiente como para que la pesa se asiente sobre la taza a cualquier velocidad.*
>
> *Se usa siempre la misma taza con el mismo peso para asegurar que el proceso de pesado de la bola se produzca en las mismas condiciones. El resto de las tazas permanecen vacías y existe una de ellas que se elimina de la cinta de transporte para proporcionar una referencia.*

Durante el experimento se adquiere continuamente a 1KHz la señal de la célula de carga de línea.

Las tazas marcadas deben llevar siempre la pesa correspondiente, y no otra para que no influyan en la estimación del peso la diferencia de pesos que existen entre diferentes tazas (hemos de tener en cuenta que el peso de las tazas es diferente, cuestión que no puede obviarse si se pretende obtener una precisión de ±1gr para cada pieza).

Una vez adquirida la señal resultante de este experimento se estudiarán diferentes tramos, correspondientes a la misma taza a su paso por la célula de carga en diferentes vueltas a la misma velocidad. La selección y solapamiento de los tramos que nos interesan es tratada a través de un procedimiento denominado "troceado y solapado" descrito en la sección 4.3. Una vez seleccionada la información que nos interesa, se procederá a la aplicación del método a los distintos tramos y a su estudio estadístico.

Pasos del método

Los resultados que se aportan en posteriores secciones, se obtienen a partir de la aplicación de los siguientes pasos:

- **Acondicionamiento de la señal.** Ha sido utilizada la deconvolución basada en el modelo ARMA cuya descripción y procedimiento de selección figura en el capítulo 4.

- **Promediado variable.** Se observa experimentalmente que un promediado de bajo orden dependiente del peso, aplicado tras el acondicionamiento de la señal, mejora los resultados. El orden, N, se obtiene a partir de una primera aproximación del cálculo del peso, estimada a partir de los 30 puntos centrales de la zona peso. El valor estimado del peso, *ap* en la expresión 9.1, es usado para obtener el orden del promedio que se aplicará. Esta expresión ha sido obtenida experimentalmente.

$$N = \begin{cases} 2 & ap > 100 \\ 3 & 25 < ap \leq 100 \\ 4 & 0 \leq ap \leq 25 \end{cases} \tag{9.3}$$

- **Obtención del peso.** Se aplica el algoritmo de estimación, que proporcionará un valor, en unidades célula, al que se le resta la taza y que posteriormente será convertido a gramos teniendo en cuenta la velocidad de la cinta. Debemos recordar que dicho algoritmo posee una limitación de uso: la variación de peso de una taza (p_n) a la siguiente (p_{n+1}) no puede ser mayor de 50 gramos ($p_{n+1} - p_n > -50gr$).

Resultados experimentales

Las tablas 9.4.1 a 9.4.1 presentan los resultados de la aplicación del método propuesto aplicado a cuatro velocidades distintas de la cinta de arrastre y seis pesos diferentes (322 gr, 197 gr, 182 gr, 149 gr, 49 gr y 37 gr). Como puede observarse, cada

uno de estos casos se ha repetido varias veces para obtener medidas estadísticas como son: media, dispersión y desviación estándar. La desviación estándar proporciona información sobre la precisión con que se puede medir un determinado peso a cada velocidad.

Como puede apreciarse, los resultados se han indicado en unidades del conversor. Como primera aproximación, cabe notar que un gramo equivale aproximadamente a 13,5 unidades célula. Se ha prescindido por el momento el uso de los gramos porque todavía se ha de aplicar un último paso, la corrección basada en la calibración.

Antes de continuar comentando los resultados, cabe notar que aunque han sido obtenidos siguiendo el experimento descrito anteriormente (en el que las tazas anteriores a las tazas marcadas estaban vacías), pueden extrapolarse a cualquier peso en la taza anterior si se cumple la limitación impuesta en la que la diferencia del peso de una taza (p_n) a la siguiente (p_{n+1}) no puede ser mayor de 50 gramos ($p_{n+1} - p_n > -50gr$). Esto es así porque la condición impone que la transición, entre una taza y la siguiente, de la señal preprocesada finalice en la zona no-peso entre ambas tazas, dando lugar a los flancos típicos de la taza en los cuales aplicar el algoritmo de obtención del peso. De esta forma, la obtención del peso de una taza no influye en la taza anterior.

No obstante, si se considera demasiado estricta dicha limitación, ésta se puede relajar disminuyendo levemente la longitud de la zona de pesado.

Peso	322 gr	197 gr	182 gr	149 gr	49 gr	37 gr	Vacía
	17113	15421	15234	14815	13465	13307	12805
	17188	15433	15233	14811	13481	13308	12811
	17218	15437	15240	14819	13479	13322	12808
	16963	15430	15227	14807	13464	13310	12807
	17294	15408	15240	14818	13477	13308	12809
	17133	15420	15229	14821	13472	13317	12809
	17133	15410	15236	14826	13474	13307	12805
	17122	15415	15233	14814	13471	13319	12814
s	96	11	5	6	6	6	3
Media	17146	15422	15234	14816	13473	13312	12809
Dispersión	331	29	13	19	17	15	9

Tabla 9.1: Resultados experimentales obtenidos para 20 frutas/segundo.

Si nos fijamos en las tablas 9.4.1 y 9.4.1, la columna correspondiente a 322gr y 20 frutas/segundo posee varias medidas resaltadas, así como su correspondiente media, dispersión y desviación estándar. Lo mismo sucede con una de las medidas de la columna de 322gr correspondiente a la velocidad de 15 frutas/segundo. Esto indica

los casos en que han sido identificadas sobre la zona-peso, señales que no coinciden con el patrón determinista seguido por el resto de tramos (la sección 4.2.3 describe como, en las mismas condiciones, las oscilaciones de la zona correspondientes a varias vueltas de la cinta son similares y siguen el mismo patrón). Esto indica que, a partir de ciertos pesos, la alta velocidad de la cinta de arrastre propicia patrones diferentes que podrían ser debidos al movimiento de la pesa sobre la taza, y que introducen una imprecisión considerablemente mayor que un gramo. De los datos tabulados se deduce que esto ocurre para pesos considerables (322gr) y la probabilidad de su aparición aumenta con la velocidad. Desechando estos falsos patrones puede verse como el peso obtenido a partir de los casos restantes presenta una desviación estándar menor que un gramo (~13,5 unidades del conversor).

Peso	322 gr	197 gr	182 gr	149 gr	49 gr	37 gr	Vacía
	17233	15429	15211	14816	13471	13299	12789
	17220	15453	15248	14814	13467	13303	12794
	17235	15442	15241	14829	13471	13304	12790
	17239	15433	15237	14814	13465	13302	12797
	17173	15437	15235	14812	13469	13296	12793
	17218	15437	15231	14812	13478	13304	12790
	17201	15438	15238	14814	13451	13291	12793
	17208	15433	15243	14813	13464	13293	12793
	17219	15444	15233	14812	13468	13308	12798
	17227	15437	15237	14837	13466	13304	12800
	17222	15439	15241	14814	13466	13308	12795
	17232	15445	15242	14811	13476	13302	12791
	17230	15453	15236	14830	13471	13304	12795
	17239	15439	15239	14828	13467	13323	12794
	17234	15440	15230	14810	13457	13308	12794
	17245	15432	15244	14826	13466	13309	12791
	17215	15435	15234	14810	13469	13308	12796
s	17	7	8	9	6	7	3
Media	17223	15439	15236	14818	13467	13304	12794
Dispersión	72	24	37	27	27	32	11

Tabla 9.2: Resultados experimentales obtenidos para 15 frutas/segundo.

Un análisis de los datos tabulados para cada velocidad, permite concluir que para 6 frutas/segundo la desviación estándar en la obtención del peso entra dentro de 0,25 gramos ($\sigma \approx 3,5$), y típicamente para el resto de velocidades dentro de 0,5 gramos

Peso	322 gr	197 gr	182 gr	149 gr	49 gr	37 gr	Vacía
	17162	15419	15227	14812	13471	13309	12788
	17157	15421	15230	14807	13481	13309	12792
	17163	15427	15233	14817	13464	13309	12792
	17173	15423	15226	14823	13477	13301	12796
	17163	15424	15228	14808	13475	13308	12793
	17163	15426	15243	14814	13469	13298	12793
	17178	15422	15234	14824	13472	13298	12798
	17164	15426	15229	14813	13485	13300	12796
	17157	15430	15237	14820	13483	13311	12792
	17168	15428	15237	14821	13470	13313	12789
	17157	15425	15238	14817	13476	13306	12789
	17173	15427	15232	14823	13487	13312	12794
	17177	15428	15237	14825	13480	13302	12794
	17157	15426	15226	14827	13470	13296	12791
	17168	15426	15235	14810	13477	13308	12794
	17173	15430	15233	14820	13469	13318	12791
	17164	15428	15227	14824	13463	13311	12794
s	7	3	5	6	7	6	3
Media	17166	15426	15232	14818	13475	13306	12793
Dispersión	21	11	17	20	24	22	10

Tabla 9.3: Resultados experimentales obtenidos para 12 frutas/segundo.

($\sigma \approx 7$). Por otro lado, la desviación estándar en la medida del valor del peso de la taza vacía también suele ser menor de 0,25 gramos a cualquier velocidad.

Como se mencionó anteriormente, no se han indicado los valores tabulados en gramos dada la necesidad de un último paso, la conversión de unidades célula a gramos a partir de las curvas de ajuste obtenidas en la calibración del sistema. Nótese en este punto cómo los valores medios obtenidos para un mismo peso varían ligeramente con la velocidad, y se dan casos en los que los intervalos de error de éstos, no encajan para velocidades diferentes. Éstas son las particularidades que la curva de ajuste dependiente de la velocidad corrige.

Calibración de la máquina

La calibración debe compensar la pérdida de linealidad con la curva de transformación de puntos a gramos. Dado que este proceso depende de la velocidad, el sistema debería ser calibrado individualmente para cada una de ellas. En tiempo de ejecución el sistema actualiza los coeficientes de calibrado en función de la velocidad de la máquina para poder realizar la conversión de puntos a gramos en cada caso

Peso	322 gr	197 gr	182 gr	149 gr	49 gr	37 gr	Vacia
	17193	15441	15232	14816	13463	13300	12781
	17192	15440	15238	14821	13463	13299	12786
	17193	15444	15238	14819	13459	13299	12785
	17192	15443	15239	14822	13463	13296	12783
	17193	15442	15233	14816	13460	13296	12784
	17190	15441	15238	14817	13463	13299	12785
	17196	15443	15239	14819	13463	13299	12786
	17194	15442	15237	14825	13457	13297	12787
	17187	15435	15237	14813	13458	13303	12786
	17194	15437	15230	14812	13459	13295	12778
	17186	15436	15229	14814	13458	13297	12780
	17190	15438	15235	14818	13460	13296	12780
	17191	15438	15234	14813	13460	13298	12780
	17193	15439	15236	14816	13463	13297	12783
	17192	15439	15226	14814	13463	13299	12783
	17189	15439	15240	14817	13460	13298	12783
	17195	15440	15239	14818	13463	13293	12783
	17195	15444	15237	14815	13461	13302	12784
	17197	15437	15239	14816	13465	13296	12785
	17188	15444	15238	14811	13461	13304	12784
s	3	3	4	3	2	3	2
Media	17192	15440	15236	14817	13461	13298	12783
Dispersión	11	9	14	14	8	11	9

Tabla 9.4: Resultados experimentales obtenidos para 6 frutas/segundo.

particular.

Como ejemplo, a continuación se aplica una calibración del rango de 0 a 200g mediante un polinomio de orden 2 para cada una de las velocidades consideradas. En este punto es necesario usar los valores exactos de los pesos patrones, por lo que se ha utilizado una báscula de alta precisión para medirlos. Los valores han sido tomados con una cifra decimal y son los siguientes:

- Pesa $37g \longrightarrow 37.0g$
- Pesa $49g \longrightarrow 49.0g$
- Pesa $149g \longrightarrow 149.5g$
- Pesa $182g \longrightarrow 181.6g$
- Pesa $197g \longrightarrow 197.0g$

Usando los promedios tabulados en las tablas 9.1 a 9.4, obtenemos la siguientes curvas de calibración:

$$y = 1.0588 \cdot 10^{-6} \cdot x^2 + 0.0452 \cdot x - 753.31 \tag{9.4}$$

$$y = 0.9311 \cdot 10^{-6} \cdot x^2 + 0.0482 \cdot x - 769.12 \tag{9.5}$$

$$y = 1.5124 \cdot 10^{-6} \cdot x^2 + 0.0320 \cdot x - 657.69 \tag{9.6}$$

$$y = 1.0211 \cdot 10^{-6} \cdot x^2 + 0.0453 \cdot x - 746.79 \tag{9.7}$$

La expresión 9.4 corresponde a 20 frutas/segundo, la 9.5 a 15 frutas/segundo, la 9.6 a 12 frutas/segundo y la 9.7 a 6 frutas/segundo. El resultado, y, es el peso en gramos, mientras que el parámetro x es el valor en unidades célula. Nótese que la curva es esencialmente lineal, dado que el coeficiente de segundo es prácticamente nulo.

Si consideramos como precisión del experimento la desviación estándar obtenida en la realización de un experimento con velocidad y peso constantes, podemos representar la transformación del intervalo [$\bar{x}-\sigma$,$\bar{x}+\sigma$] (donde $\bar{x}$ es el valor promedio y σ la desviación estándar, ambas en unidades célula) a gramos T([$\bar{x}-\sigma$,$\bar{x}+\sigma$]). La tabla 9.5 refleja estas transformaciones. En ésta, puede comprobarse en qué casos dicho intervalo encaja en el intervalo $\pm$1g alrededor del valor nominal del peso, v. En tal caso, si T([$\bar{x}-\sigma$,$\bar{x}+\sigma$]) $\in [\ v-1, v+1\ ]\ \forall\ v \in [\ 0g, 250g\ ]$, se cumplen las especificaciones requeridas en cuanto a la precisión de 1g a dicha velocidad.

Velocidad	*197.0*	*181.6*	*149.5*
20	[195.73 , 197.44]	[181.58 , 182.36]	[149.29 , 150.21]
15	[196.53 , 197.60]	[180.87 , 182.09]	[148.95 , 150.31]
12	[196.67 , 197.14]	[181.30 , 182.08]	[149.13 , 150.06]
6	[196.87 , 197.33]	[181.15 , 181.76]	[149.36 , 149.82]
Velocidad	*49.0*	*37.0*	*0*
20	[48.29 , 49.18]	[36.44 , 37.32]	[-0.01 , 0.42]
15	[48.49 , 49.37]	[36.49 , 37.51]	[-0.18 , 0.24]
12	[48.55 , 49.57]	[36.36 , 37.22]	[-0.11 , 0.30]
6	[48.77 , 49.06]	[36.85 , 37.29]	[-0.15 , 0.13]

Tabla 9.5: Transformación a gramos, de orden 2, de los intervalos [$\bar{x}-\sigma$,$\bar{x}-\sigma$].

Vel	*197.0*	*181.6*	*149.5*	*49.0*	*37.0*	*0*
20	[-1.26 , 0.45]	[-0.06 , 0.71]	[-0.20 , 0.71]	[-0.70 , 0.18]	[-0.55 , 0.32]	[-0.01 , 0.42]
15	[-0.47 , 0.60]	[-0.77 , 0.44]	[-0.54 , 0.81]	[-0.51 , 0.36]	[-0.50 , 0.51]	[-0.18 , 0.24]
12	[-0.32 , 0.14]	[-0.34 , 0.43]	[-0.36 , 0.56]	[-0.44 , 0.57]	[-0.60 , 0.23]	[-0.11 , 0.30]
6	[-0.12 , 0.33]	[-0.49 , 0.11]	[-0.13 , 0.32]	[-0.22 , 0.06]	[-0.14 , 0.29]	[-0.15 , 0.13]

Tabla 9.6: Intervalos [$\bar{x}-\sigma$,$\bar{x}-\sigma$] relativos al valor del peso.

De la tabla 9.5 podemos observar que se cumplen las especificaciones de precisión de 1gr alrededor del valor preciso del peso patrón.

El método propuesto proporciona una excelente desviación estándar, o precisión en unidades célula que debería conservarse en la conversión a gramos. Si la transformación viene dada por un polinomio de orden 2, se observa que el intervalo de precisión transformado a gramos queda encajado en un intervalo de $\pm$1gr alrededor del valor medido del peso patrón pero presenta una desviación. Como ejemplo, en el caso de la pesa de 49gr a velocidad de 20 frutos/segundo (tabla 9.6), el intervalo se transforma como muestra la figura 9.13a. Se consigue una precisión de $\pm$1gr, sin embargo presenta un sesgo respecto del valor nominal. Si se evitara dicho sesgo, figura 9.13b, podría aumentarse la precisión con la que se especifica el peso. De aqui la necesidad de un calibrado fino, que envolvería polinomios de mayor orden o la utilización de *look-up tables* (LUT).

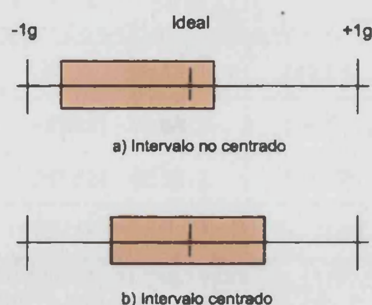


Figura 9.13: a) Intervalo transformado encajado sobre el intervalo de precisión b) Encajado ideal.

Como ejemplo, la tabla 9.7 ilustra un calibrado según un polinomio de tercer orden para una velocidad de 20 frutas/segundo. Por comparación con la fila correspondiente a 20 frutas/segundo en la tabla 9.6, se puede observar que en ésta los intervalos de error están más centrados alrededor del valor nominal. Por ello, un mayor orden contribuye

Vel	197.0	181.6	149.5	49.0	37.0	0
20	[-1.03 , 0.72]	[-0.13 , 0.65]	[-0.57 , 0.34]	[-0.49 , 0.38]	[-0.35 , 0.53]	[-0.24 , 0.19]

Tabla 9.7: Intervalos relativos al valor del peso para transformación de orden 3.

en una mejora de la precisión aunque esto a la vez redunde en un incremento de la complejidad del proceso de calibrado. De todas formas, muchos usuarios no están dispuestos a complicar la calibración a costa de ganar mayor precisión.

9.4.2 Algoritmos implementados en tiempo real

Para contrastar las prestaciones del método obtenido con las del algoritmo implementado en tiempo real en el sistema comercializado, se ha procedido al pesado a 15 frutos por segundo de la pesa de 182gr, 250 veces en la misma taza. Se le indicó al sistema una resolución de 0.25g El resultado se muestra en el histograma de la figura 9.14. Como vemos, $\bar{x}$= 181.66g, que coincide bastante bien con el valor de medido para la pesa patrón de 182g (que es 181.6g), con una desviación estándar de 0.57. Únicamente cinco casos caen fuera del intervalo de ±1g.

El histograma indica que el algoritmo de posicionamiento inteligente de sincronismos es lo suficientemente bueno para la máquina comercial, aunque no sería adecuado para velocidades mayores. Nótese que, en este caso, el sistema comercial usa un esquema de calibración muy sencillo.

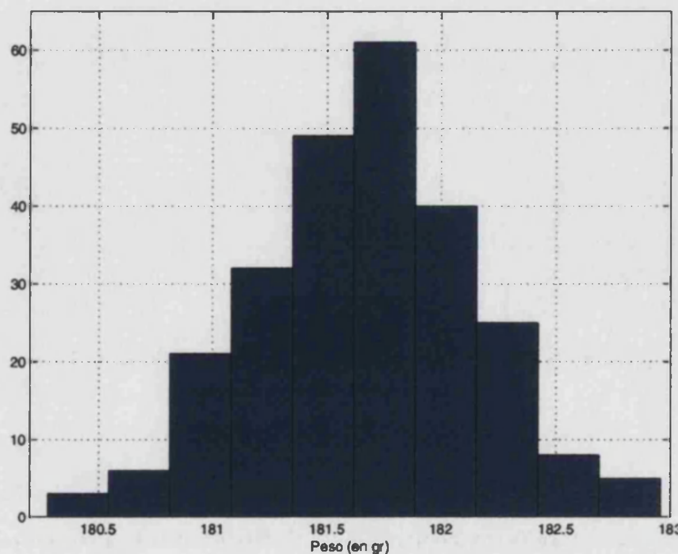


Figura 9.14: Histograma 250 pruebas a 15 frutas/segundo con la pesa de 182gr.

Capítulo 1

Conclusiones y Proyección Futura

1.1 Conclusiones

En el transcurso de la presente Tesis se ha propuesto un método de estimación precisa del peso a altas velocidades y se describe el módulo inteligente, basado en control distribuido, realizado para el sistema de calibración comercial MAXSORTER.

Para el desarrollo del trabajo se adquirieron gran cantidad de registros mediante un prototipo de mayores prestaciones que el sistema comercial. Para ello se diseñó y programó un sistema de adquisición con las siguientes características:

- Basado en sensores de aceleración ADXL150 de Analog Devices y células de carga Artech de 10Lbs.
- Se diseñaron los correspondientes módulos de acondicionamiento.
- Las señales son adquiridas mediante una tarjeta comercial de National Instruments, DAQCard-AI-16XE-50.
- Se ha programado por completo la aplicación de adquisición mediante Visual C++ y la biblioteca NI-DAQ, de National Instruments.
- Al usar la técnica de doble buffer, permite una adquisición continuada con la única limitación de espacio de disco disponible.
- El resultado es una aplicación basada en ventanas que permite controlar todos los parámetros de una adquisición multicanal:

- Canales a adquirir.
- La frecuencia de muestreo se puede particularizar para cada canal.
- Ganancia y Polaridad para cada canal.

En todos los registros tomados se capturaron cuatro señales. Dos procedentes de células de carga, una la célula de carga de línea y otra fijada al chasis pero que nunca es cargada por las tazas. Las otras dos señales proceden de los acelerómetros, para registrar la vibración.

Las tres señales adicionales a la señal útil, de la célula de carga de línea, se registraron para analizar la posibilidad de eliminar las perturbaciones de la señal de interés teniendo en cuenta las señales de vibración. Para ello los acelerómetros se fijan a partes con diferente movilidad como son el vástago y cuerpo de la célula de línea.

Sin embargo, los diferentes planteamientos, básicamente adaptativos, aplicados para la eliminación de las perturbaciones a partir de los registros de vibración procedentes de los acelerómetros, resultaron infructuosos. A pesar de todo, de dicho estudio se ha derivado un método de generación de sincronismos en tiempo real, a partir de la vibración. Los sincronismos así obtenidos indican los "impactos" de las tazas al entrar en la zona de pesado. Sin embargo, esto solo funciona para bajas velocidades (hasta 9 frutas/segundo) y es equivalente a la función de una fotocélula más el hardware asociado para la generación de interrupciones hardware.

Partiendo de las señales registradas se ha realizado una caracterización y modelización de la señal, obteniendo que:

- La célula testigo mide indirectamente la cantidad de vibración de la máquina, siendo este de banda ancha y la salida de la célula de banda estrecha. Ésta se puede interpretar como un proceso aleatorio de banda estrecha producido por la función de transferencia inherente.

- Se demostró que la perturbación que afectaba las zonas peso no se debe a ruido. La realización de un mismo experimento en las mismas condiciones produce una serie de señales que se correlacionan siguiendo un mismo patrón en las zonas peso. Esto indica que dichas perturbaciones se deben a interacciones deterministas entre la taza y la zona de pesado, como el rozamiento entre taza y Teflon. Aunque aparecen diferencias atribuibles a ruido causado por vibración, movimiento de la pesa sobre la taza, etc.

- Se ha realizado un modelo ARMA del sistema de entrada. Éste, es básicamente un ARMA de orden (2,2). La estimación del orden ha sido realizada a través de la aplicación de diferentes métodos de estimación del orden.

La precisión de una medida posee un inherente carácter estadístico. Por ello, para realizar estudios de precisión no podemos menos que tomar una cierta cantidad mínima de registros. Eso si, en condiciones similares (a saber, la misma taza, velocidad, peso, asegurar la estabilización del peso sobre la taza al paso por la zona de pesado, etc.), para poder abordar el estudio de la precisión en la medida y la validación de los algoritmos implicados en el camino hasta la obtención del peso.

Siguiendo este planteamiento, se necesitan varias vueltas de la cadena de arrastre para obtener el mismo número de tramos de interés. Los tramos de interés son extraídos de la gran cantidad de información que son los registros adquiridos, de forma automática, mediante un método denominado "Troceado". Este algoritmo ha desempeñado un papel importante en la validación de los algoritmos de acondicionamiento y de estimación de peso.

A pesar de los pobres resultados obtenidos con los acelerómetros, se han usado con éxito otros planteamientos de preprocesado (acondicionamiento de señal). Se han probado algoritmos tanto lineales como no lineales, adaptativos. Estos últimos no se encontraron adecuados porque los que mejor acondicionan la señal poseen un inadmisible "efecto memoria".

La selección de un algoritmo de preprocesado, de entre los inicialmente planteados, se ha basado en tres criterios cuantitativos. El preprocesado que mejores resultados dio fue la deconvolución de la señal a partir de la inversa del modelo.

Una vez preprocesada la señal, se aplica el algoritmo de estimación del peso, en forma de dos pasos, una primera estimación burda del peso servirá como parámetro de la estimación final. El método propuesto, sin embargo, posee una limitación: la diferencia del peso de una taza (p_n) a la siguiente (p_{n+1}) no puede ser mayor de 50 gramos ($p_{n+1} - p_n > -50gr$). Esta condición se cumple en las condiciones normales de trabajo, imposibilitando el proceso de retarado dinámico de una taza vacía que sigue inmediatamente a una taza con peso, aunque sí sus consecutivas. Esta limitación es consecuencia directa de la dinámica de la célula de carga, y puede mejorarse modificando la longitud de la zona de pesado. En el prototipo disponible, la plataforma de la célula de carga representa el 90% de la distancia entre tazas; unos 65mm de zona de pesado y 7mm de transición entre tazas. Esta relación puede variarse hasta una longitud de la zona de pesado del 70%. Con esto desaparece la citada limitación a costa de una pérdida en precisión de 0.5g a altas velocidades.

Este método ha sido probado bajo estrictas condiciones y con diferentes pesos, para evaluar el grado de precisión obtenido. A cualquier velocidad la precisión obtenida se encuentra con una desviación estándar de ±1gr. Sin embargo, por debajo de 15 frutas/segundo obtenemos una de ±1/2gr.

Por otra parte, cabe destacar que para altas velocidades se generan ocasionalmente ciertos *falsos patrones* para valores altos de peso (p.ej.322gr) a los que puede asignarse una probabilidad de aparición y en los cuales se obtienen valores dispares. En estos casos la señal de la zona peso no coincide con el patrón determinista seguido por el resto de tramos. Esto apunta a que conforme aumenta el peso, las altas velocidades de la cinta de arrastre propician patrones diferentes que serán debidos al movimiento de la pesa sobre la taza, indicando inestabilidad de la pesa y movimiento sobre la taza. Estos casos aparecen raramente, pero deben ser tenidos en cuenta porque introducen una imprecisión considerablemente mayor que un gramo.

Se ha descrito cómo, para conservar una buena precisión en gramos, es necesario una calibración adecuada de la máquina y cuidar la obtención de una curva de transformación de puntos a gramos que conserve la buena precisión obtenida en unidades célula. Son preferibles conversiones polinómicas de orden elevado para mejorar la precisión con la que se obtiene el peso del fruto, aunque también se podría usar una *Look-Up Table* (LUT).

Aunque este método proporciona mejores resultados que el implementado en tiempo real en la calibradora comercial, este no es el método utilizado porque:

- Los calibradores comercializados actualmente, a diferencia del prototipo, sólo funcionan hasta 15 frutas/segundo.

- Se necesita una calibración precisa para conseguir las ventajas del método planteado. Una buena calibración debe automatizar la toma de muestras para muestrear el plano velocidad-peso. Sin embargo, algunos clientes prefieren simplificar el proceso a costa de perder precisión.

El tema 5 proporciona una descripción de los sistemas de calibración de alta velocidad. Los temas 6 a 8 se centran en el módulo de pesado. Este es un módulo inteligente basado en control distribuido que:

- Se conecta con al control vía bus CAN.

- Posee hasta 10 canales, escalables, de adquisición de señales de célula de carga.

- Está basado en un DSP TI TMS320C26.

- Utiliza memoria no volátil para datos persistentes (taras, parámetros, etc.), memoria EPROM para código de la aplicación y SO, y memoria SRAM de cero estados de espera para acelerar los accesos externos.
- Toda la lógica está encapsulada en una CPLD. Ésta también implementa las comunicaciones entre la placa de comunicaciones y el DSP (mediante buffer intermedio), y realiza el protocolo hardware que posibilita la comunicación del puerto serie del DSP con cualquier conversor, en lugar de la comunicación uno a uno para la que este puerto fue ideado.
- La placa de comunicaciones se encarga de hacer transparente, a la tarjeta, las comunicaciones CAN.

Dada la complejidad de la aplicación, se ha diseñado y programado el microkernel de sistema operativo de tiempo real EMMOS, que:

- Proporciona servicios a la aplicación.
- Permite multiproceso.
- Realiza una planificación expulsiva con derecho preferente basada en prioridades, con ejecución FCFS de los procesos de la misma prioridad.
- Se adapta a las particularidades del sistema, especialmente para hacer transparente al usuario las comunicaciones CAN.
- Posee un tamaño de solo 7KW.
- Se graba en EPROM, junto con la aplicación.
- Reside embebido en el módulo de comunicaciones.
- Facilita mejoras futuras de la aplicación.
- Facilita el desarrollo y depuración de la aplicación.

Dadas las limitaciones que ofrece un microkernel de dimensiones tan reducidas, la compilación de una aplicación sobre EMMOS se realiza de forma cruzada, y puede ser llevada al módulo por dos posibles vías:

- Mediante cable serie cuando se deseen posibilidades de depuración.
- Mediante EPROM en funcionamiento normal.

1.2 Proyecciones Futuras

- Previendo un sistema de calibración que, a nivel comercial, funcione a plena velocidad, 20 frutos/segundo, debería implementarse en tiempo real el método propuesto. En este sentido sería objeto de estudio la simplificación de la calibración del sistema.

- Una posibilidad de mejora del módulo hardware surge de la sustitución del 'C26 por un DSP de coma fija de la familia C2XXX de Texas Instruments que, con unos cambios mínimos, permitiría doblar el número de MIPS para realizar mayor cantidad de funciones de procesado y control.

- Respecto al microkernel se podrían realizar modificaciones para tratar de disminuir la latencia de interrupción del microkernel.

- Para finalizar se podría generalizar el método. Es decir, contactar con un fabricante de célula de carga para desarrollar un nuevo producto en células de carga, que incluya el preprocesado y la digitalización de la señal embebida en la célula de carga y proporcionara conectividad mediante un bus de campo.

Apéndice A

El bus CAN

El bus CAN (*Controller Area Network*) es un bus de tipo multi-maestro definido por el estándar ISO11898 donde cada nodo puede enviar y recibir datos. Esto se consigue haciendo que los nodos estén escuchando todas las transmisiones. No existe forma alguna de enviar un mensaje a un nodo en particular, pues todos ellos atenderán a todas las tramas que circulan por el bus. Sin embargo, el hardware proporciona máscaras de filtrado que permiten que un determinado nodo procese tan sólo los mensajes que le interesan. Según esto, un mensaje CAN transmitido por un nodo cualquiera no contiene la dirección del destinatario. En vez de esto, el contenido del mensaje es etiquetado por un identificador que es único en toda la red. Todos los nodos reciben el mensaje y es el controlador CAN el que mediante un test de aceptación del identificador determina si el mensaje contiene información relevante para el nodo. Si el mensaje es relevante será procesado; en caso contrario será desechado.

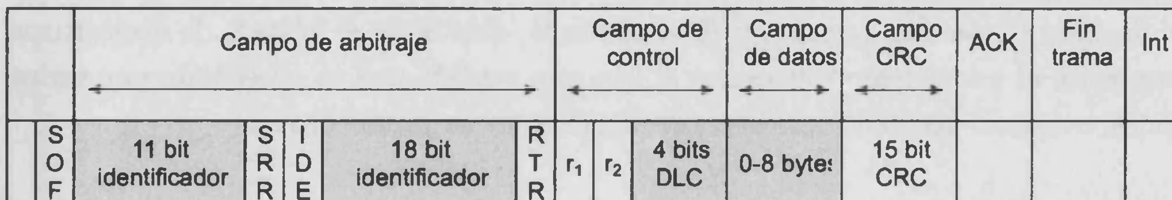


Figura A.1: Formato de los mensajes CAN

La causa por la que es necesario que el identificador de un mensaje sea único radica en el modo en que el bus CAN determina la prioridad de un mensaje. El menor valor numérico de un identificador se corresponde con la mayor prioridad, es decir, se determina la prioridad del mensaje a partir del valor numérico del identificador.

La forma en que compiten las tramas por el control del bus se resuelve mediante arbitraje, según el cual los bits dominantes (estado lógico 0) sobreescriben a los bits recesivos (estado lógico 1). El resultado es el mismo que si el mensaje de prioridad mayor fuera el único que intentara transmitirse.

Las ventajas del arbitraje no destructivo superan los beneficios de multiplexación en el tiempo o del arbitraje destructivo. De esta forma se garantiza que los mensajes de mayor prioridad se hagan con el control del bus, siendo los mensajes de menor prioridad retransmitidos en el próximo ciclo del bus, o en siguientes ciclos si existen mensajes de mayor prioridad aguardando a ser transmitidos. El identificador de cada mensaje debe establecerse durante la fase inicial de diseño del sistema.

El CAN usa la codificación *Non Return Zero* (NRZ) para la transmisión de los datos a través de un par diferencial, generalmente un par trenzado. El uso de la codificación NRZ asegura una mayor robustez de los mensajes al minimizar el número de transmisiones al tiempo que aumenta la resistencia a perturbaciones externas.

El CANbus puede operar en ambientes con condiciones extremas de ruido e interferencias, al tiempo que los mecanismos de chequeo de errores garantizan que las tramas contaminadas por ruido sean detectadas.

El estándar ISO11898 recomienda que los controladores CAN sean diseñados para que la comunicación continúe incluso si:

- Cualquiera de los dos cables del bus se rompe.
- Cualquier cable se cortocircuita a tierra.
- Cualquier cable se cortocircuita a alimentación.

Dadas las características del bus, un sistema que haga uso de él, es fácilmente escalable, ya que es posible añadir nuevos nodos sin necesidad de realizar modificaciones en la estructura o el software existente. Además, disminuye el número de dispositivos necesarios al poder transmitirse por el bus una medida que es necesitada por varios nodos, evitando que cada uno necesite disponer de su propio sensor.

A.1 Sistema de Control basado en módulos de comunicación

Un sistema de Control basado en módulos de comunicaciones, véase capítulo 6, es capaz de soportar hasta 110 nodos distintos, identificados unívocamente en el sistema a partir de un switch interno. La longitud del bus, por otra parte, puede alcanzar hasta

los 1000 metros (sin ningún tipo de dispositivos adicionales), aunque esta longitud se puede duplicar con el uso de repetidores o bridges.

Según la longitud del bus, la velocidad de transmisión varía, de modo que a mayor longitud, menor velocidad de transmisión. Según las especificaciones físicas de CAN, se puede establecer la siguiente tabla.

LONGITUD DEL BUS	VELOCIDAD EN BIT/S	Tiempo máximo de transmisión (para una longitud máxima de mensaje de 129 bits)
Hasta 25 metros	1Mbit/s	1 μs x 129 bits = 129 μs
Hasta 100 metros	500 Kbit/s	2 μs x 129 bits = 258 μs
Hasta 250 metros	250 Kbit/s	4 μs x 129 bits = 516 μs
Hasta 500 metros	125 Kbit/s	8 μs x 129 bits = 1.032 ms
Hasta 1000 metros	50 Kbit/s	20 μs x 129 bits = 2.580 ms
Hasta 2500 metros	20 Kbit/s	50 μs x 129 bits = 6.450 ms

Tabla 1.1: Velocidad de transmisión del bus CAN.

No obstante, como se puede observar, incluso a las velocidades más bajas de transmisión, el tiempo máximo de transmisión de un mensaje no alcanza la centésima de segundo. Según el formato de mensaje CAN que se ha detallado arriba, es posible enviar en un mismo mensaje hasta 8 bytes de datos. Según esto, si se empleara un sistema de control remoto, un módulo CAN podría enviar, por ejemplo, dos datos de 4 bytes cada 2.580 ms vía CAN. Sólo faltaría añadir el tiempo de proceso de dicho mensaje, tanto del módulo emisor (que no superará en cualquier caso los 2 ó 3 ms.) como del receptor (un módulo de control que puede ser un PC, o un PL). Así cualquier cambio en un módulo de comunicaciones puede ser notificado al módulo de control casi inmediatamente.

A.2 Protocolo de comunicaciones en sistemas de control

Como ya se ha comentado, la especificación CAN (ISO 11898) tan sólo se ocupa de definir la capa física y de enlace del modelo OSI. El modelo OSI aparece como una vía de comunicación entre dispositivos en cualquier red, y ha sido adoptado como un

estándar abierto. En principio, cualquier dispositivo que cumpla dicho estándar puede comunicarse electrónicamente con cualquier otro dispositivo que también adopte dicho estándar.

En el modelo OSI, se asigna a la capa de enlace las tareas de reconocer y procesar el formato de los mensajes. Esta capa se encarga de construir dichos mensajes para que sean enviados a través de la capa física y de decodificar los mensajes recibidos a través de dicha capa. En los controladores CAN, dicha capa es normalmente implementada por el hardware. Dada la complejidad y siguiendo el desarrollo de las redes más comunes, la capa de enlace se subdivide en:

- Una capa de control de enlace lógico **LLC** que se encarga de manejar la transmisión y recepción de los mensajes entre las otras capas superiores del modelo **OSI**.

- Una capa de control de acceso al medio **MAC** que se encarga de codificar y serializar los mensajes para su transmisión y decodificar los mensajes recibidos. La capa **MAC** se encarga asimismo de las tareas de arbitrajes, detección de errores y de acceso a la capa física del modelo OSI.

La capa física especifica las características físicas y eléctricas del bus, siendo el hardware el que se encarga de convertir los datos de un mensaje en señales eléctricas para ser transmitidas, y la tarea inversa para la recepción de tramas.

La especificación del protocolo CAN tan sólo rige la forma en que enviarse los paquetes de datos de una manera segura entre 2 nodos que comparten un mismo medio de comunicación. Sin embargo, no especifica nada acerca de funcionalidades tales como control de flujo, transporte de paquetes más grandes que el tamaño de la trama, asignación de direcciones a un nodo, inicialización del sistema, etc. Dichos aspectos son cubiertos por los **protocolos de alto nivel**, cuya definición viene dada por el modelo de capas **OSI**.

Las funcionalidades que presenta un protocolo de alto nivel son las siguientes:

- Estandariza los procedimientos de inicialización de un sistema, incluyendo la velocidad de transferencia de datos.

- Asigna direcciones a los distintos nodos de un sistema.

- Determina el formato de los mensajes.

- Provee de rutinas de tratamiento de errores.

Muchas aplicaciones de CAN necesitan de unos servicios que sobrepasan la funcionalidad de la especificación CAN en la capa de enlace, como puede ser la transmisión de datos que sobrepasan el tamaño máximo de una trama. Dichos servicios deben ser realizados en la capa de aplicación del modelo OSI.

A.2.1 Especificaciones del protocolo de comunicaciones

A partir de la especificación física de CAN, es posible crear un protocolo de comunicaciones de alto nivel, que permita interpretar adecuadamente la información que circula por el bus. De este modo, se permite el acceso y control remoto de los módulos de comunicaciones, de un modo sencillo y fácil de implementar en cualquier lenguaje de programación.

Tipos de mensajes

En el bus, es posible distinguir dos tipos de mensajes.

- **Mensajes de configuración**: Estos mensajes permiten acceder a ciertos parámetros programables en cada módulo. De este modo es posible establecer su comportamiento no sólo a nivel de entradas / salidas, también a nivel de acceso al bus.

- **Mensajes de estado**: Estos mensajes permiten al módulo enviar el estado de sus entradas/salidas digitales y/o las conversiones analógicas realizadas según los parámetros establecidos por el usuario.

Por otra parte, es posible controlar el **flujo de información en la red**. De este modo, se permite al sistema de control que reciba los mensajes según más le convenga:

- **Mensajes síncronos**: El control remoto debe enviar un mensaje de sincronización que o bien haga actualizar ciertos parámetros a la vez en toda la red o bien solicite el envío de información de cada uno de los módulos según le convenga.

Figura A.2: Proceso de sincronización de las tarjetas.

- **Mensajes asíncronos**: Los mensajes se envían según se haya establecido en la placa de comunicaciones. Por ejemplo, se puede establecer que el mensaje de estado de las entradas digitales se envíe cada vez que cambie alguna de ellas (con lo que la actualización es inmediata en el control), ó bien que se envíe la temperatura actual cada 30 segundos.

Estructura de un mensaje CAN

A partir del formato físico de un mensaje CAN, es posible distinguir tres campos de bits con información relevante:

- **Identificador CAN**. Da prioridad al mensaje en la red: a menor identificador, mayor prioridad. La longitud de dicho identificador puede ser de 11 bits (formato estándar) o 29 bits (formato extendido). En este protocolo se trabajará siempre con identificadores de 29 bits.

- **Campo DLC**. Indice el número de bytes de datos que contiene el mensaje (de 0 a 8).

- **Bytes de datos**. Como ya se ha señalado, un mensajes CAN puede constar de hasta 8 bytes de datos.

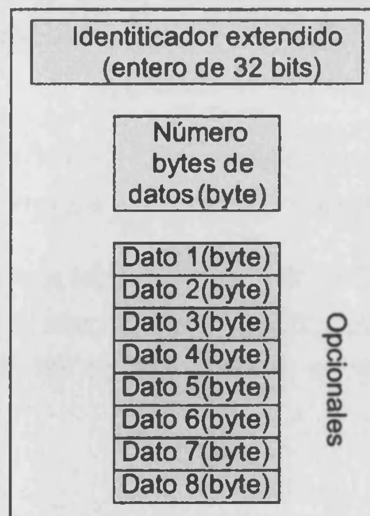


Figura A.3: Estructura de un mensaje CAN.

En el protocolo desarrollado para permitir el acceso a los módulos, los identificadores CAN establecidos se ajustan a un formato que permite realizar un direccionamiento dentro de la red.

- **Orden**. Permite distinguir entre los distintos tipos de mensajes que pueden gestionar los dispositivos en la red.
- **Tipo de tarjeta destino**. Permite distinguir al tipo de tarjeta destinatario de los mensajes. Entre los diferentes tipos de dispositivos podemos encontrar:
  - **TIPO 1: Módulos de comunicaciones**
  - **TIPO 2: Sistemas de Control y/o monitorización (PC con una tarjeta PC-CAN o PLC con conexión RS-232).**
  - **TIPO 3: Puentes RS-232 - CAN.**
  - **TIPO 15: Todos los dispositivos de la red.**
- **Identificador de la tarjeta destino** (Proporcionado por un switch en los módulos y las tarjetas PC-CAN).
- **Identificador de la tarjeta origen.**

Orden	TipoTarjeta Destino	Id Tarjeta Destino	Id Tarjeta Origen
9 bits	4 bits	8 bits	8 bits

Figura A.4: Formato del identificador del mensaje CAN.

De este modo, es posible disponer de tres tipos de direccionamiento:

- **ÚNICO**: Se especifica un tipo y una dirección única para que sólo un dispositivo reciba el mensaje.
- **MULTICAST**: Todos los dispositivos de un mismo tipo reciben el mensaje. Para ello, se debe especificar en el campo *Identificador de Tarjeta Destino* el valor 255 (0FFh), correspondiente a todos los identificadores de la red.
- **BROADCAST**: Todos los dispositivos de la red reciben el mensaje. Para ello se debe especificar en el campo Tipo Tarjeta Destino el valor 15 (0Fh) y en el campo *Identificador de Tarjeta Destino,* el valor 255 (0FFh).

Cualquier dispositivo del sistema, establece las máscaras de aceptación del su controlador CAN correspondiente para admitir estos tres tipos de direccionamiento.

Apéndice B

El puerto serie del TMS320C26

El puerto serie del 'C26 es *full-duplex*, es decir, puede realizar simultáneamente recepción y transmisión, proporcionando comunicación directa con dispositivos serie a través de un mínimo de hardware externo y puede ser usado para comunicación entre procesadores.

Como veremos posteriormente, una característica de este interfaz es el uso de pulsos de sincronización de marco (FSX/FSR), que permiten indicar el inicio del envío de una nueva palabra o marco de bits (figura B.2). Estos pueden ser usados incluso en modos de operación continua, donde la corriente de datos es ininterrumpida pero son ideales para la transmisión serie de paquetes, donde se utiliza el pulso de sincronización como indicador del comienzo del marco de bits. En un modo de operación continuo, el uso de pulsos de sincronización sólo sirve para iniciar la operación, por lo que una vez iniciada, no se requieren pulsos de sincronización de marco posteriores.

Las señales CLKX/CLKR son las señales de reloj utilizadas para la operación de transmisión y recepción respectivamente. Este tipo de enlace no requiere una frecuencia mínima de operación (frecuencia mínima f=0Hz).

Los bits, pines y registros que controlan la operación del puerto serie se listan en la tabla A.1.

El puerto serie usa dos registros mapeados en memoria: el registro de transmisión de datos (DXR), que almacena el dato a ser transmitido por el puerto serie y el registro de recepción de datos (DRR), que almacena el último dato recibido. Ambos registros pueden operar en modo byte (8 bits) o palabra (16 bits) y pueden ser accedidos de la misma forma que cualquier dirección en memoria. Cada registro dispone de un

Bits del puerto Serie/Pines/Registros		TMS320C26
FO	Bit de formato	SI
TXM	Bit de modo de transmisión	SI
FSM	Bit de modo de sincronización de marco	SI
CLKX	Señal de reloj de transmisión	SI
CLKR	Señal de reloj de recepción	SI
DX	Señal de transmisión serie	SI
DR	Señal de recepción serie	SI
FSX	Señal de sincronización de marco de transm.	SI
FSR	Señal de sincronización de marco de recepción	SI
DXR	Registro de transmisión de datos	SI
DRR	Registro de recepción de datos	SI
XSR	Registro de desplazamiento de transmisión	SI
RSR	Registro de desplazamiento de recepción	SI

Tabla B.1: Tabla de bits del puerto serie, pines y registros del 'C26.

reloj externo, un pulso de sincronización de marco, y un registro de desplazamiento asociado, figura B.1. Los registros DXR y DRR están mapeados en las posiciones 0 y 1 en el espacio de memoria de datos. Los registros XSR y RSR son los registros de desplazamiento asociados a la transmisión y recepción y no resultan directamente accesibles por software.

Se usan tres bits de estado en el registro de estado ST1 para controlar la operación del puerto serie: FO, TXM y FSM:

- El bit FO define si los datos son bytes (FO=1) o palabras de 16 bits (FO=0).
- El bit TXM determina si el pulso de sincronización de marco para la operación de transmisión es generado interna o externamente. Si TXM=1, el pin FSX se configura como pin de salida y se produce un pulso de sincronización de marco cada vez que el registro DXR es cargado (sincronizado con el flanco ascendente de CLKX). Si TXM=0, el pin FSX se configura como entrada y el 'C26 entonces espera un pulso de sincronización externa para comenzar la transmisión.
- El bit FSM (modo de sincronización de marco) es usado para determinar si son requeridos los denominados "pulsos de sincronización de marco", que se describirán posteriormente. Cuando FSM=1, los pulsos de sincronización de marco se requieren, y no se requieren cuando FSM=0. Cuando FSM=1 y los pulsos de sincronización de marco se requieren, un pulso FSX causará que XSR sea cargado por DXR, y la transmisión comience. Si un FSX se presenta antes de la transmisión del último bit de la transmisión actual, el XSR será recargado de DXR, abortando la transmisión actual y comenzando una nueva.

B.1 Operaciones de transmisión y recepción.

La transmisión y recepción del puerto serie se realizan separadamente para permitir operaciones de recepción y transmisión independientes. El interfaz del puerto serie es implementado usando los seis pines del puerto serie. La figura B.1 muestra los registros y pines uados en las operaciones de transmisión y recepción.

Los registros XSR y RSR son registros de desplazamiento que están conectados a DXR y DRR respectivamente.

Las señales CLKX y CLKR se requieren durante la operación del puerto serie, y pueden ser detenidas (a un nivel lógico válido) cuando no sean transferidos datos válidos. La señal de reloj CLKX es la encargada de dirigir el ritmo de las operaciones de transmisión, mientras que la señal CLKR es la equivalente para las operaciones de recepción. En los flancos ascendentes de CLKX, el DSP saca un nuevo bit a la línea DX, mientras que la recepción toma un nuevo bit de DR en los flancos descendientes de CLKR. El bit más significativo del dato se transfiere primero.

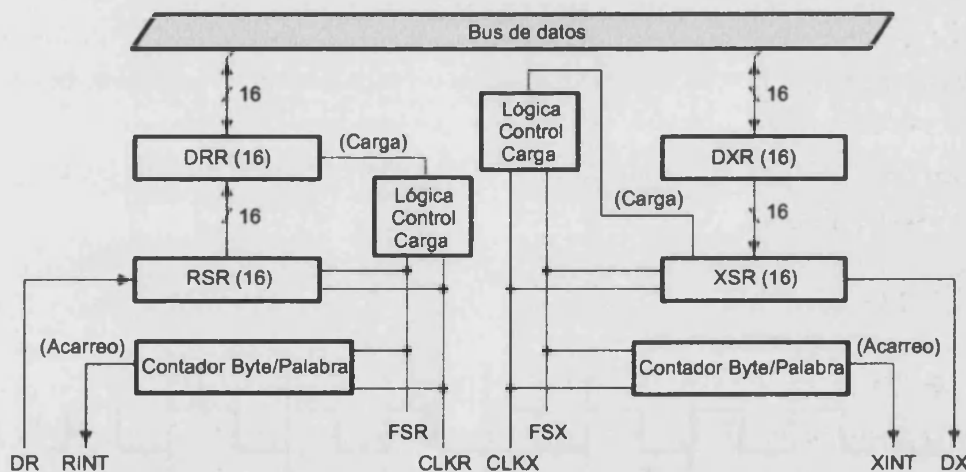


Figura B.1: Diagrama de bloques del puerto serie.

Las transferencias del puerto serie sobre el 'C26 son siempre iniciadas por un pulso de sincronización de marco.

En el cronograma de la figura B.2 se observa la operación de transmisión. Ésta comienza cuando se escribe el dato en el registro de transmisión DXR. El 'C26 comienza la transmisión de datos cuando el pulso de sincronización de marco pasa a nivel bajo mientras CLKX está a nivel alto o pasa a dicho nivel. El dato se carga en el registro de desplazamiento XSR que expondrá un nuevo bit a través de la línea DX con cada flanco ascendente de CLKX. Cuando todos los bits han sido transmitidos, se genera una interrupción interna de transmisión (XINT). Cuando el puerto serie no

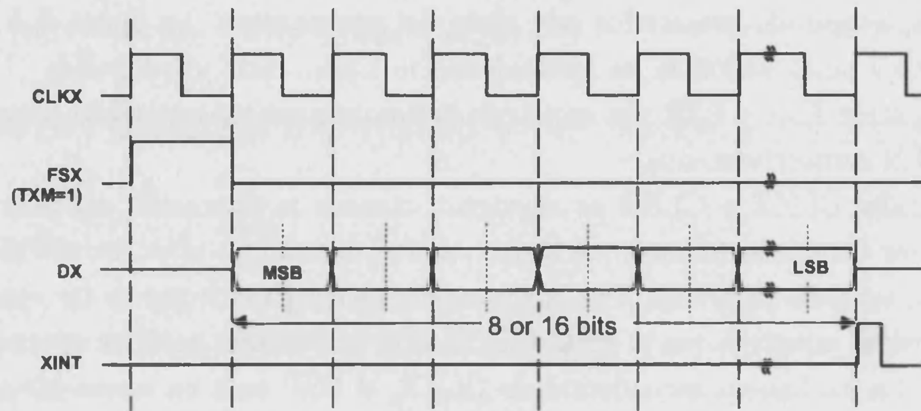


Figura B.2: Cronograma de transmisión del puerto serie del 'C26.

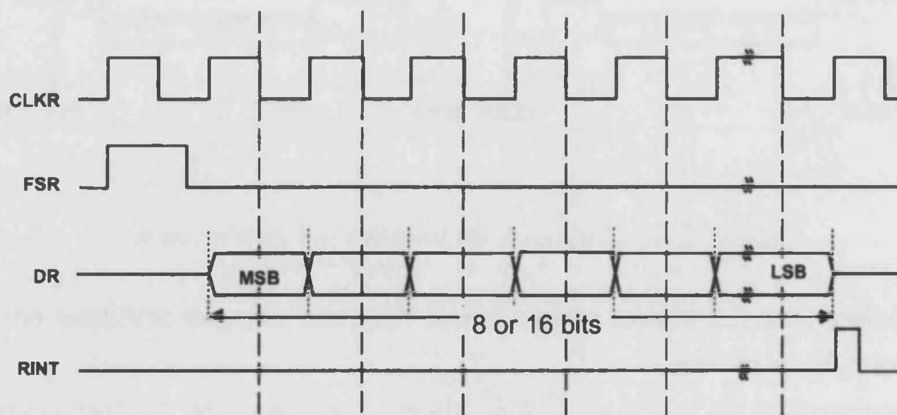


Figura B.3: Cronograma de recepción del puerto serie del 'C26.

está transmitiendo, DX se pone en alta impedancia.

La operación de recepción es similar a la operación de transmisión. El cronograma de recepción se muestra en la figura B.3. La recepción es iniciada por un pulso de sincronización de marco sobre el pin FSR. Tras la transición a bajo de FSR, será tomado un bit en el registro de desplazamiento RSR en cada flanco negativo de CLKR. El primer dato se considera como el bit más significativo. Cuando todos los bits han sido recibidos (según la longitud de palabra que especifica FO), se genera una interrupción interna de recepción (RINT) en el flanco positivo de CLKR, y el contenido de RSR se transfiere a DRR.

Apéndice C

El puerto serie del AD7730

Las funciones del AD7730 son controladas a través de un conjunto de registros internos que pueden ser accedidos a través de un interfaz serie. Sin embargo, la operación de transmisión y recepción no es completamente independiente una de otra, como sucedía en el DSP. El AD7730 necesita conocer el tipo de operación que se realizará a continuación. Tras el encendido o /RESET, el dispositivo espera una escritura en su registro de comunicaciones. El dato escrito en este registro determina si la siguiente operación del componente es de lectura o escritura y también determina a qué registro está dirigida. Por lo tanto, el acceso de escritura a uno de los registros comienza con una escritura en el registro de comunicaciones, seguido de una escritura en el registro seleccionado. La lectura de los registros internos puede tomar una forma simple o continua. Una lectura sencilla de un registro consiste en una escritura en el registro de comunicaciones seguido por la lectura del registro especificado. Para realizar una lectura continua se debe indicar ésta en el registro de comunicaciones junto con la especificación del registro a ser leído. El registro especificado puede entonces ser leído continuamente hasta que ocurra una operación de escritura en el registro de comunicaciones.

El interfaz serie del AD7730 consiste de cinco señales, /CS, SCLK, DIN, DOUT y /RDY. La línea DIN es usada para la transferencia de datos a los registros internos, mientras la línea DOUT es usada para el acceso a los datos de los registros internos. SCLK es la entrada de reloj serie para el dispositivo. Todas las transferencias de datos son gobernadas por esta señal.

C.1 Operación de escritura

La operación de escritura se realiza sobre un registro de desplazamiento de entrada. Cuando se completa una operación de escritura, el dato es transferido al registro especificado. Dicha transferencia no tendrá lugar hasta que el número correcto de bits para el registro especificado haya sido cargado al registro de desplazamiento, siendo éste un número variable (8, 16 ó 24 bits) dependiendo de la longitud de palabra del registro destino. La figura C.1 muestra el cronograma para la operación de escritura al AD7730. Con la entrada POL a nivel lógico alto, el microprocesador debe sacar el dato a DIN durante el flanco descendente de la señal de reloj serie SCLK, ya que el registro de desplazamiento toma un nuevo bit en los flancos ascendentes de ésta. Con POL a estado bajo, debe sacarse el dato a DIN durante el flanco ascendente, ya que el bit será tomado por el registro de desplazamiento durante el flanco descendente de SCLK.

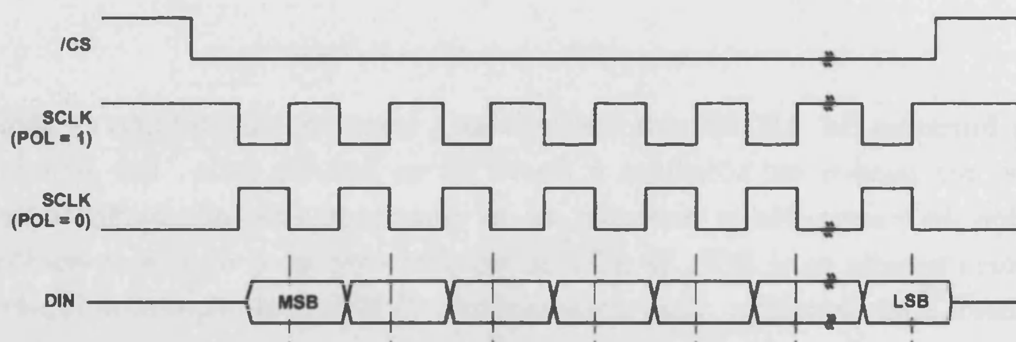


Figura C.1: Cronograma de escritura de datos al AD7730.

La figura C.1 ilustra el uso de la entrada de selección /CS durante la operación de escritura al AD7730. Ésta debería pasar a nivel bajo con anterioridad al primer flanco descendente para POL a nivel alto (ascendente para POL a nivel bajo).

C.2 Operación de lectura.

La operación de lectura se realiza desde un registro de desplazamiento de salida. Al comienzo de la operación de lectura, el dato es transferido del registro especificado al registro de desplazamiento. El desplazamiento de bits a la salida dependerá de la longitud del registro especificado. La figura C.2 muestra un cronograma de una operación de lectura del registro de desplazamiento de salida. Con la entrada POL a nivel alto, el registro de desplazamiento saca un nuevo bit en cada flanco descendente de la señal de reloj serie SCLK, el microprocesador debe tomar el dato de DOUT

en el flanco ascendente de ésta. Con POL a estado bajo, se saca un dato a DOUT durante el flanco ascendente de SCLK, en este caso el bit debe ser tomado por el microprocesador durante el flanco descendente de ésta.

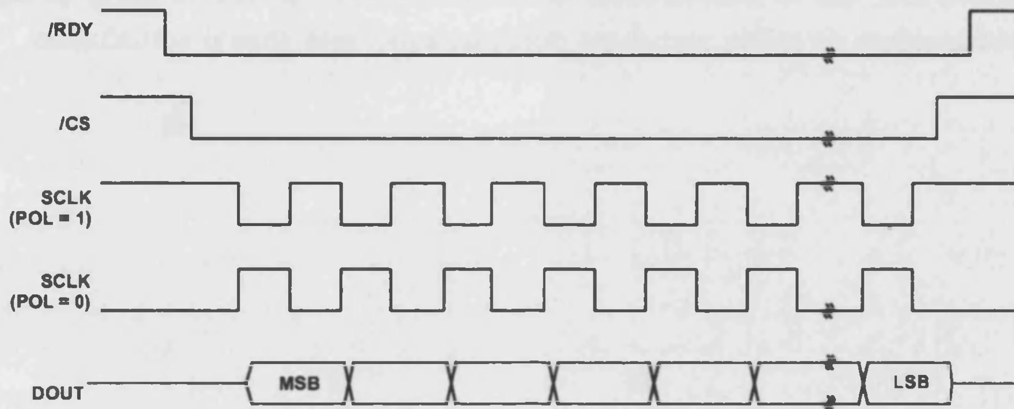


Figura C.2: Cronograma de lectura de datos del AD7730.

La figura C.2 ilustra el uso de la entrada de selección /CS. Ésta debería pasar a nivel bajo con anterioridad al primer flanco descendente para POL a nivel alto (ascendente para POL a nivel bajo).

C.3 Interconexión

En aplicaciones en las que se utilice un microcontrolador, la realización del interfaz, con ayuda de las líneas de los puertos, es simple. Sin embargo, esta flexibilidad redunda en contra de las prestaciones en la velocidad de comunicación.

En aplicaciones DSP las comunicaciones son rápidas, la señal SCLK es generalmente generada por un reloj contínuo. En este caso existen diferentes formas de tratar la señal /CS:

- Por un lado es posible operar en un modo de tres hilos, donde la entrada /CS esté permanentemente a nivel bajo. En este caso, la línea SCLK debería permanecer a alto entre transferencias cuando la entrada POL está a alto (o a estado bajo cuando la entrada POL esté a estado bajo).
- Otra forma será la de operar con SCLK contínua y generar la señal /CS a partir de un pulso de sincronización.

La línea /RDY es usada como una señal de estado para indicar cuando un dato está listo para ser leído por el registro AD7730. /RDY pasa a nivel bajo cuando una nueva palabra de datos está disponible en el registro de datos. Ésta es conmutada a

nivel alto cuando se completa una operación de lectura del registro de datos (también pasa a estado alto antes de una actualización del registro de datos para indicar cuando no debería ser iniciada una lectura del registro de datos). Este mecanismo permite asegurar que la transferencia de datos desde el registro de datos al registro de desplazamiento de salida no ocurre mientras aquel está siendo actualizado.

Apéndice D

El fichero de Comandos

El fichero de comandos contiene información útil que usará el *linker* durante el enlazado de los módulos objeto. Como ya sabemos, los procesos de edición, compilado y enlazado se realizan sobre una plataforma distinta del subsistema de pesado. Este proceso se denomina *compilación cruzada*.

```
MEMORY
{
    PAGE 0:  ROM:     origin= 08000h, length= 07000h
             B0:      origin= 0FA02h, length= 0Ch
    PAGE 1:  RAM:     origin= 03000h, length= 01000h
             NVRAM:   origin= 08000h, length= 08000h
             B1B3:    origin= 00400h, length= 00400h
}
SECTIONS
{
    .text: {  vecs.obj(.text)
              boot.obj(.text)
              kernel.obj(.text) } > ROM PAGE 0
    .cinit: {} > ROM PAGE 0
    .data: {} > RAM PAGE 1
    .bss:  {} > RAM PAGE 1
    nvvar: {} > NVRAM PAGE 1
    mivar: { rts.lib(.bss) } > B1B3 PAGE 1
    virtualvectors 0FA02h : { vecs.obj(virtualvectors) } > B0 PAGE 0
}
```

Código A.1. Contenido del fichero de comandos.

Al crear la aplicación sobre una plataforma distinta, ésta no posee información de la arquitectura y mapas de memoria del sistema destino, por lo que se ha de utilizar el fichero de comandos, que proporciona al enlazador (*linker*) la definición de la estructura de memoria que utilizará el sistema destino e información específica de cómo se ha de realizar el proceso de enlazado para crear la salida ejecutable (sólo sobre el DSP) en formato COFF (*Common Object File Format*).

El código A.1 expone el fichero de comandos utilizado que, como veremos, contiene información sobre el subsistema de pesado y sobre la forma de proceder del proceso de enlazado para que la aplicación trabaje como se le ordena y pueda encajar perfectamente en el sistema destino.

Las directivas que se utilizan para personalizar la aplicación son MEMORY y SECTION. MEMORY define la configuración de memoria del sistema al que se destina y SECTIONS indica cómo serán construidas y gestionadas las secciones.

De la directiva MEMORY se desprende lo siguiente:

- **ROM**: Se denomina así a la zona de memoria SRAM externa de programa entre 08000h que se extiende hasta casi el final del rango de direccionamiento de memoria. Nótese que la EPROM estará en memoria de programa global (en lugar de externa), con idéntico rango de direcciones de memoria

- **B0**: Es una porción de uno de los bloques de memoria interna, configurado como memoria de programa.

- **RAM**: 03000h es la dirección a partir de la que comienza la memoria externa de datos.

- **NVRAM**: define el rango de direcciones de memoria externa que ocupa la NVRAM.

- **B1B3**: son los bloques contiguos de memoria interna B1 a B3, configurados como memoria de programa.

La figura D.1 ilustra el significado de la directiva MEMORY, nótese como se especifican cada uno de los espacios de memoria disponibles, dividiéndolos en páginas según sea memoria de datos o de programa. El restante espacio de memoria no se encuentra definido y por lo tanto el enlazador no podrá hacer uso de él.

Por otra parte, la directiva SECTIONS indica dónde deben situarse cada una de las secciones (TI-C, 1990; TI-Asm, 1990; TI-Sim, 1988; TI-C2x, 1993). En la directiva SECTIONS del fichero de comandos utilizado (ver código A.1) se indica, por ejemplo, que la tabla secundaria de vectores de interrupción debe ser referida a una dirección absoluta del bloque B0. Así, el enlazador sabe donde debe emplazar este trozo de

código, el cual no puede relocalizar. Por otra parte, los datos de las secciones .data, .bss y nvvar se deben situar en memoria externa de datos, los de .cinit en memoria externa de programa y los de .mivar en memoria interna de datos.

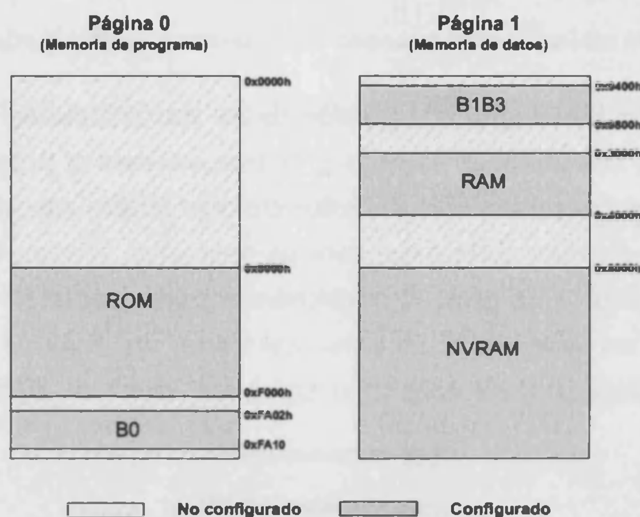


Figura D.1: Representación gráfica de las especificaciones de la directiva MEMORY.

La programación de la aplicación, basada en EMMOS, afectará únicamente a tres secciones:

- La sección .text contendrá todo el código de programa.
- La sección .bss reservará espacio para todas aquellas variables globales y no volátiles usadas introducidas por el programador.
- La sección .cinit incluirá todos aquellos valores que utilizarán las variables globales para su inicialización.

Como vemos, cuando el programador realiza código, se produce el uso de las tres secciones citadas de forma transparente a él y a nivel de compilación y enlazado.

Las estructuras de datos del sistema operativo y de la aplicación han sido asignadas a las secciones .mivar, .nvvar y .data, que han sido codificadas en ensamblador. La codificación en ensamblador posee varias ventajas, entre las cuales destaca:

- Facilidad de disponer las variables y estructuras de datos en el espacio de memoria seleccionado, con lo cual se optimiza la velocidad de ejecución. En otras palabras, esta codificación simplifica al diseñador de sistemas la selección de la zona de memoria donde cada variable o estructura de datos va a situarse. Así las estructuras muy usadas y que necesiten de un rápido acceso por ser críticas

se situarán en memoria interna, mientras que las menos críticas o que puedan ser penalizadas pueden situarse en memoria externa de datos.

- La gestión de estas estructuras a nivel de ensamblador permite que puedan ser accedidas tanto desde ensamblador, como ocurre con aquellos trozos del SO escritos en ensamblador por razones de rapidez y optimización, como desde C.

Cabe destacar que la mencionada gestión de las estructuras del SO y la aplicación *se realiza a nivel del diseñador de sistema y es transparente al programador.* Es decir, el programador de la aplicación sólo necesita conocer cuales son las estructuras de la aplicación y las estructuras públicas del sistema operativo, sección 7.2, y cuáles son los servicios de que dispone. Con esto, *el programador puede utilizar cualquier estructura o variable pública con sólo declarar como cabeceras los ficheros que contienen los prototipos de las llamadas al sistema y las variables, que son "kernel.h" y "data.h"* .

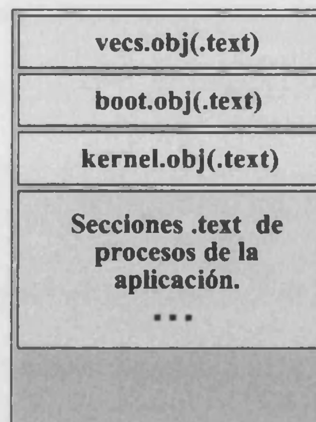


Figura D.2: Ordenación de las secciones durante la construcción.

Volviendo a la descripción del contenido de la directiva SECTIONS, nótese como se establece un orden de construcción para las secciones .text. Esta directiva ordena procesar primero *vecs.obj*, después *boot.obj* seguida de textitkernel.obj (figura D.2). Nótese que todos los procesos escritos por el programador añaden código a la sección .text por lo que estos se construirán como si estuviesen apilados sobre los primeros, aunque no se indique explícitamente en la directiva. Esta orden describe al linker el orden que debe seguir para enlazar los módulos con secciones .text, lo cual es muy importante porque se pretende que la primera sección del fichero ejecutable de salida sea la de la tabla secundaria de vectores de interrupción, tras lo cual debe venir el código del *bootloader* secundario, etc. Estos requerimientos son esenciales para el arranque del sistema (véase Apéndice F).

Las órdenes que invocan a las herramientas de ensamblado y compilación para que se genere el fichero ejecutable de salida (A.OUT)en formato COFF son:

```
> dspcl -c -k -s -al vecs.asm boot.asm kernel.c ini.c sizes.asm [fuentes de la aplicación]

> dsplnk -c vecs.obj boot.obj kernel.obj ini.obj sizes.obj [módulos objeto de la aplicación]
  dir.cmd -m map.out -l rts.lib flib.lib
```

Apéndice E

La Declaración de las Estructuras de Datos

Las estructuras de datos del sistema operativo y de la aplicación han sido asignadas a las secciones .mivar, .nvvar y .data, codificadas en ensamblador. La codificación en ensamblador posee varias ventajas, entre las cuales destaca la flexibilidad de disponer las variables y estructuras de datos en el espacio de memoria seleccionado, con lo cual se optimiza la velocidad de ejecución. En otras palabras, esta codificación simplifica al diseñador de sistemas la selección de la zona de memoria donde cada variable o estructura de datos va a situarse. Así, las estructuras muy usadas y que necesiten de un rápido acceso por ser críticas se situarán en memoria interna, mientras que la menos críticas o que puedan ser penalizadas, en cuanto a tiempo de acceso, pueden situarse en memoria externa de datos.

Otra ventaja de la gestión de estas estructuras en ensamblador es que permite que éstas puedan ser accedidas tanto desde ensamblador, como ocurre con aquellos trozos del SO escritos en ensamblador por razones de rapidez y optimización, como desde C.

La gestión de las estructuras del SO y de la aplicación se realiza a nivel del diseñador de sistema y es transparente al programador. Éste sólo necesita conocer cuáles son las estructuras de la aplicación y las estructuras públicas del sistema operativo, y cuáles son los servicios de que dispone. Con ésto, *el programador puede utilizar cualquier estructura o variable pública con solo declarar como cabeceras los ficheros que contienen los prototipos de las llamadas al sistema y las variables, que son "kernel.h" y "data.h"* .

```
;*****************************************************************************************
; SIZES.ASM las ctes. del sistema y las variables y estructuras de datos del SO y la aplicación
;*****************************************************************************************

    .global _LONGTRANS,_LONGRECEP,_LONGNIVEL1
    .global _LONGNIVEL2,_LONGNIVEL3,_LONGCANAL
    .global _LONGHUELLATRANS,_LONGHUELLARECEP
    .global _LONGPUNTESTRUCT,_IMR,_DXR,_DRR,_tempST1
    .global _INNERSYNCHROS,_PRD,_GREG

    _LONGTRANS .set 200 ;longitud en palabras de la cola de mensajes de transmision
    _LONGRECEP .set 200 ;longitud en palabras de la cola de mensajes de recepcion
    _LONGNIVEL1 .set 100 ;longitud en palabras de la cola de prioridad 1
    _LONGNIVEL2 .set 100 ;longitud en palabras de la cola de prioridad 2
    _LONGNIVEL3 .set 100 ;longitud en palabras de la cola de prioridad 3
    _LONGCANAL .set 19 ;long de buffers canales, sincroniz y tiempo, y de buffer1 y 2.
    _LONGHUELLATRANS .set 13 ;longitud del buffer de vaciado de un mensaje
    _LONGHUELLARECEP .set 13 ;longitud del buffer de vaciado de un mensaje
    _INNERSYNCHROS .set 3 ;Sincronismos a llegar antes de empezar a pesar

    _LONGPUNTESTRUCT .set 19 ;longitud del array de punteros

    ;************ DIRECCIONES ********************

    _IMR .set 4 ;dirección de la máscara de interrupciones
    _DXR .set 1 ;dirección del registro de transmisión del puerto serie
    _DRR .set 0 ;dirección del registro de recepción
    _PRD .set 3 ;dirección del registro de periodo del timer
    ...

;*****************************************************************************
; Se definen las estructuras y otras variables usadas en memoria principal
;*****************************************************************************

    .global _cabzl1, _colal1, _longl1, _listl1
    .global _cabzl2, _colal2, _longl2, _listl2
    .global _cabzl3, _colal3, _longl3, _listl3
    .global _cabzl5, _colal5, _longl5, _listl5, _buffrecep, _wordsaunr, _nextwordr
    .global _cabzl4, _colal4, _longl4, _listl4, _bufftrans, _wordsaunt, _nextwordt
    .global _colachan0,_listchan0,_colachan1,_listchan1
    .global _colachan2,_listchan2,_colachan3,_listchan3
    .global _colachan4,_listchan4,_colachan5,_listchan5
    .global _colachan6,_listchan6,_colachan7,_listchan7
    .global _colachan8,_listchan8,_colachan9,_listchan9
    .global _colasynch,_listsynch,_colatemp,_listtemp
    .global _puntarray,_temp,_flagxserie,_flagrserie
    .global _lostxmens,_modelostxmens,_fintprior
    .global _attendCOMmens,_attendCANmens
```

```
.global _lineassystem,_tazassystem,_origendefault,_needconf
.global _modo,_velocidad
.global _masklineasutiles,_genchanoffset

...

; ________ Sección de MEMORIA EXTERNA ______________

.bss _listl4,_LONGTRANS
.bss _listl5,_LONGRECEP
.bss _listl1,_LONGNIVEL1
.bss _listl2,_LONGNIVEL2
.bss _listl3,_LONGNIVEL3

.bss _cabzl4,1
.bss _cabzl5,1
.bss _cabzl1,1
.bss _cabzl2,1
.bss _cabzl3,1

.bss _colal4,1
.bss _colal5,1
.bss _colal1,1
.bss _colal2,1
.bss _colal3,1

...

; ________ Sección de MEMORIA INTERNA ______________

_wordsaunt .usect "mivar",1c
_nextwordt .usect "mivar",1
_bufftrans .usect "mivar",_LONGHUELLATRANS

_wordsaunr .usect "mivar",1
_nextwordr .usect "mivar",1
_buffrecep .usect "mivar",_LONGHUELLARECEP

_colachan0 .usect "mivar",1
_listchan0 .usect "mivar",_LONGCANAL
_listchan0buf1 .usect "mivar",_LONGCANAL
_listchan0buf2 .usect "mivar",_LONGCANAL

_colachan1 .usect "mivar",1
_listchan1 .usect "mivar",_LONGCANAL
_listchan1buf1 .usect "mivar",_LONGCANAL
_listchan1buf2 .usect "mivar",_LONGCANAL
```

```
_colachan2 .usect "mivar",1
_listchan2 .usect "mivar",_LONGCANAL
_listchan2buf1 .usect "mivar",_LONGCANAL
_listchan2buf2 .usect "mivar",_LONGCANAL

_colachan3 .usect "mivar",1
_listchan3 .usect "mivar",_LONGCANAL
_listchan3buf1 .usect "mivar",_LONGCANAL
_listchan3buf2 .usect "mivar",_LONGCANAL

_colachan4 .usect "mivar",1
_listchan4 .usect "mivar",_LONGCANAL
_listchan4buf1 .usect "mivar",_LONGCANAL
_listchan4buf2 .usect "mivar",_LONGCANAL

_colasynch .usect "mivar",1
_listsynch .usect "mivar",_LONGCANAL

_colatemp .usect "mivar",1
_listtemp .usect "mivar",_LONGCANAL

_genchanoffset .usect "mivar",1

_puntarray .usect "mivar", _LONGPUNTESTRUCT ;gestiona sitio estructura de punteros.
_fintprior .usect "mivar",1 ;Flags de interrupción de las listas de mensajes.
_temp .usect "mivar",1 ;uso temporal de código ensamblador.
_lostxmens .usect "mivar",1
_modelostxmens .usect "mivar",1 ;Modo de actuación ante mensajes perdidos
_lineassystem .usect "mivar",1 ;Variable que contendrá las lineas del sistema en SRAM
_tazassystem .usect "mivar",1 ;Variable que contendrá las tazas del sistema en SRAM
_origendefault .usect "mivar",1 ;Variable que contendrá la distancia al origen en SRAM
_needconf .usect "mivar",1 ;Contendra tras la inicialización si necesita (1) o no (0) config
_modo .usect "mivar",1 ;Variable que contendrá el modo de funcionamiento del sistema
_velocidad .usect "mivar",1 ;Velocidad que contendrá la velocidad del sistema

_attendCOMmens .usect "mivar",1 ;Activacion atención mensajes placa de comunicaciones.
_attendCANmens .usect "mivar",1 ;Activación de la atención de los mensajes CAN

...

; ___________ Sección de NVRAM _________________


.global _NVLineas,_NVTazas,_NVOrigen,_NVTaras
.global _CRCTazas,_CRCLineas,_CRCOrigen
.global _NVmCal,_NVnCal,_NVlCal
.global _CRCmCal,_CRCnCal,_CRClCal
```

```
_NVTaras .usect "nvvar",28000
_NVTazas .usect "nvvar",2 ;Numero de tazas de cada linea.
_NVLineas .usect "nvvar",2 ;Numero de lineas del sistema.
_NVOrigen .usect "nvvar",2 ;Offset del indice que viene con los sincronismos.

_CRCTazas .usect "nvvar",2
_CRCLineas .usect "nvvar",2
_CRCOrigen .usect "nvvar",2

...
```

Código E.1. Gestión de las variables y estructuras en SIZES.ASM

Declaración en ensamblador		**Declaración en C**
Variables		
.global _feedbackmessage	⟷	extern int feedbackmessage;
_feedbackmessage .usect "mivar",1		
Vectores		
.global _ puntarray	⟷	extern void *puntarray[19];
		_puntarray .usect "mivar", 19
Variables		
.global _NVLineas	⟷	struct Bibyte
_NVLineas .usect "nvvar",2		{ int MSB;
		int LSB; }
Vectores de estructuras		
.global _NVOffsetConv	⟷	extern struct NVOffset
_NVOffsetConv .usect "nvvar",30		{ int lowbyte;
		int middlebyte;
		int highbyte; }

Código E.2 Referencia en C de variable declaradas en ensamblador.

Nótese cómo se reserva el espacio para todas las estructuras de datos del SO y de la aplicación. No se aporta todo el código por razones de privacidad y de espacio, dado que el listado completo no aporta más información que el indicado. En éste, se listan las declaraciones de las estructuras usadas, reservando un espacio determinado para cada una de ellas. También puede observarse cómo se distribuyen cada una de éstas

entre los diferentes espacios de memoria de datos dependiendo de las caracterísiticas de optimización de la velocidad, rapidez en el acceso, volatilidad, etc.

Por otra parte, el programa listado sólo declara las variables y reserva espacio para cada una en el espacio de memoria seleccionado por el diseñador de sistema. Pudiera parecer que no se están declarando estructuras de datos, sino variables y vectores. Sin embargo esto es engañoso ya que el espacio que reserva SIZES.ASM será declarado como variable o estructura en C mediante las cabeceras "kernel.h" y "data.h". Para ver un ejemplo de esto, el código E.2 ilustra cómo diferentes tipos de variables declaradas en ensamblador se referencian como externas en las cabeceras. Nótese cómo este poceso posee una gran flexibilidad.

Así, las variables y estructuras de datos son declaradas al compilador de C como tales en las cabeceras y no en SIZES.ASM, aunque este fichero es clave para la declaración de éstas y la reserva de espacio. Nótese que el uso de las cabeceras "kernel.h" y "data.h" hace transparente su uso para el programador. El código E.3 muestra como se la cabecera "kernel.h" declara prototipos de las llamadas del SO. De la misma forma, la cabecera "data.h"declara las variables y estructuras públicas del sistema operativo y de la aplicación.

```
#include "ioports.h"

#define SINMODO 0
#define CONFIGURACION 1
#define CLASIFICACION 2
#define MANTENIMIENTO 3

#define TABLE_BEGIN void GetCaracteristicas(int orden,
  int modo,int *prioridad,int *dir) { switch (orden) {
#define AMODAL_BEGIN
#define TABLE_MIDLE }; switch(modo){
#define MODE0_BEGIN case 0: switch (orden) {
#define CASE case
#define PRIORITY : *prioridad=
#define ADDRESS ;*dir= (int)&
#define CEND ;return;
#define MODE_END default : *dir=0; return;}
#define MODE1_BEGIN case 1: switch (orden) {
#define MODE2_BEGIN case 2: switch (orden) {
#define MODE3_BEGIN case 3: switch (orden) {
#define MODE4_BEGIN case 4: switch (orden) {
#define MODE5_BEGIN case 5: switch (orden) {
#define MODE6_BEGIN case 6: switch (orden) {
#define MODE7_BEGIN case 7: switch (orden) {
#define MODE8_BEGIN case 8: switch (orden) {
```

```
#define MODE9_BEGIN case 9: switch (orden) {
#define MODE10_BEGIN case 10: switch (orden) {
#define TABLE_END default : *dir=0; return;} }

#define VEC_ADQ_IS void InitVecAdq() {
#define VEC_ADQ_END ();}
void GetCaracteristicas(int orden,int modo,int *prioridad,int *dir);
void InitVecAdq();
...
```

Figura E.3. Cabecera "kernel.h".

Sin embargo, éstas no son las únicas variables que el programador puede utilizar: también puede utilizar en su código cualquier tipo de variables locales, que se crearan eventualmente en la pila hasta el final del proceso, por lo que éstas son de carácter volátil con un alcance local y después son "destruidas". El programador también podrá reservar variables y estructuras globales y no volátiles, que automáticamente reservarán espacio en la sección .bss (véase Apéndice F).

Apéndice F

La Inicialización del Sistema

Este Apéndice muestra todo el proceso de arranque del sistema, pormenorizando las particularidades de cada uno de los pasos. En primer lugar muestra las posibilidades de carga y arranque del 'C26. Tras lo cual se detalla el proceso de carga primaria y secundaria del programa en el sistema. Finalmente se describe una serie de pasos en que consiste la inicialización del sistema (inicializaciones de estructuras y variables del SO y de la aplicación, tests, interrupciones, programación de conversores, etc.), hasta que se cede el control al proceso nulo.

F.1 Carga y arranque del DSP

El TMS320C26 se diferencia del resto de la familia TI-TMS320C2X de procesadores DSP, en que lleva grabado en la PROM interna un sencillo código que le permite la carga del programa y el arranque, cuando funciona en modo microcontrolador, a través de tres mecanismos diferentes. El resto de la familia debe arrancar en modo microprocesador si no contiene una PROM personalizada programada en fábrica, a excepción del TMS320C28 que posee una EEPROM fácilmente programable.

El arranque del 'C26 permite tres tipos de carga del programa (TI-C2x, 1993):

Modo 1: Carga paralela desde un puerto de E/S.

Es el primer modo de carga que intenta el procesador. El DSP realizará la carga en este modo si la señal /BIO se encuentra en estado bajo tras el *reset.*En ese caso, las señales que se utilizan para realizar el protocolo de carga son /BIO y XF. En este modo, el DSP realiza la carga a través de un interface paralelo conectado a un

procesador *host* via puerto 0 (PA0) paralelo de E/S. Este modo de carga permite la transferencia tanto de palabras de 8 como de 16 bits.

Modo 2: Carga serie a través de puerto RS232.

Si la señal /BIO se encuentra a nivel alto después de 39 ciclos de reloj tras el reset, se realizará un test para verificar si existe una EPROM en memoria global externa. Si este test falla, se intentará indefinidamente, a partir de entonces, realizar una carga serie.

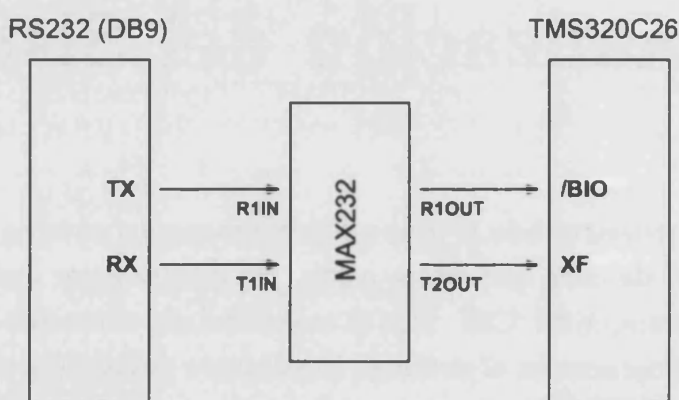


Figura F.1: Comunicación serie.

Durante la carga serie a través de RS232, se usan niveles TTL estándar en los pines /BIO y XF. Para realizar la conversión entre niveles RS232 y TTL, el pin /BIO recibe el dato de un receptor de línea RS232, y el pin XF envia la información vía driver de línea RS232, figura F.1.

Modo 3: Carga procedente de una EPROM en memoria global externa.

Si la señal /BIO se encuentra a nivel alto después de 39 ciclos de reloj tras el reset, se realiza un test para determinar si existe una EPROM en memoria global externa. Si este test falla se intenta a partir de entonces, indefinidamente, una carga serie.

La presencia de una EPROM en memoria global viene determinada por un test de existencia del patrón de bits correspondiente a una instrucción de salto incondicional (B) al comienzo de memoria global externa, en lo que sería la primera palabra de programa codificada en la EPROM. Pero, ¿por qué se busca esta instrucción aquí? ¿Es este un buen test de presencia de una EPROM en memoria global? Para responder a esto, nótese que la tabla de vectores de interrupción codificada en ROM no puede ser modificada por lo que se usa un salto a una tabla de vectores que debe estar en el bloque B0 configurado como programa. Por ello, la primera instrucción del bloque

B0 debería ser un salto incondicional. Por otra parte, el *bootloader* carga el programa inicialmente sobre el bloque B0 (después sobre el B1 y finalmente sobre el B3), véase figura F.2. En consecuencia, la primera palabra codificada en la EPROM debe ser la instrucción de salto incondicional (B), con lo que podemos concluir que el test realizado es excelente para encontrar la presencia de una EPROM en memoria global externa.

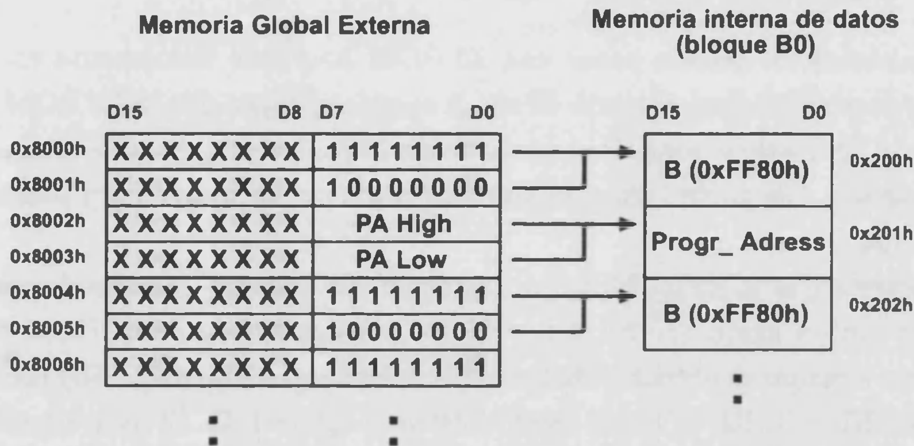


Figura F.2: Proceso de carga del programa a partir de la EPROM.

El test consiste en restar el valor de la instrucción B (0xFF80) del resultado de combinar las dos primeras palabras en la localización 0x8000h, ver figura F.2. Si el resultado es cero, indica que fue encontrada una instrucción de salto. El TMS320C26 realiza este test en memoria global poniendo el registro mapeado en memoria GREG=0x80h. También se realiza el mismo test pero ahora en memoria local poniendo GREG=0 (TI-C2x, 1993). Si se encuentra de nuevo la instrucción de salto (B), se aborta el modo 3 y se pasa al modo 2 de arranque para realizar carga vía serie. Como vemos, este test únicamente se realiza con éxito si el patrón es encontrado en memoria global pero no en memoria local.

Si la EPROM no existe y la señal BR se decodifica, el patrón de bits que recoge de memoria global será aleatorio. Mientras que si BR no se decodifica se recoge el mismo patron de bits, por lo que solo se detecta la presencia de una EPROM en memoria global cuando ésta únicamente está mapeada en memoria global y contenga inicialmente la tabla secundaria de vectores de interrupción.

Si el test de presencia ocurre con éxito se continúa la carga hasta que B0, B1 y B3 son rellenados con datos (1536 palabras).

En cualquiera de los tres modos de carga, el programa comienza a cargarse desde el bloque B0 (0x0200h) en el espacio de memoria interna, y continúa hasta que la longitud especificada por el modo de carga es alcanzada. Entonces los bloques de

memoria interna, que han estado hasta este momento configurados como memoria de datos, son configurados como memoria de programa y se transfiere el control al programa cargado invocando el vector secundario de RESET de la tabla secundaria de vectores de interrupción, sita al comienzo del bloque B0.

F.2 El proceso de carga primaria

El subsistema de pesado posee una EPROM mapeada únicamente en memoria global, por lo que éste será el modo de carga elegido siempre que la EPROM ocupe su zócalo. De lo contrario se elegiría el modo de arranque serie, que también es soportado por el sistema, y que posteriormente veremos como puede usarse para realizar tareas de depuración.

Supongamos que la EPROM ocupa su zócalo. Entonces el proceso de carga desde EPROM se realiza hasta que B0, B1 y B3 son rellenados con datos (1536 palabras). El programa a cargar es obtenido de los bytes menos significativos (LSB) (bits D0-D7) en un orden HI, LO, HI, LO, etc. como ilustra la figura F.2. El byte superior de las lecturas de la EPROM se enmascara, ya que la longitud de palabra de la EPROM es de 8 bits.

En el apéndice D se describe cómo el fichero de comandos ordena al *linker* un determinado orden en la construcción del fichero ejecutable de salida. Ahora estamos en disposición de comprender porqué se construyen primero las secciones .text de los módulos objeto vecs.obj y boot.obj. Como sabemos, el *bootloader*, o cargador primario, comienza a cargar el programa desde el bloque B0 (0x0200h) en el espacio de memoria interna, y continua leyendo hasta 3072 bytes de la EPROM y transformándolos en las 1536 palabras (formando cada dos bytes una palabra de 16 bits) que se cargarán en los bloques de memoria interna B0, B1 y B3, ocupando las direcciones consecutivas 0xFA00h - 0xFFFF de memoria de programa. La especificación de que el primer código que se construya sea la sección .text del módulo vecs.obj (figuras D.1 y D.2 del apéndice D), se realiza porque esta sección contiene la tabla secundaria de vectores de interrupción que debe ser cargada exactamente al principio de B0.

Por otra parte, el fichero de comandos especifica que después de la sección .text de vecs.obj se construya la sección .text de boot.obj. Esto es así porque la carga del bootloader primario está limitada a 1536 palabras y *si se necesita la recuperación adicional de datos, el programa cargado es el que debe llevarla a cabo.* Entonces, el código del *cargador secundario* se construye otra tabla secundaria de vectores de interrupción para que ambas secciones de código puedan ser cargadas en la primera carga, reducida a 1536 palabras. Nótese que si el fichero de salida no se hubiera construido

en este orden, posiblemente se hubieran construido otras secciones de código antes y el cargador secundario no sería cargado dentro de la primera carga, con lo cual sería imposible realizar la carga secundaria y definitiva del programa.

Cuando finaliza el proceso de carga por parte del *bootloader* primario, se invoca el vector de interrupción de RESET de la tabla secundaria, el cual pasa el control al cargador secundario.

F.3 El proceso de carga secundaria

Este cargador secundario es necesario para cargar en memoria principal la totalidad del programa grabado en EPROM. El cargador secundario permite la carga personalizada del código en el lugar adecuado. La carga se realizará en memoria local externa a partir de 0x8000h, ya que la memoria interna se reserva para la pila general y las variables y estructuras más utilizadas, evitando así penalizaciones de tiempo en el acceso a los datos.

```
;****************************************************************************
;         TABLA SECUNDARIA DE VECTORES DE INTERRUPCIÓN (FA00)
;****************************************************************************

; VECTORES DE INTERRUPCIÓN PARA ARRANQUE DESDE EPROM

.global _c_int1,_c_int2,_c_int4,_Boot,_c_int0
.text

B ((_Boot-08000h)+0FA00h)      ; RESET - Punto de entrada al programa
                               ; que utiliza el cargador de EPROM
                               ; primario. Este debe apuntar al cargador
                               ; de EPROM secundario.
EINT                           ; INT0
RET
B _c_int2                      ; INT1
EINT
RET                            ; INT2
B _c_int4                      ; TINT
EINT
RET                            ; RINT
EINT
RET                            ; XINT
```

Código F.1 Tabla secundaria de vectores de interrupción.

Como podemos ver, código F.1, el vector de interrupción secundario de RESET, al que se invoca tras finalizar la carga primaria, apunta al cargador secundario situado en memoria interna. La expresión que calcula la dirección del salto donde reside el cargador secundario tras la carga primaria (nótese que se ha de realizar el cálculo porque el código del cargador se enlazó asumiendo que residía en memoria externa).

Cuando se ejcuta el salto del vector de RESET de la tabla secundaria, se pasa el control al cargador secundario, cargado en el bloque B0. Este cargador, ha sido escrito en ensamblador por razones de flexibilidad.

El cargador secundario, código F.1, realizará la carga del programa al lugar adecuado, después configura todos los bloques de memoria interna como memoria de datos, excepto B0 que debe quedar como memoria de programa para contener la tabla secundaria de vectores de interrupción y en el caso de depuración mantener código residente del *debugger* en el DSP. Posteriormente, pasa el control al punto de entrada de la inicialización del sistema operativo.

```
.global _Boot,_c_int0

GREG .set 5

LEPROM .set 07000h ;Longitud de la EPROM.
EPROM .set 08000h ;Comienza la EPROM en memoria global de datos.
ADRESS .set 08000h ;Comienza el programa en memoria externa local

;Durante el arranque primario que realiza el DSP, no puede cargarse
;en memoria todo el programa, sino una parte: los vectores de interrupción,
;el código de esta función 'boot' y algo más que no interesará posteriormente.
; El DSP pasará todo esto a 0xFA00 y después saltará al vector de reset,
;que envía el flujo del programa a la función _Boot y esta cargará todo el
;programa en EPROM a partir de la dirección ADRESS.

;GREG para acceder a memoria global GREG=0x80;

_Boot: LACK 080h
LDPK 0
SACL GREG

;Anulamos las interrupciones para que se realice todo el proceso
;sin interrupciones hasta que el sistema esteé inicializado.

DINT

;Programa caragador del programa completo en la SRAM de programa.

LRLK AR1,LEPROM
```

```
LRLK AR7,EPROM-1
LRLK AR3,ADRESS
LRLK AR6,078h
LRLK AR5,077h

MAR *,AR7
more: ADRK 2
LALK 0FFh
AND *-
ADD *+,8,AR6
SACL *,AR5
SAR AR3,*
LAC *,AR6
TBLW *,AR5
SAR AR1,*
LAC *,AR3
MAR *+,AR1
BANZ more-08000h+0FA00h,*-,AR7

L2: ZAC
LDPK 0
SACL GREG

;HAY QUE TENER MUCHO CUIDADO EN NO RELOCALIZAR EL BOOT
;PARA QUE TRAS EL CONF, QUEDE ESTE BLOQUE CON EL 'BOOT'
;ES DECIR, EL BOOT SIEMPRE HA DE QUEDAR ÍNTEGRO EN B0

CONF 1

;Salta a la entrada de la inicialización en _c_int0.

B _c_int0,AR1
```

Código F.2 Código del cargador secundario.

F.4 La inicialización del sistema operativo

Cuando el control pasa al punto de entrada de la inicialización del sistema operativo, las interrupciones se encuentran deshabilitadas para que ningún evento interrumpa el proceso de inicialización cuando el sistema aún no está preparado para operar. Los pasos que se dan en la inicialización del sistema son los siguientes:

F.4.1 Inicialización de los punteros a pila

Se configura el puntero a la pila *stack pointer* y el puntero al marco *runtime stack*. Éstos serán registros auxiliares usados, principalmente por el compilador, para gestionar la pila:

- **AR1**: Es el puntero de pila (*stack pointer*). Apunta a la palabra siguiente de la cabeza de la pila.
- **AR0**: Es el puntero de marco (*frame pointer*). Apunta al principio del marco actual. Cada función crea un nuevo marco en la cabeza de la pila, donde son gestionadas las variables locales y temporales.

El entorno C manipula automáticamente estos registros, pero si se pretende escribir en ensamblador rutinas que usen el puntero de pila, se ha de estar seguro de su correcto manejo.

F.4.2 Inicialización de las variables globales

Se inicializan las variables globales mediante la copia de los datos de las tablas de inicialización en .cinit a las variables globales correspondientes, en memoria RAM.

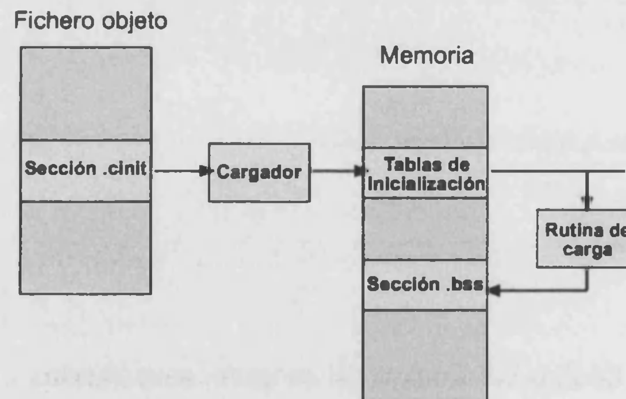


Figura F.3: Inicialización modelo ROM.

Este método de inicialización se denomina *inicialización modelo ROM*. En éste, la sección .cinit se carga en memoria de programa, conteniendo los datos con los que inicializar las variables globales. Para ello, el enlazador define un símbolo especial llamado cinit que apunta al principio de las tablas en memoria. La sección .cinit, junto con el resto de secciones de código, serán grabadas en EPROM. En este punto, la rutina de arranque copia en tiempo de ejecución los datos desde la tabla .cinit

al espacio gestionado para las variables globales correspondientes, cada vez que el sistema comienza.

Así, el programador puede definir cualquier tipo de variables globales inicializadas, que con absoluta transparencia serán inicializadas en este punto de la inicialización del sistema.

F.4.3 Configuración de memoria, inicialización de variables y estructuras de datos

En este punto, se siguen una serie de inicializaciones que son:

- Configuración de los bloques de datos B1-B3 como memoria interna de datos.
- Inicialización de la estructura array de punteros, que utilizará los punteros a las estructuras más usadas, colas y listas, que permite a la gestión de éstas acelerar el acceso a ellas y minimizar el código necesario.
- Se inicializa longitud, cabeza y cola de cada una de las colas de mensajes (cola de trabajos en background, nivel 1, nivel 2, nivel 3, cola de recepción y cola de transmisión).
- Inicialización de las variables requeridas para el funcionamiento de los buffer circulares de los canales de adquisición 0 a 9, sincronismos y tiempos (estos dos últimos no se utilizan en la aplicación).
- Se inicializa la estructura encargada de la recepción de mensajes ("molde de recepción") y las variables relacionadas con éste.
- Se inicializa la estructura encargada de la transmisión de mensajes ("molde de transmisión") y las variables relacionadas.
- Se inicializan a desactivados los flags de atención/interrupción ('fintprior') software de las colas de mensajes:
  - Bit 0: Background
  - Bit 1: Nivel de Prioridad 3
  - Bit 2: Nivel de Prioridad 2
  - Bit 3: Nivel de Prioridad 1
  - Bit 4: Recepción de Mensajes
  - Bit 5: Transmisión de mensajes

- Bits 6 a 15: N/A

- Inicialización del contador de mensaje perdidos y el modo de atención ante mensajes perdidos.

- Inicialización del nivel de trabajo actual a 0 ('workinglevel').

- Inicialización de la recepción y transmisión serie. Configura el pin FSX como pin de entrada, hace necesarios los Frame Syncronization Pulses FSR y FSX (TI-C2x, 1993) e indica al DSP que las transmisiones y recepciones serán de 16 bits.

- Se realiza un proceso de confirmación de los siguientes datos de la NVRAM:
  - Configuración del sistema (Tazas, Líneas y Origen). Afecta al bit 1. Si alguno de los datos no está confirmado, se asignan valores por defecto.
  - Coeficientes de calibración. Afecta al bit 2.
  - Otros coeficientes. Afecta al bit 3.
  - Offset de los canales de adquisición. Afecta al bit 4.
  - Offset en gramos de cada canal. Afecta al bit 5.
  - Ganancia para cada canal. Afecta al bit 6.

  La confirmación consiste en el test de que cada una de las variables citadas han sido inicializadas. Cada variable en NVRAM, excepto los pesos de las tazas, poseen un doblete valor + flag testigo. Este último es un código que se aplica cada vez que una variable es escrita en NVRAM y posibilita que el control conozca si las variables que existen en la memoria no volátil están o no preparadas. Este proceso genera un resultado formado por 6 bits, si el test encuentra que sólo una de las variables que testea no está preparada pone a cero el bit del resultado al cual afecta. Esto puede ocurrir cuando se pone una NVRAM nueva o si algún proceso descontrolado escribe indiscriminadamente en esta memoria.

- Inicialización de otras variables ('modo', 'velocidad', ...).

- Inicialización a desactivado del modo de depuración basado en mensajes.

F.4.4 Habilitación de las interrupciones y programación del *timer*

- Se activan las interrupciones que pueden intervenir.

- Se habilitan las interrupciones.
- Se programa el timer.

F.4.5 Inicialización de las comunicaciones CAN

La inicialización de las comunicaciones se realiza de la siguiente manera:

- Se procede a enviar el mensaje **TARJETA_PESADO_PRESENTE** cada 100 ms. Este mensaje se deja de enviar cuando se recibe un mensaje de respuesta del Control (**CONFIRMACION_CONTROL**), un mensaje de petición (**CONFIRMACION_PRESENCIA**) o tras 10 envíos sin haber recibido ninguno de estos dos mensajes.
- En el envío del mensaje **TARJETA_PESADO_PRESENTE** se enviarán dos bytes de datos. Uno de ellos es el que se obtuvo del resultado del test de disponibilidad de la configuración ('needconf'). El control puede basarse en esto para configurar las variables necesarias o comenzar la aplicación.
- Si se recibiera el mensaje **CONFIRMACIÓN_PRESENCIA** antes de los 10 envíos, se manda un mensaje **VACIAR_COLA_CAN** a la placa de comunicaciones (para evitar que se envíen los mensajes anteriores si todavía los tiene en el buffer), seguido de la respuesta **TARJETA_PESADO_PRESENTE**.
- Si se llegaran a emitir los 10 envíos sin recibir respuesta alguna, se esperará a recibir el mensaje **CONFIRMACION_PRESENCIA** para considerar que se han establecido correctamente las comunicaciones CAN.

F.4.6 Test y programación de los conversores

- Se realiza un test del funcionamiento de los canales de adquisición del sistema.
- Se resetean y programan los conversores de todos los canales válidos.

F.4.7 Paso del control del programa al proceso nulo

Finalmente, tras todas las inicializaciones se pasa el control al proceso nulo, que consiste simplemente en un bucle que ocupará la CPU si ningún otro proceso la utiliza. Éste puede verse como el proceso de menor prioridad de todos, ejecutándose en *background.*

EMMOS aprovecha este bucle para realizar polling de la cola de mensajes de *background* para servirlos si existe alguno de ellos en cola. La planificación dirige la

ejecución de la aplicación independientemente de este proceso. El proceso nulo puede considerarse también como un planificacador a largo plazo cuando existen mensajes en la cola de background, de lo contrario como un bucle de espera.

Apéndice G

Registros, contextos de interrupciones y marcos de función

G.1 Registros del 'C26

A continuación se indica el conjunto de registros que el 'C26 posee, tanto en forma de registros internos como mapeados en memoria, describiendo brevemente su uso.

Nombre del Registro	Dirección	Definición
DRR (15-0)	0	Registro de recepción del puerto serie.
DXR (15-0)	1	Registro de transmisión del puerto serie.
TIM (15-0)	2	Registro temporizador.
PRD (15-0)	3	Registro de periodo.
IMR (5-0)	4	Máscara del registro de interrupciones.
GREG (7-0)	5	Registro de memoria global.

Tabla G.1: Tabla de registros mapeados en memoria.

Registros mapeados en memoria

El 'C26 posee seis registros mapeados en las direcciones 0x0000h a 0x0005h de memoria interna de datos. Estos pueden ser referenciados como posiciones de memoria convencionales, excepto en lo referente a instrucciones de movimiento de bloques.

Registros auxiliares

El TMS320C26 dispone de un fichero de registros que contiene 8 registros auxiliares de 16 bits (AR0-AR7). Estos son utilizados para direccionamiento indirecto o almacenamiento temporal de datos. Un puntero de 3 bits (ARP: Puntero de Registro Auxiliar) direcciona el registro auxiliar actual.

Los registros auxiliares están conectados a una unidad funcional denominada ARAU, que puede autoindexar el AR (registro auxiliar) actual mientras la posición de memoria está siendo direccionada. El indexado puede modificarse por +1, -1 y AR0, permitiendo el acceso a tablas. También puede modificarse según el reverso del orden de los bits de AR0, acelerando así el cálculo de Transformadas Rápidas de Fourier.

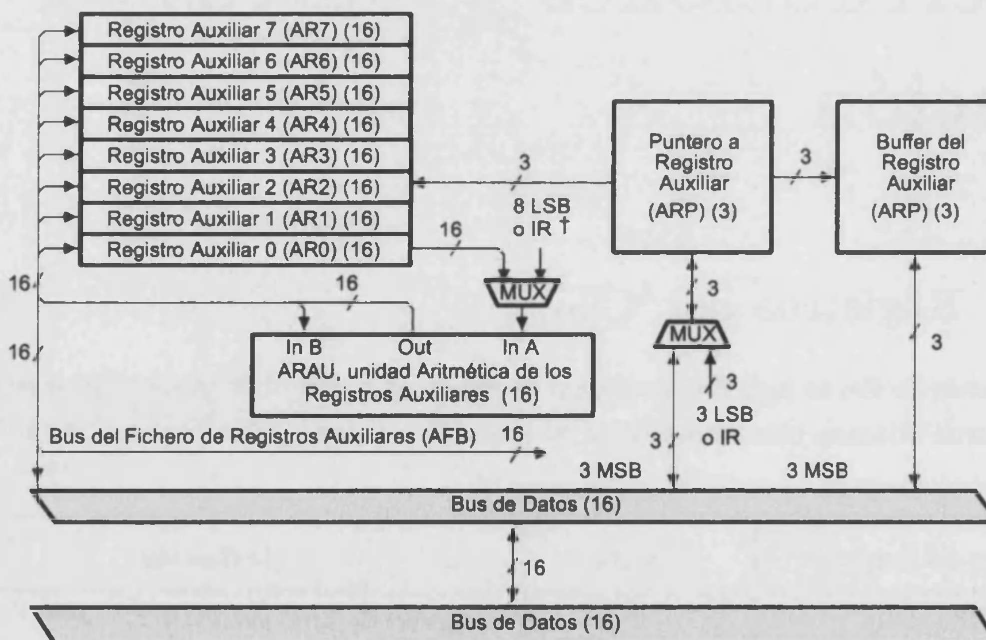


Figura G.1: Fichero de registros auxiliares.

Acumulador

La ALU realiza operaciones aritméticas y lógicas. Produce una salida de 32 bits que es almacenada en el acumulador. El acumulador (ACC) tiene 32 bits y está segmentado en dos partes de 16 bits, ACCL y ACCH. Los desplazadores de la salida del acumulador producen desplazamientos a la izquierda de 0 a 7 bits, que se producen en el proceso de almacenamiento del ACC a memoria, por lo que el contenido de éste permanece inalterado.

Multiplicador: Registros T Y P

El multiplicador hardware implementa productos entre dos números de coma fija en complemento a dos, dando un resultado de 32 bits también en formato de complemento a dos. El registro TR (16 bits) es uno de los operandos de entrada y la salida se almacena en PR (32 bits).

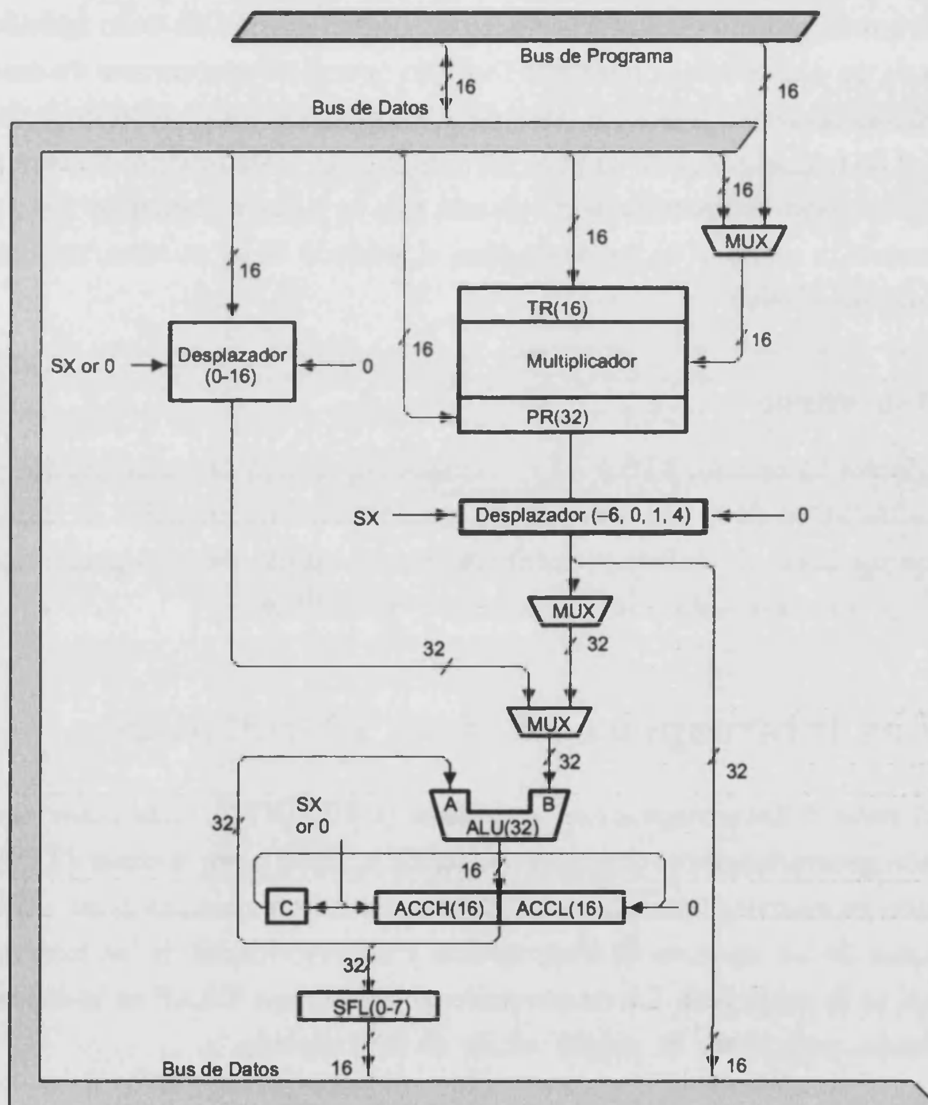


Figura G.2: Unidad Central Aritmético-Lógica (CALU).

Contador del programa

El 'C26 contiene un contador de programa de 16 bits (PC) que referencia direcciones de memoria de programa tanto internas como externas para las instrucciones de búsqueda.

La pila

Existen ocho registros internos que forma una pila y se usa básicamente para almacenamiento del contador del programa durante las interrupciones y subrutinas. La pila es de una anchura de 16 bits y de una profundidad de ocho niveles.

Cuando se apila el contador de programa en la pila, los contenidos de ésta son empujados hacia abajo y el contenido de la parte baja de la pila se pierde. Por lo tanto ocurre una pérdida de datos siempre que ocurren más de ocho apilados sucesivos antes de un desapilado. Lo inverso también ocurre en operaciones de desapilado, cualquier desapilamiento tras siete desapilamientos sucesivos proporciona siempre el mismo valor de la parte baja de la pila. El conjunto de instrucciones incluye posibilidades de apilamiento y desapilamiento de una pila en y desde memoria. Esto permite que sea construida una pila en memoria para el anidado de subrutinas/interrupciones más allá de ocho niveles.

Registros de estado

Dos registros de estado, ST0 y ST1, contienen el estado de varios modos y condiciones. Los registros de estado pueden ser almacenados en memoria de datos y cargados desde memoria de datos, permitiendo que el estado del procesador pueda ser almacenado y restaurado para interrupciones y subrutinas.

G.2 Las interrupciones en el TMS320C26

El 'C25 tiene **3 interrupciones externas** (INT2-INT0). Las **interrupciones internas** son generadas por el puerto serie (RINT y XINT), por el timer (TINT) y por la instrucción de interrupción software (TRAP). Las interrupciones están priorizadas. Las posiciones de los vectores de interrupción y las prioridades de las interrupciones se muestran en la tabla G.2. La interrupción generada por TRAP es la única que no está priorizada, pero posee su propio vector de interrupción.

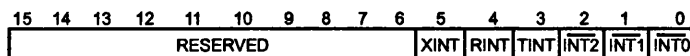


Figura G.3: Registro máscara de interupciones.

Cada dirección de interrupción ocupa dos posiciones para sustituir el vector por una instrucción de salto a una dirección de servicio. El 'C26 dispone de un registro

mapeado, IMR (Interrupt Mask Register) para enmascarar las interrupciones internas y externas.

Interrupción.	**Vector de interrupción.**	**Prioridad**	**Función**
/RS	0h	1 (superior)	Reset .
/INT0	2h	2	Int. externa #0.
/INT1	4h	3	Int. externa #1.
/INT2	6h	4	Int. externa #2.
	8-17h		Reservado.
TINT	18h	5	Interrupción timer.
RINT	1Ah	6	Int. recepción serie.
XINT	1Ch	7 (menor)	Int. transmisión serie.
TRAP	1Eh	-	instrucción TRAP.

Tabla G.2: Vectores de interrupción y prioridades.

Las interrupciones no son aceptadas inmediatamente después de una instrucción EINT, sino que se procesa la siguiente instrucción antes de aceptarlas. Esto asegura el retorno de las rutinas de servivio de la interrupción.

Las interrupciones externas pueden activarse asíncronamente por nivel o flanco. Cuando se produce una interrupción, pasan 2 ciclos hasta que se guarda en la pila la posición correspondiente a la siguiente instrucción. Al romper la '*pipeline*' se generan 3 ciclos *dummy* de ejecución, durante los cuales no se ejecuta instrucción mientras se repone la *pipeline*. La primera posición buscada es la del vector de servicio de interrupción.

G.3 El contexto en las interrupciones

Un aspecto importante de las interrupciones es que las interrupciones puedan suceder de forma transparente al proceso interrumpido. Cuando una interrupción acontece, debe salvarse el contexto durante la entrada al servicio de interrupción: Los registros del DSP que puedan ser modificados, el contador del programa y el estado del procesador. Cuando el servicio de la interrupción finalice, se debe restaurar el contexto salvado para que la CPU pueda seguir ejecutando el proceso interrumpido desde el punto en que se quedó como si nada hubiera pasado.

El almacenamiento del contexto de una interrupción comprende:

- PC (Contador del programa)

- ARP (Puntero al registro auxiliar activo)

- ST1 (Registro de estado)
- ST0 (Registro de estado)
- ACCH y ACCL (Partes baja y alta del acumulador)
- Registro P
- Registro T
- Registros auxiliares 0, 2, 3, 4, 5, 6 y 7

Éste es un almacenamiento de contexto completo, usado por el compilador para todas las interrupciones a través de la rutina I$$SAVE, escrita en ensamblador, totalmente personalizada y sobreescrita a la original, almacenada en rts.lib. Para almacenar el contexto se usa la pila en memoria interna. Como vemos, el contador de programa también se apila. La dirección de retorno se obtiene de la pila de ocho niveles del DSP y se pasa a memoria de datos para evitar el riesgo de que múltiples anidamientos puedan redundar en una pérdida de información en la pila del DSP (ya que la pila del procesador es de una profundidad muy pequeña, únicamente ocho niveles).

El contexto se restaura en orden inverso al restaurado. Esto permite restaurar el sistema exactamente al estado en que comenzó la interrupción. Así, el proceso interrumpido continuará su ejecución sin ser consciente de que ha sido interrumpido.

G.4 Marcos de función

El mecanismo de creación de los marcos de función y el paso de argumentos a una función, son aspectos de muy bajo nivel del tratamiento que el compilador realiza en las llamadas a funciones y para la gestión de variables locales. Además del atractivo que este mecanismo posee en sí mismo, es el fundamento de los acontecimientos que ocurren en la rutina de servicio del temporizador, encargada de la planificación, y clave esencial del microkernel EMMOS.

Generación de marcos locales

Cuando se llama una función, el compilador construye un *marco* (o *registro de activación*) para almacenar información en tiempo de ejecución en la pila. El marco de la función que se está ejecutando se denomina *marco local*. El entorno C usa el marco local para salvar información de la función que realiza la llamada, pasar argumentos y generar variables locales. Cada vez que se llama a una función, se crea

un nuevo marco para generar información acerca de esta función. Sin embargo, el marco de la función que origina la llamada está todavía en la pila, por lo que puede seguir utilizándose su información.

El registro AR1 es el SP (*stack pointer*) o *puntero de pila* y el registro AR0 es el *frame pointer* (FP) o *puntero al marco*. El SP apunta a la siguiente palabra disponible en la pila, mientras que el FP apunta al marco local.

El compilador realiza las siguientes tareas cuando construye el marco local:

- Desapila la dirección de retorno de la llamada de la pila de ocho niveles del procesador y la apila en memoria.
- Apila el contenido de los punteros SP y FP y coloca el nuevo valor de FP y SP.
- Incrementa el SP por el número de palabras que se necesitan para mantener las variables locales, más una palabra al principio del marco para almacenamiento temporal.
- Si la función usa AR6 y AR7, también apila sus contenidos en pila y entonces los carga con las direcciones de variables locales apropiadas.

```
POPD *+           ;Apila la dirección de retorno
SAR AR0, *+       ;Apila FP
SAR AR1, *        ;
LARK AR0, SIZE    ;
LAR AR0,*0+       ; SP += SIZE
SAR AR6, *+       ;Apila AR6
SAR AR7, *+       ;Apila AR7
```

Código G.1 Almacenamiento de contexto y creación del marco de función.

El segmento de código G es una muestra de código que realiza estas tareas, donde SIZE depende del número de variables internas utilizado más una variable temporal.

La figura G.4, refleja los cambios que se producen en la pila y los punteros SP y FP desde momentos antes de la llamada a una función hasta que se ejecuta. En primer lugar, en la figura G.4a vemos las direcciones donde apuntan SP y FP y el marco de la función que realiza la llamada, momentos previos a la llamada. Antes de realizar la llamada a una función, la función que realiza la llamada apila los argumentos que le pasará a ésta, figura G.4b. Tras la llamada, puede comprobarse como se almacena información sobre la función que realiza la llamada y se crea el nuevo marco, el marco local. Este proceso puede ir anidándose sin que ello implique la pérdida de los marcos de las llamadas anteriores.

Hay diversos puntos importantes a tener en cuenta:

- Cuando una función es iniciada, el compilador asume que el ARP apunta al SP (AR1)
- No hay un puntero separado para la lista de argumentos. El puntero al marco puede ser usado con offsets negativos para acceder a argumentos de llamada, y con offset positivos para apuntar a variables locales.
- El puntero al marco (AR0) apunta a una palabra, situada antes de las variables locales. Esta palabra es usada como una variable temporal, esencial para crear funciones C reentrantes, usada entre otras cosas para permitir transferencias registro a registro. Nótese que esta localización de memoria puede ser siempre accedida a través de AR0.
- El compilador usa AR2 para calcular la dirección de variables locales. Generalmente, se le añade el offset de las variables locales se situa en AR2, y se le añade AR0. Éste valor no se preserva entre llamadas de función.

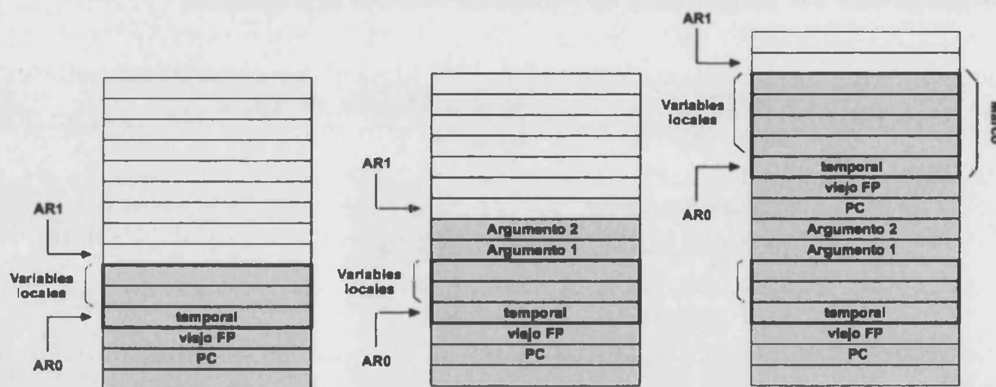


Figura G.4: Cambios en pila para la generación de un marco de función.

Terminación de la función

Cuando una función termina, debe realizar las siguientes tareas para restaurar el entorno de llamada:

- Gestionar los valores de retorno que deben ser pasados a la función llamadora.
- Restaurar AR6 y AR7 si han sido usados.
- Desasignar el espacio usado para variables locales y memoria temporal (segmento de código G.2, el tamaño SIZE representa la cantidad de este espacio).

- Restaurar el anterior puntero de marco.
- Desapilar la dirección de retorno a la pila del 'C26 y realizar el retorno.

El código G.2 es una muestra de código que realiza estas tareas. La observación en sentido inverso de la figura G.4, de G.4c hacia G.4a, refleja los cambios que se producen en la pila y los punteros SP y FP desde la ejecución de la función llamada a la vuelta a la función que realizó la llamada.

```
LAR AR7, *-        ;Restaura AR7
LAR AR6, *-        ;Restaura AR6
SBRK SIZE          ;Desasigna el marco
LAR AR0, *-        ;Restaura FP
PSHD               ;Poner el valo de retorno en la pila
RET                ;Vuelve a la función de llamada
```

Código G.2 Desasignación de marco de función y restauración del contexto.

Cabría notar los siguientes aspectos:

- Los valores se devuelven en el acumulador. Los enteros y punteros son devueltos en los 16 LSBs del acumulador y los valores de punto flotante en el acumulador completo.
- Cuando el acumulador contiene un valor de retorno, no puede ser modificado por el código posterior.
- Los argumentos no son desapilados de la pila por la función llamada, por lo que deben ser desapilados por la función que llamó. Esto permite que se pase cualquier número de argumentos a la función, y que la función no necesite saber cuántos fueron pasados.
- Tras el retorno de la función, el ARP apunta a AR1.
- Las funciones no pueden devolver estructuras.

Apéndice H

La tabla de edición de procesos

La tabla de edición de procesos es el medio por el cual el programador expone al sistema operativo atributos e información sobre los procesos de servicio programados. El sistema operativo considerará inexistentes aquellos procesos que no esten definidos en dicha tabla. Por ello, cada vez que el programador quiera añadir un proceso, además de programarlo debe indicar sobre la tabla citada los atributos del mismo.

En la tabla, el programador ha de indicar en la primera mitad (entre TABLE_BEGIN y TABLE_MIDLE) los procesos que sean amodales, es decir, que se pueden ejecutar cualquiera que sea el modo de funcionamiento del sistema. En la mitad inferior (entre TABLE_MIDLE y TABLE_END) se han de indicar aquellos procesos que el programador quiere que solo se ejecuten si el sistema está en un modo concreto de funcionamiento. Esto se hace incorporando los procesos entre MODEx_BEGIN y MODE_END del modo en que se pretenda hacer posible la ejecución del proceso. Hay diez posibles modos (de MODE0 a MODE9). Sin embargo, la aplicación desarrollada solo utiliza 7.

Considerando el ejemplo del código H.1, observamos que el programador ha indicado que los procesos *ProcessA* y *ProcessB* son amodales, es decir, se podrán ejecutar al ser invocados, sea cual sea el modo de funcionamiento de la aplicación. Sin embargo *GetA* y *MensA* solo se podran ejecutar cuando el sistema se encuentra en modo 0, *GetB* y *StopA* cuando el sistema se encuentra en modo 1, y finalmente *PutC* y *Stop* cuando el sistema se encuentra en modo 2. Si se invocara un proceso existente para su ejecución en un modo de funcionamiento diferente del modo en dicho momento, la orden no es suspendida ni aplazada, simplemente será abortada.

```
TABLE_BEGIN

    CASE 0x30 PRIORITY 1 ADDRESS ProcessA CEND
    CASE 0x40 PRIORITY 2 ADDRESS ProcessB CEND

TABLE_MIDLE

    MODE0_BEGIN
        CASE 0x330 PRIORITY 1 ADDRESS GetA CEND
        CASE 0x340 PRIORITY 2 ADDRESS MensA CEND
    MODE_END
    MODE1_BEGIN
        CASE 0x350 PRIORITY 1 ADDRESS GetB CEND
        CASE 0x360 PRIORITY 3 ADDRESS StopA CEND
    MODE_END
    MODE2_BEGIN
        CASE 0x370 PRIORITY 1 ADDRESS PutC CEND
        CASE 0x380 PRIORITY 0 ADDRESS Stop CEND
    MODE_END

TABLE_END

VEC_ADQ_IS putmens VEC_ADQ_END
```

Código H.1 Ejemplo de uso de la tabla de edición de procesos.

La declaración de los procesos describe la orden que los invoca, la prioridad que les asigna el programador y la dirección donde se encuentra el código del proceso estático en memoria (el modo de funcionamiento en que se puede ejecutar es implícito al lugar de la tabla donde se declara el proceso).

```
CASE (número de orden) PRIORITY (prioridad) ADDRESS (Nombre del proceso que lo sirve) CEND
```

Aunque la dirección del proceso no se especifica directamente, el nombre del proceso es utilizado por el enlazador para obtener esta dirección en su lugar.

La figura H.1 ilustra los niveles de prioridad software de mayor a menor prioridad (de arriba a abajo), y el código numérico que se utiliza para cada prioridad. Nótese como el nivel de más baja prioridad es reservado para los trabajos en *background*, y recibe la prioridad numérica 0.

Nótese que la declaración de procesos de esta forma, proporciona al programador un método flexible de exponer al SO los procesos programados. Esta tabla hace transparente al programador lo que realmente es. Usando las definiciones del código H.2 (contenidas en "kernel.h"), el preprocesador del compilador transforma la tabla de edición de procesos en una función, llamada 'GetCaracteristicas', que devuelve la prioridad y la dirección del proceso que realiza el servicio de la orden indicada, pasandole

la información del modo actual de funcionamiento y el número de orden. Si no se encuentra el proceso de servicio de la citada orden entre la lista de los procesos amodales y los del modo actual, se devuelve una dirección cero, que indica al planificador que el mensaje debe ser desapilado por falta de servicio.

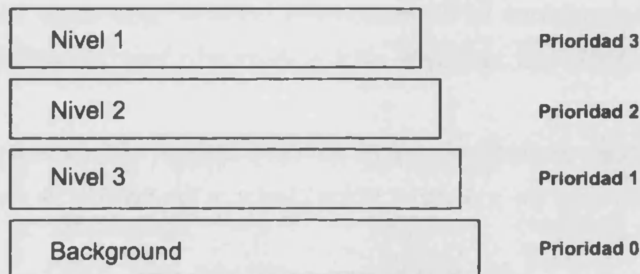


Figura H.1: Jerarquia de prioridades software.

```
#define TABLE_BEGIN void GetCaracteristicas(int orden,
  int modo,int *prioridad,int *dir) { switch (orden) {
#define AMODAL_BEGIN
#define TABLE_MIDLE }; switch(modo){
#define MODE0_BEGIN case 0: switch (orden) {
#define CASE case
#define PRIORITY : *prioridad=
#define ADDRESS ;*dir= (int)&
#define CEND ;return;
#define MODE_END default : *dir=0; return;}
#define MODE1_BEGIN case 1: switch (orden) {
#define MODE2_BEGIN case 2: switch (orden) {
#define MODE3_BEGIN case 3: switch (orden) {
#define MODE4_BEGIN case 4: switch (orden) {
#define MODE5_BEGIN case 5: switch (orden) {
#define MODE6_BEGIN case 6: switch (orden) {
#define MODE7_BEGIN case 7: switch (orden) {
#define MODE8_BEGIN case 8: switch (orden) {
#define MODE9_BEGIN case 9: switch (orden) {
#define MODE10_BEGIN case 10: switch (orden) {
#define TABLE_END default : *dir=0; return;} }

#define VEC_ADQ_IS void InitVecAdq() {
#define VEC_ADQ_END ();}
void GetCaracteristicas(int orden,int modo,int *prioridad,int *dir);
void InitVecAdq();
```

Código H.2 Definiciones en "kernel.h" relativas a la tabla de edición.

El preprocesador transforma la tabla en un servicio privado del SO, que posee el siguiente prototipo de función:

```
void GetCaracteristicas(int orden,int modo,int *prioridad,int *dir);
```

Nótese como a esta función se le ha de pasar el modo de funcionamiento actual del sistema para proporcionar la dirección del proceso que debe servir el mensaje. El modo de funcionamiento del sistema está gobernado por la variable pública 'modo' del SO.

Finalmente, también es posible que el usuario asocie uno de sus procesos al servicio de la adquisición (proceso de máxima prioridad), a través de la sentencia:

VEC_ADQ_IS *Process* VEC_ADQ_END

donde Process es el nombre del proceso de servicio de la interrupción. Se asume que dicha función posee el siguiente prototipo de función:

```
void Process()
```

De la misma forma que ocurría con la tabla de edición de procesos, el preprocesador convierte esta sentencia en una función que será llamada ante cualquier evento de máxima prioridad.

Apéndice I

Tiempos Carcaterísticos de EMMOS

I.1 Medida de tiempos de funciones críticas

A continuación, se listan las medidas de los tiempos de las funciones más críticas del microkernel:

- **void transmite(register int canal, int word)** → 7.4μs
- **int recibe(register int puerto)** → 7.3μs
- **void reset()** → 3.3μs
- **int putmens(int prioridad,int longitud,int *parray)**

  Si la cola está situada en memoria externa:

  ⟶ $10.7 + 5.9 \cdot p$ μs , con p la longitud del mensaje.

  ⟶ 35μs si no puede encolar el mensaje y apila en el nivel 1 el aviso de pérdida.

  ⟶ 11.2μs si no puede encolar mensaje y sólo incrementa el contador 'lostxmens'.

  Si la cola está situada en memoria interna:

  ⟶ $9.6 + 5.5 \cdot p$ μs, con p la longitud del mensaje.

  ⟶ 30.9μs si no puede encolar el mensaje y apila en el nivel 1 el aviso de pérdida.

⟶ $10.1\mu s$ si no puede encolar mensaje y sólo incrementa el contador 'lostxmens'.

Con éxito, máximo de $87.4\mu s$ (mensaje 13 bytes) y mínimo de $22.5\mu s$ (mensaje interno 2 bytes). Puede considerarse la práctica totalidad del tiempo con las interrupciones deshabilitadas.

- **int getmens(int prioridad,int longitud,int *pnewpos)**

  Si la cola está situada en memoria externa:

  ⟶ $14.9 + 6.1 \cdot p$ μs,con p la longitud del mensaje.

  Si la cola está situada en memoria interna:

  ⟶ $13.4 + 5.6 \cdot p$ μs, con p la longitud del mensaje.

  Con éxito, máximo de $88.1\mu s$ (mensaje 12 bytes) y mínimo de $22.5\mu s$ (interno 2 bytes). Puede considerarse la práctica totalidad del tiempo con las interrupciones deshabilitadas.

- **void GetCaracteristicas(int orden,int modo,int *prioridad,int *dir)**

  El tiempo de ejecución depende del lugar de la tabla en el que se encuentre la definición del proceso. El mejor de los casos responde a que las características buscadas correspondan al primer proceso amodal definido:

$$3.7\mu s$$

  En el peor de los caso se ha de contemplar el número de procesos amodales (ca), número de modos (m), y el numero máximo de procesos definidos en un modo (cm). Se ha obtenido una expresión aproximada para el tiempo de ejecución de esta función en el peor de los casos:

$$t = (7.1 + (ca + m + cm) \cdot 0.3)\ \mu s$$

  Con lo que podemos hallar para nuestra aplicación ($ca = 9$, $m = 7$, $cm = 8$) $14.3\mu s$ como cota máxima.

- **void InitVecAdq()**

  $1.7\mu s$ + el tiempo de ejecución del proceso de adquisición.

- **void CANreceive()**

	Byte 1	Resto	Último
Mens. CAN (con error)	11.0	9.4	42.5
Mens. CAN (sin error)	11.0	9.4	$30.2 + 5.9 \cdot n$ (n numero bytes)
Mens. Comunicaciones	11.0	9.4	42.1

Máximo tiempo de apilamiento con éxito 106.9μs. Se puede considerar que esta función deshabilita las interrupciones durante la llamada a putmens, por lo que el tiempo de la zona crítica se puede calcular a partir de éste.

- **void CANtransmit()**

Byte 1	$31.1 + 6.1 \cdot n$ (n numero bytes)
Resto	9.6

- **I$$RESTORE** (restauración del contexto completo)

3.9μs

- **I$$SAVE**

4.2μs

I.2 Medida de tiempos de servicios de interrupción

A continuación se listan los tiempos de los servicios de interrupción:

ISR Recepción. → Duración de CANReceive + 9.6μs

ISR Conversión. → 11.6μs + Tiempo ejecución procesado adquisición.

ISR timer:

Tiempo de ejecución de invocación sin transmisión ni planificación: → 18.4μs

Si se necesita transmisión puede añadirse 1.8μs + tiempo de CANTransmit.

Si se necesita planificación puede añadirse 14.5μs + tiempo de getmens para desapilar el mensaje + llamada a proceso planificado.

Con esto podemos calcular los tiempos máximos de ejecución del servicio del timer, pero no se cuenta la llamada al proceso planificado, sólo el tramo que se ejecuta con las interrupciones deshabilitadas.

- Mínimo: 18.4μs
- Máximo con sólo transmisión: 130μs
- Máximo con sólo planificación: 120μs
- Máximo con transmisión y palnificación: 227.2μs (cuando la transmisión despila el mensaje de la cola de transmisión y envia el primero, y para planificación de un proceso cuyo mensaje de orden ocupa la mayor longitud, que no es un caso realista, porque los mensajes de recepción cuando la máquina debe funcionar en estricto tiempo real son de a lo sumo dos datos)

I.3 Referencias temporales

Latencia de interrupción Máximo intervalo temporal en el que el código correspondiente al sistema se ejecuta con las interrupciones deshabilitadas.

En el peor de los casos 214μs (desde invocación del timer hasta habilitación de las interrupciones antes de la llamada al proceso planificado; nótese que éste es un caso muy al límite), se esperan muchos mínimos en la invocación del timer.

Típicamente no excede de 120μs.

Latencia de planificación Mide el máximo tiempo en que una tarea expulsa a otra de menor prioridad.

El máximo es exactamente 214μs coincidiendo con el servicio del timer, que es el que introduce la latencia. Nótese también lo irreal de este valor por ser conservador en demasía.

Reanudación por interrupción Tiempo entre la ocurrencia de una interrupción que despierta una tarea y el instante en que ésta comienza a ejecutarse.

4.2μs

Apéndice J

Mensajes, parámetros y errores

A continuación se muestran las tablas de mensajes utilizados en la aplicación, y las tablas de parámetros y errores.

MENSAJES CAN DE LA PLACA DE PESADO

Rx/ Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Tx	TARJETA_PESADO_ PRESENTE Orden: 330		0000 Control		0000000 Control	Número de Tarjeta Origen	Tarjeta configurada (valor)	Indice de envío (De 1 a 10)	*Si bit 0 de valor: (=0, no existe conf. De tazas, líneas, y/o origen =1, si existe)* *Si bit 1 de valor: (=0, al menos uno de los coefs. De calibración no existe en NVRam | =1, existen todos)* *Si bit 2 de valor: (=0, al menos uno de los coeficientes de convergencia no existe | =1, existen los dos)* *Si bit 3 de valor: (=0, no existen los coeficientes de offset obtenidos del ajuste de cero | =1, si que existen los coeficientes de offset)*					
Rx	CONFIRMACION_ CONTROL Orden: 305		0101		1111111	00000								
Rx	CONFIRMACION_ PRESENCIA Orden: 301		0101		1111111	00000								
Rx	CONFIGURACION Orden: 101		0101 1111		1111111 1111111	00000								
Rx	LINEAS_SISTEMA Orden: 112		0101 1111		1111111	00000	Número de líneas							

Rx/ Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Rx	TAZAS_SISTEMA Orden: 110		0101 1111		1111111	00000	Número de tazas por línea (hasta 1400)							
Rx	DISTANCIA_ORIGEN_PESADO Orden: 140		0101		0000000	00000	Distancia en tazas al punto de origen (entero con signo, puede ser una distancia negativa)							
Rx	FIN_CONFIGURACION Orden: 199		0101 1111		0000000 1111111	00000								
Rx	INICIO_CLASIFICACION Orden: 415		0101 1111		0000000 1111111	00000								
Rx	VELOCIDAD Orden: 1405		1111		1111111	00000	Número de tazas por segundo.	VELOCIDAD restaura los coeficientes de calibración activos con los valores guardados en NVRam. Si alguno de los que toma de NVRam no estan validados por su CRC, envía un mensaje de errror.						
Rx	INICIO_LINEA Orden: 50		1111		1111111	00000								
Rx	SINCRONISMO_PESADO Orden: 40		0101		1111111	00000	Indice de Taza actual (hasta 1400) (entero sin signo) LSB	MSB						

Rx/ Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Tx	PESO_FRUTA Orden: 1200		0000 Control		0000000 Control	Índice línea PESO 1 (De 1 a 10)	Indice de taza pesada (De 0 a 1399) (1 word)		PESO 1 Peso en gramos de la taza en la linea indicada en el identificador (1 word) TAZA VACÍA = 0		PESO 2 Peso en gramos de la taza en la linea siguiente (1 word) TAZA VACÍA = 0		*[Estado taza en línea 1]* (1 byte)	*[Estado taza en línea 2]* *(1 byte)*
							LSB	MSB	LSB	MSB	LSB	MSB		
Rx	FIN_CLASIFICACION Orden: 405		0101 1111		0000000 1111111	00000								
Tx	ERROR_PERDIDA_ SINC_PESADO Orden: 208		0000 Control		0000000 Control	00000	Número de sincronismos Perdidos							
Tx	ERROR_PERDIDA_ GRAVE_SINC_ PESADO Orden: 204		0000 Control		0000000 Control	00000	Número de sincronismos Perdidos							
Rx	MANTENIMIENTO Orden: 105		0101 1111		0000000 1111111	00000								

Estado: 1 à Peso OK
2 à Tara Vacía,
4 à Todas las taras con cualquier indice de taza (en cualquier linea) no se pesaron porque se perdió el sincronismo.

Rx/ Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen
	11 bits	1	4 bits	1	7 bits	5 bits
Rx	FIN_ MANTENIMIENTO Orden: 195		0101 1111		0000000 1111111	00000
Rx	ORDEN_TARAR Orden: 1300		0101		0000000	00000
Rx	ORDEN_ SOLICITUD_ TARAS Orden: 1306		0101		0000000	00000
Tx	DATOS_TARAS Orden: 1308		0000 Contro l		0000000 Control	Índice línea TARA 1 (De 1 a 10) Si 0 Final del envío de taras
Rx	ORDEN_ INTERRUMPIR_ TARADO Orden: 1305		0101		0000000	00000
Rx	ORDEN_SOLICITUD_ ESTADO_LINEAS Orden: 1307		0101		0000000	00000

DATOS (8 BYTES)							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8

DATOS_TARAS:

Indice de taza tarada (De 0 a 1399) (1 word)		Grado de fiabilidad + TARA 1 Valor digital de la tara de la linea indicada en el identificador.		Grado de fiabilidad + TARA 2 Valor digital de la tara de la **siguiente taza**.		Grado de fiabilidad + TARA 3 Valor digital de la tara de la **siguiente taza**.	
LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

GRADO DE FIABILIDAD: **ntaras** (bit 15-bit14) Numero de veces que se ha tarado la taza en cuestión.
Erbit (bit13-bit12) Error detectado tarando la taza.
TARA: (Bits 11 a 0) Son el redondeo de los 12 bits más significativo del valor en puntos de la tara.

Rx/Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Tx	RESPUESTA_ ESTADO_LINEAS Orden: 1309		0000 Control		0000000 Control	00000	Máscara del estado de las líneas (Funciona/No Funciona) LSBit = Línea 1							
							LSB	MSB						
Rx	OSCILOSCOPIO Orden: 1317		0101		0000000	Índice línea a visualizar (De 1 a 10)	**Muestras que pierde entre dos envíos.**	Muestras (0) Salida filtr (1)						
Tx	DATOS_ OSCILOSCOPIO Orden: 1319		0000 Control		0000000 Control	Índice línea (De 1 a 10) / Número de página	***Marca de llegada de sincronismo de pesado (2 bits)*** + *[Valor digital (14 bits)]*		***Marca de llegada de sincronismo de pesado (2 bits)*** + *[Valor digital (14 bits)]*		***Marca de llegada de sincronismo de pesado (2 bits)*** + *[Valor digital (14 bits)]*		***Marca de llegada de sincronismo de pesado (2 bits)*** + *[Valor digital (14 bits)]*	
Rx	FIN_OSCILOSCOPIO Orden: 1318		0101		0000000	Índice línea a visualizar (De 1 a 10)								
Tx	ERROR_TARJETA_ PESADO Orden: 85=055h		0000 Control		0000000 Control	00000	NUMERO ERROR							
Tx	DEBUG_MESSAGE Orden: 86=056h		0000 Control		0000000 Control	00000	Sentido del mensaje	Byte 1	Byte 2					

Rx/Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Rx	ORDEN_ AJUSTE_CERO Orden: 1310		0101		0000000	00000	Líneas a ajustar LSB=línea 1 LSB	MSB						
Tx	ORDEN_ FIN_AJUSTE_CERO Orden: 1315		0000 Control		0000000 Control		Líneas a ajustar LSB=línea 1 LSB	MSB						
Rx	ENVIO_ PARAMETRO Orden: 1328		0101		0000000	Índice línea (si procede)	Dato LSB	MSB	MMSB	Código del parámetro pedido.	Dato auxiliar 1	Dato auxiliar 2		
Rx	PETICIÓN_ PARAMETRO Orden: 1312		0101		0000000	Índice línea (si procede)	Código del parámetro pedido.	Dato auxiliar 1	Dato auxiliar 2	**Ver Tabla I**				
Tx	ENTREGA_ PARAMETRO Orden: 373		0000 Control		0000000 Control	Índice línea (si procede)	Dato LSB	MSB	MMSB	Código del parámetro pedido.	Dato auxili ar 1	Dato auxiliar 2		
Rx	ORDEN_CALIBRAR_ LINEA Orden: 1320		0101		0000000	00000	Máscara de líneas a Calibrar. LSBit = Línea 1 LSB	MSB						
Tx	PESA_CALIBRE_UNID ADES Orden:1326		0000 Control		0000000 Control	Índice línea PESO 1 (De 1 a 10)	Indice de taza calibrada (De 0 a 1399) (1 word) LSB	MSB	Peso en unidades célula de la taza en la línea indicada en el identificador. LSB	MSB				

Rx/ Tx	Mensaje	r	Tipo de Tarjeta Destino	r	Número de Tarjeta Destino	Índice de Línea / Tarjeta Origen	DATOS (8 BYTES)							
	11 bits	1	4 bits	1	7 bits	5 bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Rx	ORDEN_FIN_ CALIBRADO Orden: 1325		0101		0000000	00000								
Rx	ORDEN_ VERIFICACION Orden: 104		0101		0000000	0	Máscara de lineas a verificar. LSBit = Linea 1							
							LSB	MSB						
Tx	VERIFICACION_ TAZA Orden: 1205		0000 Contro l		0000000 Control	Índice línea PESO 1 (De 1 a 10)	Indice de taza pesada (De 0 a 1399) (1 word)		*Valor digital de la pesada*		*Valor digital de la tara anterior*		*Peso de la pieza en gramos*	
							LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB
Rx	ORDEN_FIN_ VERIFICACION Orden: 194		0101		0000000	00000								
Rx	ORDEN_FIN_ASY_ AJUSTE_CERO Orden: 1311		0101		0000000	00000								

TABLA DE PARÁMETROS

CÓDIGO	FUNCIÓN	LINEA	DATO AUXILIAR 1	DATO AUXILIAR 2
1	**Máscara de líneas útiles:** Máscara de bits que informa de qué conversores están funcionando. (Conversor linea 1 OK bit 0 a 1, Conversor linea 2 OK bit 1 a 1, ... , Conversor linea n OK (bit n-1) a 1.	-	-	-
2	**Modo actual**: 0 (No hay modo), 2 (Clasificación), 3 (Mantenimiento), 1 (Configuración), 4 (Tarado), 5 (Osciloscopio), 6 (Calibrado), 7 (Verificación)	-	-	-
3	**'Numero de tazas' almacenado.**	-	-	-
4	**'Distancia origen' almacenado.**	-	-	-
5	**'Numero de lineas' almacenado.**	-	-	-
6	**'Coeficiente de convergencia en Tarado' almacenado.**	-	-	-
7	**'Coeficiente de convergencia en Clasificación' almacenado.**	-	-	-
8	**'Numero de tazas' activo.**	-	-	-
9	**'Distancia origen' acyivo.**	-	-	-
10	**'Numero de lineas' activo.**	-	-	-
11	**'Coeficiente de convergencia en Tarado' activo.**	-	-	-
12	**'Coeficiente de convergencia en Clasificación' activo.**	-	-	-
13	**'Coeficiente de Calibración' guardados.**	Linea	Velocidad (0 a 20)	Tipo de coeficiente (l=2, m=1, n=0)
14	**'Coeficiente de Calibración' activos.**	Linea	Tipo de coeficiente (l=2, m=1, n=0)	-
15	**Velocidad.**	-	-	-
16	**Ultimo índice llegado** (con el offset sumado).	-	-	-
17	**Eco a los mensajes recibidos por la pesadora** (se produce un eco si es TRUE y no si es FALSE) (Depuración).	-	-	-
18	**Eco del modo actual** (tras recibir cada mensaje se envía uno con el modo actual tras realizar el servicio del mensaje).	-	-	-
19	**Offset del ajuste de cero.** El valor de offset obtenido del ajuste de cero.	Linea	-	-
20	**Offset al nivel de cero.** Se le resta al obtenido por el ajuste de cero para obtener un offset de cero en x puntos.	Linea	-	-
22	**Offset en gramos.** Offset que se le suma al resultado en gramos (Q3).			
23	**Ganancia en gramos.** Ganancia que se le aplica al resultado en gramos (positiva en Q14).			

TABLA DE ERRORES

ERROR	DESCRIPCIÓN
0X10	Número de lineas del sistema no válido (>10 o <1).
0X11	Número de tazas fuera del rango 1 a 1400.
0X14	La línea del osciloscopio pedida no responde (el conversor). (Avisa con el error pero no sale del modo.)
0X15	No se inicia el modo ''TARADO' porque no hay líneas útiles (que repondan o existan) que tarar. (No sale del modo)
0X16	No se inicia el modo 'CLASIFICACIÓN' porque no hay líneas útiles (que repondan o existan) que pesar. (No sale del modo)
0X17	No se inicia el modo 'CALIBRACIÓN' porque no hay líneas útiles (que repondan o existan) que calibrar. (No sale del modo)
0X18	No se inicia el modo 'VERIFICACIÓN'. (No sale del modo)
0X19	No se puede comenzar el modo 'AJUSTE DE CERO' (Sale del modo)
0X20 + i (i□ [0,9])	Se está usando el offset de un conversor sin ajustarlo.
0X2A + i (i□ [0,9])	Se está usando un nivel de offset añadido sin que previamente se haya introducido.
0X34	Velocida > 20 (se satura en velocidad 20 frutos/segundo).
0X35	Se ha pedido un parámetro de código no existente.
0X36	Se escribe un parámetro de código no existente.
0X37	Uno de los parámetros activos tomado de NVRam no está validado por su CRC.
0x38	Enviado número de muestras hacia atrás demasiado elevado.

Apéndice K

Pines del bloque de conectores

Señal	Pin	Pin	Señal
ACH8	34	68	ACH0
ACH1	33	67	AIGND
AIGND	32	66	ACH9
ACH10	31	65	ACH2
ACH3	30	64	AIGND
AIGND	29	63	ACH11
ACH4	28	62	AISENSE
AIGND	27	61	ACH12
ACH13	26	60	ACH5
ACH6	25	59	AIGND
AIGND	24	58	ACH14
ACH15	23	57	ACH7
Reserved	22	56	AIGND
Reserved	21	55	Reserved
Reserved	20	54	Reserved
DIO4	19	53	DGND
DGND	18	52	DIO0
DIO1	17	51	DIO5
DIO6	16	50	DGND
DGND	15	49	DIO2
+5 V	14	48	DIO7
DGND	13	47	DIO3
DGND	12	46	SCANCLK
PFI0/TRIG1	11	45	EXTSTROBE*
PFI1/TRIG2	10	44	DGND
DGND	9	43	PFI2/CONVERT*
+5 V	8	42	PFI3/GPCTR1_SOURCE
DGND	7	41	PFI4/GPCTR1_GATE
PFI5/UPDATE*	6	40	GPCTR1_OUT
PFI6/WFTRIG	5	39	DGND
DGND	4	38	PFI7/STARTSCAN
PFI9/GPCTR0_GATE	3	37	PFI8/GPCTR0_SOURCE
GPCTR0_OUT	2	36	DGND
FREQ_OUT	1	35	DGND

Figura K.1: Asignación de pines del conector 68-AI.

Apéndice L

Diagramas de clases

En la metodología O.M.T. (Object Modelling Technique), el objeto es el ejemplar de una clase que combina datos estructurados y operaciones.

El modelado de los objetos describe la estructura estática del sistema. Para ello se utilizan diagramas de clases como los que a continuación se presentan, para realizar la descripción de los objetos. Los diagramas describen cada clase mediante un listado de los atributos y operaciones de ésta.

A continuación se presentan los diagramas de clase de los objetos que intervienen en la aplicación de la adquisición, descrita en el capítulo 3.

Clase CMainFrame (: CFrameWnd)
□ *Atributos:* **CSplitterWnd m_wndSplitter;** **CSplitterWnd m_wndSplitter2;** **CSplitterWnd m_wndSplitter3;** Estos objetos permiten la gestión del uso del área del cliente de la ventana de aplicación en varias partes independientes. **CToolBar m_wndToolBar;** La barra de herramientas.
□ *Operaciones* **CmainFrame();** Constructor. **virtual ~CMainFrame();** Destructor. **virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext);** Se invoca en el momento de la creación del área del cliente. Este método crea una partición estática de dicha área según la definición del interfaz de la sección 3.4. Cada una de dichas partes es asociada a los objetos *CsplitterWnd* , tras ello se les asociará la vista correspondiente.

Clase CVibraApp (: CWinApp)

□ *Atributos:*

□ *Operaciones*

CvibraApp();
Constructor.
virtual BOOL InitInstance();
Inicialización de la aplicación.

Clase CViewGeneral (: CFormView)

□ *Atributos:*

CEdit m_frecbase;
Objeto cuadro de edición de texto para introducir la frecuencia de muestreo base.
Cbutton m_start;
Botón de comienzo de la adquisición.
Cbutton m_butabort;
Botón de finalización de la adquisición.
CEdit m_gtadq;
Cuadro de edición tiempo adquisición.
CEdit m_nomfichero;
Control de Cuadro de edición para introducir el nombre del fichero donde esta se guarda.
CComboBox m_combonum;
Desplegable para selección del número de dispositivo NI-DAQ de la tarjeta seleccionada en el combo *m_combonombre*.
CComboBox m_combonombre;
Desplegable para seleccionar la tarjeta NI-DAQ bajo la que se realizará la adquisición.
BOOL m_gc1, m_gc2, m_gc3, m_gc4, m_gc5, m_gc6, m_gc7, m_gc8;
BOOL m_gcd1, m_gcd2;
Booleanos que se asocian con los cuadros de selección del formulario para indicar los canales de los que se realizará adquisición.

□ *Operaciones*

CviewGeneral();
Constructor.
virtual ~CViewGeneral();
Destructor.
virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
Actualización del form a partir de los datos almacenados.
void OnGCanal();
Se ocupa de actualizar las variables booleanas asociadas con los cuadros de selección de canal, y de hacer coherente con estos los tabs de CviewTabs. Solo funciona si no se está adquiriendo.
void OnKillFocusEdits();
Validaciones tras la pérdida del foco por parte de los controles de edición.
void OnKillFocusCombos();
Almacena la selección actual del combo, y actualiza el combo de ganancias de CviewCanal según la selección.
void OnStart();
Comienza la adquisición.
void OnAbort();
Detiene la adquisición.

Clase CViewCanal (: CFormView)

□ *Atributos:*

CComboBox m_ganancia;
Combo para la selección de la ganancia del canal seleccionado.
float m_de, m_hasta;
Acotan en tensión la visualización de la adquisición.
CSliderCtrl m_slider;
Slider que indica el segmento visualizado sobre la escala completa de adquisición.
float m_intervalo;
Intervalo temporal de la visualización, a modo de base temporal de la visualización.
UINT m_paso;
Intervalo temporal de desplazamiento de la adquisición tras cada nuevo refresco.
int m_escalaf;
Indica la escala de la frecuencia base para este canal.
Int m_numcanal;
Indica el número de canal.
CString m_observaciones;
String para la introducción de observaciones.
BOOL m_grid;
Indica si se ha de visualizar grid o no en la visualización.
BOOL inicializado;

□ *Operaciones*

CviewCanal();
Constructor.
virtual ~CViewCanal();
Destructor.
virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
Actualización del form con los datos del tab seleccionado.
virtual void OnInitialUpdate();
Inicializaciones previas de los controles del form.
virtual void DoDataExchange(CDataExchange* pDX);
Intercambio de datos entre campos del form y los atributos asociados y/o asociación de los restantes campos a instancias de controles del mismo tipo.
void OnKillFocusEdits();
Los cambios se introducen cuando se cambia el foco del campo de edición.
void OnKillfocusCombo();
En el caso de los combo, los cambios también se actualizan cuando estos pierden el foco.
void OnPressGrid();
Conmuta el estado del atributo asociado.
void ChangeCtrlStyle(long lStyle, BOOL bSetBit);
Cambio de estilo para el control CSliderCtrl.

Clase CViewVibra (: CView)

□ *Atributos:*

CFile Adqf;
CFileException Adqe;
Instancias del objeto que gestiona el fichero donde se guardará la información, y el objeto que gestiona las excepciones de este.
struct ScanData datscan;
Estructura que mantiene los datos seleccionados en el form de canal, que serán tenidos en cuenta para la configuración de la siguiente adquisición.
struct VibraBuffers bufs;
Estructuras usadas para el almacenamiento intermedio y fuente del almacenamiento en disco duro de la adquisición.
struct ScreenBuffers screenbufs;
Estructuras usadas para la visualización.

□ *Operaciones*

CviewVibra();
Constructor.
virtual ~CViewVibra();
Destructor.
CVibraDoc* GetDocument();
Obtiene el puntero al objeto documento asociado con la aplicación.
void EscribeEnFichero();
Escritura de los buffers de adquisición correspondientes a los distintos canales analógicos, en disco. Proceso que se realiza dinámicamente durante la adquisición.
void StopAdq();
Para la adquisición.
void OnAdqDaqDb();
Inicia la adquisición indefinida basada en doble buffer.
virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
Actualización de la visualización.
virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
Se define en el procedimiento de ventana dos eventos del usuario, que corresponden a la actualización de la visualización y al vaciado de parte del buffer de adquisición.

Clase CVibraDoc (: CDocument)

□ *Atributos:*

BOOL IsAcquiring;
Indicador de si está en marcha el proceso de adquisición.
struct AnalogCanalStruct adata[10];
Almacena los parámetros de cada canal.
struct GeneralStruct gdata;
Almacena parámetros generales.

□ *Operaciones*

CvibraDoc();
Constructor: Inicializaciones y carga de los parámetros de la adquisición guardados.
virtual ~CVibraDoc();
Destructor: Guarda la configuración para la próxima sesión.

Clase CViewTabs (: CFormView)
□ Atributos: **CtabCtrl m_tab;** Control de tabs.
□ Operaciones **CviewTabs();** Constructor. **virtual ~CViewTabs();** Destructor. **virtual void OnInitialUpdate();** Se inicializa el tamaño de la partición de la ventana. **virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);** Actualización del control de tabs. Visualiza tantos *tabs* como canales activados existan. **void OnSelchangeTab1(NMHDR* pNMHDR, LRESULT* pResult);** La selección de un tab actualiza el form de canal y la vista de la visualización.

Las estructuras referenciadas son las siguientes:

Struct AnalogCanalStruct
□ Atributos: **int scalebasefreq;** Escala respecto de la frecuencia de muestreo base. **int ganancia;** Ganancia. **CString observaciones;** Observaciones. **BOOL grid;** Grid. **float de, hasta;** Acotan en tensión la visualización de la adquisición. **float intervalo;** Intervalo temporal de la visualización, a modo de base temporal de la visualización. **float paso;** Intervalo temporal de desplazamiento de la adquisición tras cada nuevo refresco.
□ Operaciones

Struct GeneralStruct
□ *Atributos:*
BOOL c1, c2, c3, c4, c5, c6, c7, c8, cd1, cd2; Canales seleccionados. **int freqbase;** Frecuencia de muestreo base. **long int tadq;** Tiempo en milisegundos. **int numdisp;** Número de dispositivo NI-DAQ. **int namedisp;** Nombre de la tarjeta NI en el número de dispositivo seleccionado. **CString namefich;** Nombre del fichero destino de la adquisición. **int canaldefault;** Tab seleccionado por defecto.
□ *Operaciones*

Struct ScanData
□ *Atributos:*
i16 numChans; Numero de canales a adquirir. **i16 chanVector[18];** Vector de canales a adquirir en el orden de adquisición correcto. **i16 gainVector[18];** Vector de las ganancias correspondientes a cada canal. **i16 RateVector[18];** Vector de divisiones de frecuencia correspondientes a cada canal. **i16 scansPorSecuencia;** **i16 muestrasPorSecuencia;** Una vez realizado el SCAN_Sequence_Setup, se almacenan el número de canales escaneados que completan una secuencia y el número de muestras en cada secuencia. **i16 muestrasSecuenciaVector[2000];** Este vector almacenará la muestras de los canales de una secuencia.
□ *Operaciones*

Struct ScreenBuffers

□ *Atributos:*

i16 sc_canal1[10000],sc_index_canal1;
Espacio para guardar los datos que debe visualizar el canal1 y puntero del próximo dato.
i16 sc_canal2[10000],sc_index_canal2;
Idem para el canal 2.
i16 sc_canal3[10000],sc_index_canal3;
Idem para el canal 3.
i16 sc_canal4[10000],sc_index_canal4;
Idem para el canal 4.
i16 sc_canal5[10000],sc_index_canal5;
Idem para el canal 5.
i16 sc_canal6[10000],sc_index_canal6;
Idem para el canal 6.
i16 sc_canalcd1[10000],sc_index_canalcd1;
Idem para el canal cd1.
i16 sc_canalcd2[10000],sc_index_canalcd2;
Idem para el canal cd2.

□ *Operaciones*

Struct VibraBuffers

□ *Atributos:*

i16 piHalfBuffer[50000];
Espacio para el buffer mitad donde se copiaran las mitades de los buffer. La longitud es muy grande para que sea cual sea la longitud del buffer intermedio, no haya de escalarse.
i16 canal1[10000], index_new_data_canal1;
Espacio para guardar los datos del canal1 y el índice donde irá el próximo dato.
i16 canal2[10000], index_new_data_canal2;
Idem para el canal3.
i16 canal3[10000], index_new_data_canal3;
Idem para el canal4.
i16 canal4[10000], index_new_data_canal4;
Idem para el canal5.
i16 canal5[10000], index_new_data_canal5;
Idem para el canal6.
i16 canal6[10000], index_new_data_canal6;
Idem para el canalcd1.
i16 canalcd1[10000], index_new_data_canalcd1;
Idem para el canalcd2.
i16 canalcd2[10000], index_new_data_canalcd2;
Idem para el canal2.

□ *Operaciones*

Bibliography

AD7730 (1998). *Bridge Transducer ADC AD7730/AD7730L.* Analog Devices.

ADXL (1998). *5g to 50g, Low Noise, Low Power, Single/Dual Axis iMEMS Accelerometers. ADXL150/ADXL250.* Analog Devices.

Analog (1999). *Applications and Technical Notes.* Analog Devices.

Calleja, J. (1999). *Diseño Y Programación de un Sistema de Control Distribuido Para Clasificación de Frutos: Control Y Comunicaciones CAN Y LAN.* Universitat de València. Proyecto Final de Carrera. Ingeniería Electrónica.

Calpe, J., F. J. e. a. (2000). DSP based weighting system for fruit sorting and grading machinery. *ICSPAT.*

Calpe, J., F. J. V. e. A. (Zaragoza, 1996). Aplicación de técnicas convencionales de procesado digital a la lectura de células de pesada. *Seminario Anual de Automática y Electrónica Industrial (SAAEI '96).*

Castelli, G., R. G. (Octubre 1995). A real-time operating system adapts to application architectures. *IEEE Micro.*

Clarkson, P. (1993). *Optimal and Adaptive Signal Processing.* CRC Press.

Dedé, E. J., E. J. (Barcelona, 1983.). Diseño de circuitos y sistemas electrónicos. *Marcombo Boixareu Editores.*

Deitel, H. M., D. P. J. (1994). *Como Programar En C/C++.* Prentice Hall, Englewood Cliffs.

Deitel, H. M. (1990). *Operating Systems.* Addison-Wesley Publishing Company.

Doscher, J. (1998). *Accelerometer Design and Applications.* Analog Devices Datasheet.

Embree, P. M., K. B. (1991). *C Language Algorithms for Digital Signal Processing.* Prentice-Hall, New Jersey.

Embree, P. M. (1995). *C Algorithms for Real-Time DSP.* Prentice Hall, Englewood Cliffs, NJ.

Frances, J. V., C. J. e. a. (Houston. Noviembre, 2000a). Application of ARMA modelling in fruit sorting and grading machinery. *ICSPAT 2000.*

Frances, J. V., C. J. e. A. (Istambul. Junio 2000.b). Application of ARMA modelling to the improvement of weight estimation in fruit sorting and grading machinery. *International Congress of Speech and Signal Processing, ICASSP 2000.*

Freedman, D., P. R. P. R. (1991). *Statistics.* Norton International StudentEdition.

Gottfried, B. S. (1991). *Programación En C.* McGraw-Hill.

Group, M. (1999). *Interactive Guide to Strain Gage Technology.* Measurements Group.

Grover, D., D. J. R. (1998). *Digital Signal Processing and the Microcontroller.* Motorola University. Prentice Hall, Englewood Cliffs, NJ.

Haykin, S. (1996). *Adaptive Filter Theory.* Prentice-Hall.

Ifechor, E. C., J. B. W. (1997). *Digital Signal Processing.* Addison-Wesley Longman.

Intel (1989). *Introducción al 80386.* Ediciones Anaya Multimedia.

Johnson, H. W., G. M. (1993). *High-Speed Digital Design.* Prentice-Hall, Englewood Cliffs.

Johnsonbaugh, R., K. M. (1996). *Applications Programming in ANSI.* Prentice Hall, Englewood Cliffs, NJ.

Kay, S.M.; Marple Jr, S. (1981). Spectrum analysis - a modern perspective. *Proc IEEE,* 69(11):1380–1419.

Kay, S. M. (1993). *Fundamentals of Statistical Signal Processing. Estimation Theory.* Director A.V.Oppenheim. Prentice Hall, Englewood Cliffs, NJ.

Kerninghan, B. W., R. D. M. (1988a). *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ.

Kerninghan, B. W., R. D. W. (1988b). *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ.

Krauss, T. P., S. L. L. J.Ñ. (1994). *Signal Processing Toolbox, for Use with MATLAB.* The Math Works, Inc.

Langsdam, Y., A. M. J. e. A. (1997). *Estructuras de Datos con C Y C++.* Prentice Hall, Englewood Cliffs, NJ.

Ljung, L. (1995). *System Identfication Toolbox, for Use with Matlab.* The Math Works, Inc.

Ljung, L. (1996). *System Identification: Theory for the User.* Prentice Hall, Englewood Cliffs, NJ.

LT (1994). *LTC1100 Datasheet.* Linear Technology.

Makhoul, J. (1975). Linear prediction: A tutorial review. *Proc IEEE,* 63(4):561–580.

Marple, S. L. (1987). *Digital Spectral Analysis with Applications.* Prentice Hall.

Marven, C., E. G. (1994). *A Simple Approach to Digital Signal Processing.* Texas Instruments. Alden Press Limited.

Matlab (1993). *MATLAB. Reference Guide.* The Math Works, Inc.

Matlab (1997). *Matlab Identification Toolbox.* The Math Works, Inc.

McClellan, J. H., B. C. S. e. a. (1998). *Computer Based Exercises for Signal Processing Using MATLAB.* Prentice Hall, Englewood Cliffs, NJ.

Milenkovic, M. (1988). *Sistemas Operativos. Conceptos Y Diseño.* McGraw-Hill.

Moon, T.K., S. W. C. (1999). *Mathematical Methods and Algorithms for Signal Processing.* Prentice Hall, Englewood Cliffs, NJ.

Newland, D. E. (1997). *An Introduction to Random Vibrations, Spectral and Wavelet Analysis.* Longman.

NI-AT (1996). *AT-MIO/AI E Series User Manual.* National Instruments. Part Number 320517E-01.

NI-Cat (1999). *Measurement and Automation Catalogue.*

NI-DAQ (1998). *NI-DAQ User Manual for PC Compatibles.* National Instruments. Part Number 321644C-01.

NI-DAQCard (1999). *DAQ-Card E Series Manual.* Part Number 321138C-01.

Oppenheim, A., S. R. (1975). *Discrete Time Signal Processing.* Prentice Hall, Englewood Cliffs, NJ.

Oppenheim, A., S. R. (1989). *Discrete Time Signal Processing.* Prentice Hall, Englewood Cliffs, NJ.

Oppenheim, A.V., S. R. B. J. (1999). *Discrete-Time Signal Processing.* Prentice-Hall.

Pallás, R. (Barcelona, 1993). *Adquisición Y Distribución de Señales.* Marcombo.

Pallás, R. (Barcelona, 1994). *Sensores Y Acondicionamiento de Señal.* Marcombo.

Parks, T. W., B. C. S. (1987). *Digital Filter Design.* John Wiley and sons.

Polo, S. B. (1999). *Aplicación Informática Para la Programación Y Mantenimiento de Las Instalaciones de Clasificación de Frutos.* Universitat de València. Proyecto Final de Carrera. Ingeniería Electrónica.

Proakis, J. G., M. D. G. (1998). *Tratamiento Digital de Señales. Principios, Algoritmos Y Aplicaciones.* Prentice-Hall, Madrid.

Quinnell, R. A. (Abril, 1995). Microkernel and modular OSs. *EDN.*

Rabiner, L. R., G. B. (1975). *Theory and Application of Digital Signal Processing.* Prentice-Hall.

Rojiani, K. B. (1996). *Programming in C with Numerical Methods for Engineers.* Prentice Hall, Englewood Cliffs, NJ.

Schildt, H. (1995). *Programación En C Y C++ En Windows 95.* Osborne McGraw-Hill.

Shay, W. (1997). *Introduction to Operating Systems.* Prentice Hall.

Siemens (1997). *C515C Data Sheet.* Siemens AG. version 12.97.

Soria, E., C. J. e. a. (1998a). Algorithm for improving the performance of adaptive systems. *J. of Electrical Eng.*, pages 270–272.

Soria, E., F. J. V. e. a. (1998b). Programación de dispositivos lógicos programables a través del entorno XILINX. *TAEE'98.*

Soria, E., F. J. V. e. A. (Noviembre, 1997). Estudio de las prestaciones en tiempo real del algoritmo LMS más fletcher-reeves encadenado. *XV Congreso de la Sociedad Española de Ingeniería Biomédica.*

Srinath, M. D., R. P. K. e. A. (1996). *Introduction to Statistical Signal Processing with Applications.* Prentice Hall, Englewood Cliffs, NJ.

Stearns, S. D., D. R. A. (1988). *Signal Processing Algorithms.* Prentice Hall, Englewood Cliffs, NJ.

Tanenbaum, A. (1992). *Modern Operating Systems.* Prentice Hall.

TI-Asm (1990). *TMS320C1X/TMS320C2X Assembly Language Tools Users's Guide.* Texas Instruments.

TI-C (1990). *TMS320C25 C Compiler Guide.* Texas Instruments.

TI-C2x (1993). *TMS320C2X. User's Guide.* Texas Instruments. 1604907-9721 revision C.

TI-Dac (1995). *Data Acquisition Circuits.* Texas Instruments. Data Book (SLAD001).

TI-Dsg (1996). *TMS320 DSP Designers's Notebook.* Texas Instruments. SPRT125.

TI-DSK (1993). *TMS320C2x DSP Starter Kit. Users Guide.* Texas Instruments. 2617630-9721 revision.

TI-Dtc (1993). *Data Transmission Circuits.* Texas Instruments. Data Book (SLLD001).

TI-Sim (1988). *TMS320 Family Simulator User's Guide.* Texas Instruments.

TI-Sup (1997). *TMS329 DSP Development Support. Reference Guide.* Texas Instruments. Catalog (SPRU011E).

Weiss, R. (October, 1994). DSP tools : Navigating the hardware/ software interface. *Computer Design.*

Xilinx (1998). *Designing with XC9500 CPLDs.* Xilinx. Aplication Note XAPP073 (version 1.3).

Xilinx (19997). *XC95108 In-System Programmable CPLD Data Sheet.* Xilinx. October 26, (version 2.0).

UNIVERSITAT DE VALÈNCIA FACULTAD DE CIÈNCIES FÍSIQUES

Reunit el Tribunal que subscriu, en el dia de la data, acordà d'atorgar, per unanimitat, a aquesta Tesi Doctoral d'En/ Na/ N' JOSÉ VICENTE FRANCÉS VILLORA la qualificació de SOBRESALIENTE CUM LAUDEM

València a 20 de JULIO de 2000

El President, El Secretari,

José Espí López MANUEL BATALLER