Manual del Micro Controlador Propeller

Versión en Español 1.1

GARANTÍA

Parallax Inc. garantiza sus productos contra defectos en materiales y fabricación por un periodo de 90 días a partir de la recepción del producto. Si el producto tiene un defecto, Parallax Inc. a su conveniencia reparara o remplazara la mercancía o reembolsara el monto de la compra. Antes de regresar un producto a Parallax se debe llamar a Parallax y obtener un numero de RMA (Autorización de Regreso de Material). Para regresar la mercancía a Parallax escriba el número de RMA es la parte exterior de la caja. Por favor incluya la siguiente información con la mercancía a regresar: Nombre, Número de Teléfono, Dirección de envío y una descripción del problema. Parallax regresara el producto o su reemplazo usando el mismo método de envío que se utilizo para enviar el producto a Parallax.

14-DÍAS DE GARANTÍA PARA REGRESO DEL DINERO

Si dentro de los 14 días de haber recibido el producto usted decide que no es útil para sus necesidades usted puede regresar el producto y obtener el un reembolso completo. Parallax Inc. Regresara el precio del producto excluyendo costos de envío y/o manejo. Esta garantía no es válida si el producto ha sido alterado o dañado. Vea la sección de Garantía mencionada en el párrafo anterior para instrucciones de regreso del producto a Parallax.

DERECHOS RESERVADOS Y MARCAS REGISTRADAS

ISBN 9-781928-982470

1.1.0-09.03.05-HKTP

ESTIPULACIONES DE REPONSABILIDAD LEGAL

Parallax Inc. no se hace responsable por incidentes especiales o daños resultantes de cualquier violación de garantía o cualquier otra teoría legal incluyendo perdidas de ganancias, tiempo perdido, daños o reemplazos de equipo o propiedad, así como cualquier costo de recuperación, reprogramación o reproducción de cualquier dato almacenado en o usado con los productos Parallax.. Parallax Inc. tampoco es responsable por daños personales incluyendo vida o salud resultante por el uso de cualquiera de nuestros productos. Usted toma completa responsabilidad.

LISTAS DE DISCUSIONES EN INTERNET

Mantenemos foros de discusión activos en internet para gente interesada en los productos Parallax. Estas listas son accesibles desde www.parallax.com :

- Propeller chip – Esta lista es específicamente para nuestros clientes que utilizan Propeller.
- BASIC Stamp – Esta lista es utilizada ampliamente por ingenieros, aficionados y estudiantes que comparten sus proyectos de BAIC Stamp y responden preguntas.
- Stamps in Class® – Creado para educadores y estudiantes, los suscriptores discuten el uso de la serie de tutoriales Stamps in Class en sus cursos. Esta lista proporciona una oportunidad para ambos educadores y estudiantes para obtener respuestas de sus preguntas.
- HYDRA – Para entusiastas del desarrollo de sistemas de video juego HYDRA
- Parallax Educadores – Un foro privado exclusivo para educadores y aquellos que contribuyen al desarrollo de materiales educativos para Stamps in Class y Propeller. Aquí se proporciona un lugar para educadores que desean desarrollar y compartir recursos de clase.
- Robotics – Diseñado para robots Parallax, este foro tiene la intención de abrir diálogos. Los temas incluyen ensamble, código fuente, expansiones y actualizaciones de manuales. En este foro se discute acerca del Boe-Bot®, Toddler®, SumoBot®, Scribbler® and Penguin™.
- SX Microcontrollers and SX-Key – Discusión de la programación del micro controlador SX con lenguaje ensamblador Parallax, herramientas SX-Key® y compiladores BASIC y C.
- Javelin Stamp – Discusión de aplicaciones y diseño usando Javelin Stamp, un modulo de Parallax que se programa usando un lenguaje una versión de Java de Sun Microsystems®.

ERRATA

A pesar de que se realiza un gran esfuerzo para lograr la precisión de nuestros textos podría existir algún error. Si encuentra un error por favor envíenos la información a editor@parallax.com. Continuamente mejoraremos nuestros materiales educativos. Ocasionalmente una lista de erratas será publicada en nuestro sitio web www.parallax.com. Por favor verifique las páginas de los productos individuales para encontrar el archivo de erratas correspondiente.

HARDWARE Y FIRMWARE SOPORTADO

Este manual es válido para las siguientes versiones de hardware y firmware:

Hardware	Firmware
P8X32A-D40 P8X32A-Q44 P8X32A-M44	P8X32A v1.0

CREDITOS

Autor: Jeff Martin. Formato y Edición, Stephanie Lindsay.
Cubierta: Jen Jacobs; Gráficos Técnicos: Rich Allred; muchas gracias a todos en Parallax Inc.
Traducción al Español: Oscar Villarreal. FIX ingenieria

Tabla de Contenido

Prefacio

Gracias por comprar un Chip Propeller. ¡En breve estarás programando en poco tiempo!

Los chips Propeller son micro controladores increíblemente capaces; el muy anticipado resultado de ocho años de trabajo y esfuerzos de Chip Gracey y el equipo entero de Parallax Inc.

Este libro tiene la intención de ser una guía de referencia para el chip Propeller y sus lenguajes nativos de programación, Spin y Ensamblador Propeller. Para un tutorial de programación y detalles de la herramienta Propeller por favor diríjase a la ayuda en línea que está instalada con el programa de la herramienta Propeller. ¡Diviértase!

Aun con nuestro mejor esfuerzo hay aun preguntas sin responder en este Manual por sí solo. Por favor revise los foros de consulta – (accesibles desde www.parallax.com vía Support → Discussion Forums menu) – este es un grupo especial para usuarios Propeller donde puedes publicar tus preguntas o revisar discusiones que quizá respondan actualmente a tus preguntas.

Además de le foro puedes visitar el Intercambio de Objetos Propeller (obex.parallax.com) para tener acceso a cientos de objetos Propeller hechos por clientes e ingenieros de Parallax. Aparte de ser útiles inmediatamente para tus propias aplicaciones, los objetos propeller escritos por varios autores son una gran fuente de técnicas de estudio y trucos empleados por la comunidad activa Propeller.

Nota del Editor: Acerca de la Versión 1.1

El mayor contenido agregado, correcciones y eliminaciones hechas al Manual Propeller 1.0 para producir esta edición fueron marcados en la versión PDF de este documento del Manual Propeller v1.1. Recomendamos que lo revise si usted ha leído la edición original del Manual Propeller. Una lista completa de cambios se puede encontrar en el suplemento del Manual Propeller y Errata v1.4. Ambos documentos están disponibles para descarga en www.parallax.com/Propeller.

Los más significativos, el pasado *Capitulo 2: Usando la Herramienta Propeller*, se movieron a la ayuda en línea de la herramienta Propeller donde puede actualizarse frecuentemente para permanecer en sincronía con las mejoras al desarrollo del software. Del mismo modo el *Capitulo 3: Tutorial de Programación Propeller* se movió a la ayuda en línea de la herramienta Propeller donde se pudo expandir y hacer más legible.

Cambios importantes incluyen:

- Se agrego una instrucción del Ensamblador Propeller; TESTN (ver página 347)
- Las siguientes secciones se escribieron para clarificar:
  - ADDSX en página 268
  - ADDX en página 270
  - CALL en página 274
  - CMPSX en página 283
  - CMPX en página 286
  - JMPRET en página 308
  - ORG en página 338
  - RES en página 352
  - RET en página 356
  - SUBSX en página 366
  - SUBX en página 368
- Se hicieron mejoras extensivas a las siguientes secciones para proporcionar detalles en características no documentadas previamente:
  - BYTE en página 54
  - COGINIT en página 79
  - COGNEW en página 81
  - DAT en página 102
  - LONG en página 132
  - WORD en página 232
- Se hicieron revisiones extensivas a La Estructura del Ensamblador Prop comenzando en la página 243, al mismo tiempo que al inicio de la sección Elementos de Lenguaje Ensamblador que inicia en la pagina 255.
- Se agregaron Tablas de verdad en la explicación de cada instrucción de Ensamblador Propeller. Cada una de estas tablas incluye valores clave y combinaciones de banderas que revelan aspectos importantes de la naturaleza de la instrucción.
- Se proporcionan efectos individuales y registros en las propias secciones en la Referencia de Lenguaje Ensamblador para una fácil localización en el Manual.
- Se agregaron ejemplos de Multiplicación, División y Raíz Cuadrada al Apéndice B.
- Cientos de detalles importantes se mejoraron o corrigieron. Ver la versión PDF del Manual o el suplemento y Errata v1.4 para mayor información.

Capítulo 1: Introducción al Chip Propeller

Este capítulo describe el hardware del chip Propeller. Para entender completamente y usar el Propeller efectivamente es importante entender primero su arquitectura. Este capítulo presenta los detalles del hardware tales como tipos de paquetes, tamaños, descripciones de pins y funciones.

Concepto

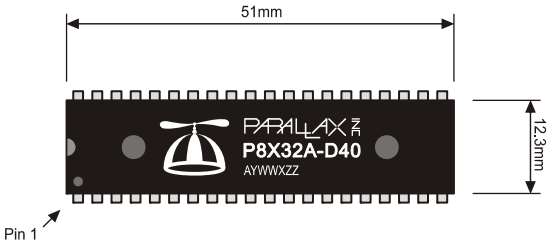
El chip Propeller está diseñado para proporcionar procesamiento de alta velocidad para sistemas incrustados y al mismo tiempo mantener bajo consumo de corriente e impresos físicos pequeños. Además de ser rápido el Propeller proporciona flexibilidad y potencia a través de sus ocho procesadores, llamados cogs, que pueden desarrollar independientemente tareas cooperativas o simultaneas, todo mientras se mantiene una arquitectura relativamente simple la cual es fácil de entender y utilizar.

El resultado del diseño del Propeller libera aplicaciones a los desarrolladores de complejidades comunes en sistemas incrustados. Por ejemplo:

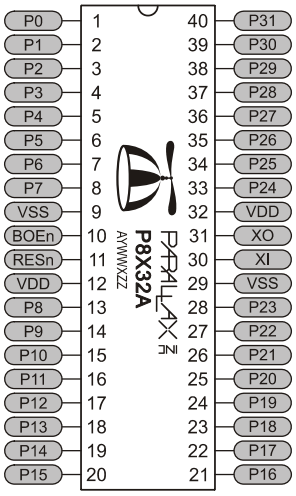
- El mapa de memoria es plano. No hay necesidad de esquemas de páginas con bloques de código, datos o variables. Esto es un ahorrador de tiempo durante el desarrollo de la aplicación.
- Eventos asíncronos son más sencillos de manejar que aquellos que usan interrupciones. El Propeller no tiene necesidad de interrupciones, solo asigne algunos cogs a tareas individuales, alto ancho de banda, y mantenga los otros cogs libres y sin compromiso. EL resultado es una aplicación de mayor respuesta y fácil de mantener.
- El lenguaje Ensamblador Propeller tiene características de ejecución condicional y escritura opcional de los resultados para cada instrucción individual. Esto lo hace crítico, bloques de código mejor temporizados, manejadores de eventos son menos propensos a fluctuaciones y los desarrolladores generan menos tiempo de relleno y dejan de apretar ciclos aquí y allá.

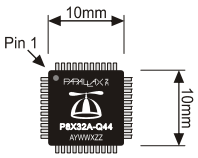
Tipos de Encapsulados

El chip Propeller está disponible en los tipos de paquetes mostrados aquí:

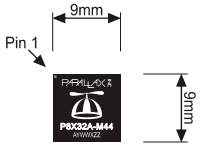


P8X32A-D40 ***40-pin DIP***

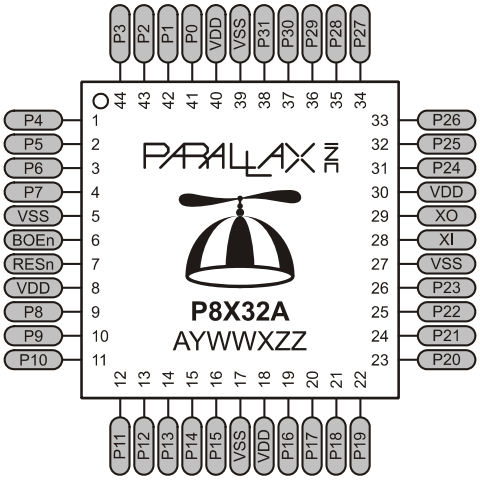




P8X32A-Q44
44-pin LQFP



P8X32A-M44
44-pin QFN



Descripción de Pins

Tabla 1-1: Descripciones de Pins		
Nombre	**Dirección**	**Descripción**
P0 – P31	E/S	Propósito General Puerto A E/S. Puede proporcionar 40 mA a 3.3 VDC. Disparo Lógico es ≈ ½ VDD; 1.65 VDC @ 3.3 VDC. Los pins mostrados tienen un propósito especial al encender o reiniciar pero son de propósito general posteriormente. P28 - I2C SCL Conexión a opcional EEPROM externa. P29 - I2C SDA Conexión a opcional EEPROM externa. P30 - Serial Tx a receptor. P31 - Serial Rx de receptor.
VDD	---	3.3 volt Potencia (2.7 – 3.3 VDC).
VSS	---	Tierra.
BOEn	E	Brown Out Enable (Activo bajo). Debe conectarse a VDD o VSS. Si esta en bajo, RESn se convierte en una salida débil (proporcionando VDD a través de 5 KΩ) para propósitos de monitoreo pero aun puede manejarse bajo para ocasionar un Reinicio. Si esta en alto, RESn es una entrada CMOS con Schmitt Trigger.
RESn	E/S	Reinicio (activo bajo). Cuando está en bajo reinicia el chip Propeller: Todos los cogs se deshabilitan y las E/S flotan. El Propeller reinicia 50 ms después de la transición de RESn de bajo a alto.
XI	E	Entrada Cristal. Puede conectarse a la salida de un paquete de cristal/oscilador (con XO desconectada) o a una terminal del cristal (con XO conectado a la otra terminal) dependiendo del programa de los registros CLK. No requiere resistencias o capacitares externos.
XO	S	Salida Cristal. Proporciona retroalimentación de un cristal externo o puede dejarse desconectado dependiendo de la programación del registro CLK. No requiere resistencias o capacitores externos.

El Propeller (P8X32A) tiene 32 pins de E/S (Puerto A, pins P0 a P31). Cuatro de estos pins de E/S, P28-P31 tienen un propósito especial durante el inicio o reinicio. Al iniciar-reiniciar los pins P30 y P31 se comunican con un host para programación y P28 y P29 interactúan con una EEPROM externa de 32KB (24LC256).

Especificaciones

Tabla 1-2: Especificaciones	
Modelo	P8X32A
Requerimientos de potencia	3.3 volts DC. (Max consumo de corriente debe limitarse a 300 mA).
Velocidad de Reloj Externo	DC a 80 MHz (4 MHz a 8 MHz con Clock PLL corriendo)
Velocidad de Reloj de Sistema	DC a 80 MHz
Oscilador Interno RC	12 MHz o 20 kHz (aproximado; rango de 8 MHz – 20 MHz, o 13 kHz – 33 kHz, respectivamente)
RAM/ROM Principal	64 K bytes; 32 KB RAM + 32 KB ROM
RAM del Cog	512 x 32 bits cada uno
Organización de RAM/ROM	Long (32-bit), Word (16-bit), o Byte (8-bit) direccionables
Pins E/S	32 señales CMOS con umbral de entrada VDD/2.
Fuente de Corriente/Consumo por E/S	40 mA
Consumo de Corriente @ 3.3 vdc, 70 °F	500 µA por MIPS (MIPS = Freq en MHz / 4 * Numero de Cogs Activos)

Conexiones de Hardware

La Figura 1-1 muestra un ejemplo del cableado que proporciona acceso al hospedaje y EEPROM que accesa el chip Propeller. En este ejemplo el acceso al hospedaje se logra a través del Propeller Plug (un convertidor serial USB a TTL).

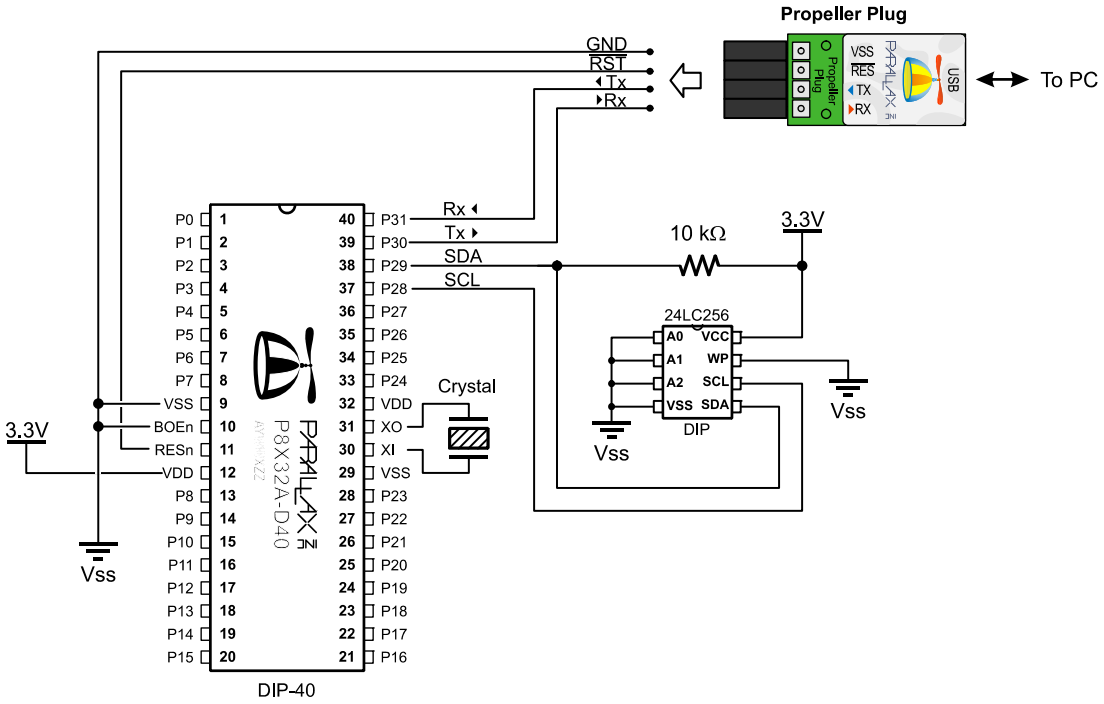


Figura 1-1: Ejemplo del diagrama de conexiones que permite programar el Propeller y una EEPROM externa de 32 Kbyte así como el Propeller usando un cristal externo.

Procedimiento de Inicio

Una vez que se enciende (+100 ms), RESn va de bajo-alto o se reinicia el software:

1. El chip Propeller inicia su reloj interno en modo lento (≈ 20 kHz), se retrasa por 50 ms (retraso de reinicio), cambia el reloj a modo rápido (≈ 12 MHz) y luego carga y corre el programa de inicio que esta cargado en el primer procesador (Cog 0).
2. El cargador de inicio desarrolla una o más de las siguientes tareas en orden:
    a. Detecta comunicación de un host, tal como una PC o pins P31 y P31. Si se detecta comunicación de un host el programa se comunica con él para identificar el chip Propeller y la posiblemente descargar un programa en la RAM principal y opcionalmente en la EEPROM externa de 32 KB.
    b. Si no hay comunicación con un host el cargador busca la memoria externa EEPROM de 32 KB (24LC256) en los pins P28 y P29. Si se detecta la EEPROM la imagen completa de los 32 KB se carga en la memoria RAM principal del chip Propeller.
    c. Si no se detecta la EEPROM, el cargador se detiene, el Cog 0 finaliza, el chip Propeller se va a modo de apagado y todos los pins E/S quedan como entradas.
3. Si cualquiera de los pasos 2a o 2b cargaron un programa en la memoria RAM y no se dio una instrucción de suspender el comando por el host entonces el Cog 0 se recarga con el interprete spin y el código de usuario se corre desde la RAM principal.

Procedimiento de ejecución

Una aplicación Propeller es un programa de usuario compilado en su forma binaria y descargada a la RAM del chip Propeller y posiblemente a la EEPROM externa. La aplicación consiste de código escrito en lenguaje spin del chip Propeller (código de alto nivel) con componentes de Ensamblador Propeller opcionalmente (código de bajo nivel). El código escrito en lenguaje spin se interpreta por un Cog al momento que corre el intérprete mientras que el código escrito en Ensamblador Propeller corre en su forma pura directamente por el Cog. Cada aplicación Propeller consiste de al menos un pequeño código spin y quizá puede estar escrito completamente en spin o con varias partes de spin y ensamblador. El interprete spin del chip Propeller inicia en el paso 3 del procedimiento de arranque para poder correr la aplicación.

Una vez que el procedimiento de arranque se complete y una aplicación está corriendo en el Cog 0 la demás actividad se define por la aplicación misma. La aplicación tiene control

completo sobre cosas como velocidad del reloj interno, uso de E/S, configuración de registros y cuando y cuantos cogs corren en determinado tiempo. Todo esto variable mientras corre ya que es controlado por la aplicación, incluyendo la velocidad del reloj interno.

Procedimiento de apagado

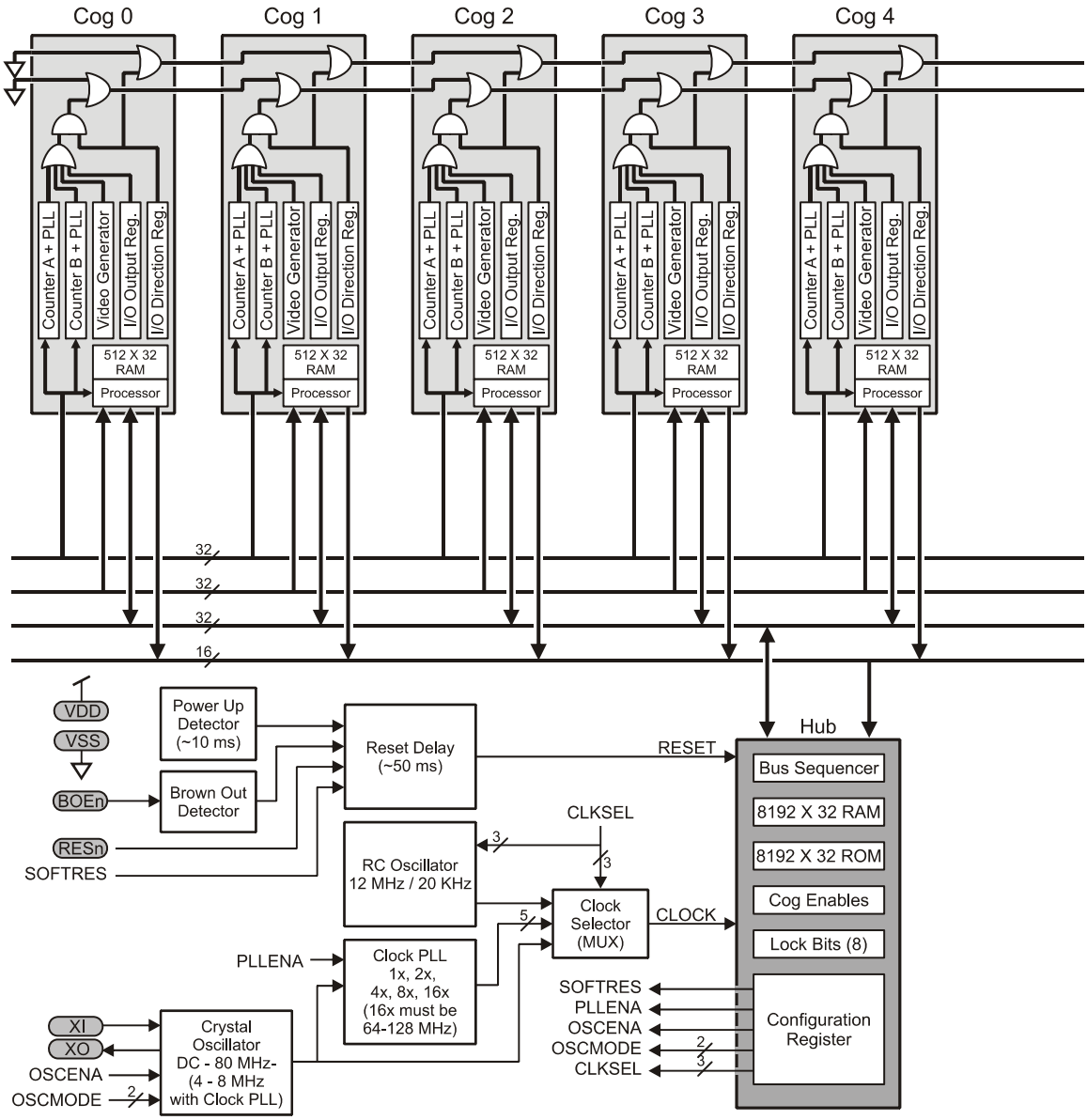
Cuando el Propeller queda en modo de apagado el reloj interno se detiene y al mismo tiempo detiene los Cogs, así mismo todos los pins de E/S quedan activados como entradas (alta impedancia). El modo de apagado puede ser ocasionado por tres de los siguientes eventos:

1) VDD queda por debajo del límite brown-out (≈2.7 VDC), cuando el circuito brown-out está habilitado.
2) El pin RESn está en modo bajo, o
3) La aplicación solicita un reinicio (ver comando `REBOOT`, pagina 191).

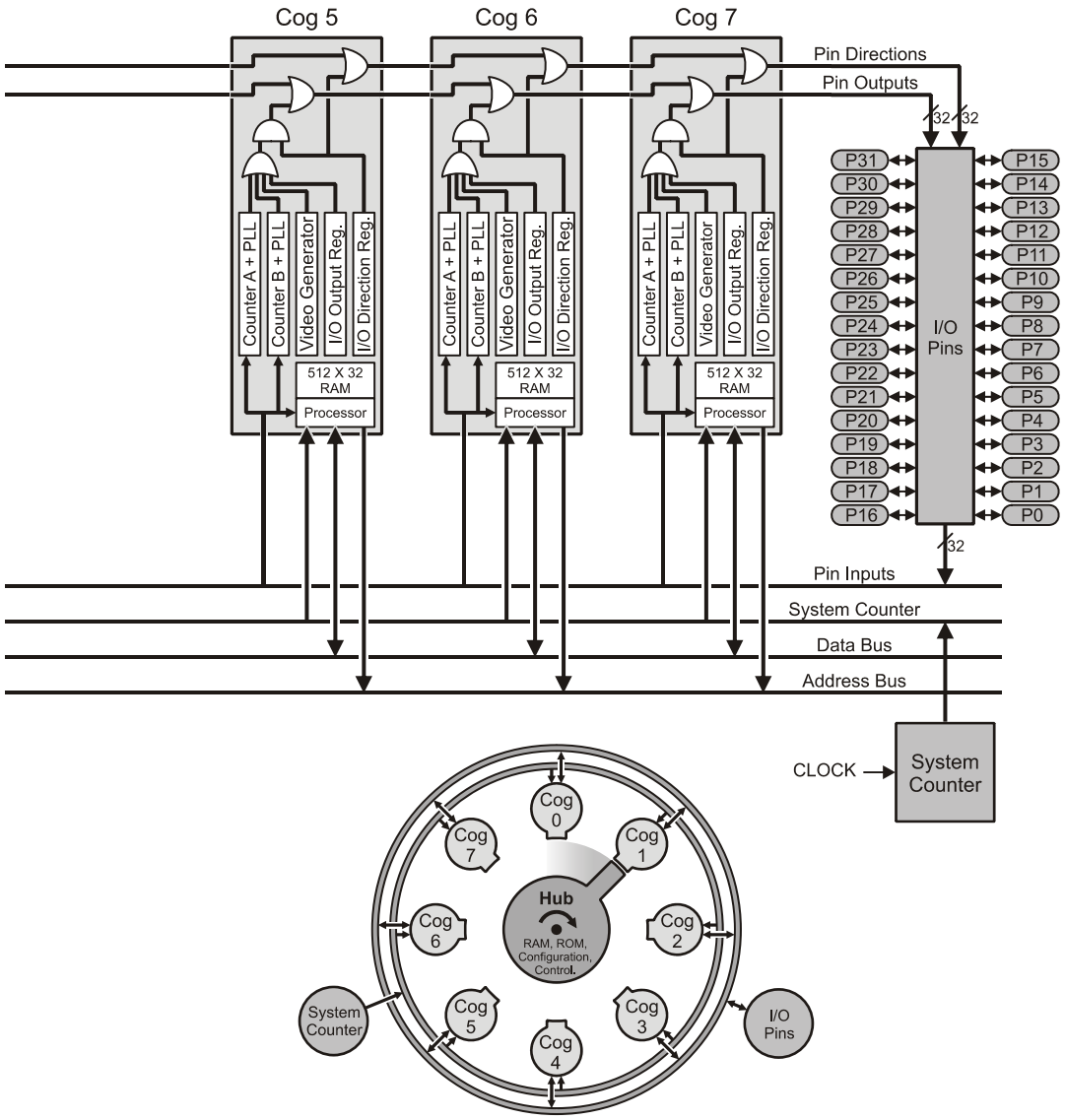
El modo de apagado termina cuando el nivel del voltaje queda por arriba del disparo brown-out y el pin RESn queda en alto.

Diagrama de Bloques

Figura 1-2: Diagrama de Bloques del Chip Propeller



La relación Cog y Hub es crítica para el chip Propeller. El Hub controla al Cog que tiene acceso a los recursos mutuamente exclusivos, tales como RAM/ROM principal, configuración de registros, etc. El Hub le da acceso exclusivo a cada cog en determinado momento de forma “round robin” sin importar cuantos cogs estan corriendo.



Hub and Cog Interaction

Recursos Compartidos

Hay dos tipos de recursos compartidos en el Propeller: 1) Común y 2) Mutuamente exclusivos. Los recursos comunes pueden accesarse en cualquier momento por cualquier número de cogs. Los recursos mutuamente exclusivos pueden accesarse por todos los cogs pero solo por un Cog a la vez. Los recursos comunes son los pins E/S y el contador del sistema. Todos los demás recursos son mutuamente exclusivos por naturaleza y su acceso es controlado por el hub. Vea la sección Hub en la pagina 24.

Reloj del Sistema

El reloj del sistema (mostrado como CLOCK en la Figura 1-2) es la fuente del reloj central para casi cada componente del chip Propeller. La señal del reloj del sistema viene de una de tres posibles Fuentes: 1) El oscilador interno RC, 2) El reloj de Fase de Ciclo Cerrado (PLL), o 3) el cristal oscilador (un circuito interno que se alimenta de un cristal externo o un paquete cristal/oscilador). La fuente se determina por la programación del registro CLK, el cual se puede seleccionar al compilar o mientras está corriendo. Los únicos componentes que no usan el reloj del sistema directamente son el Hub y el Bus; estos dividen el reloj del sistema por dos (2).

Cogs (Procesadores)

El Propeller contiene ocho (8) procesadores, llamados Cogs, numerados del 0 al 7. Cada Cog contiene los mismos componentes (ver Figura 1-2): un bloque procesador, 2KB de RAM local configurada como 512 longs (512 x 32 bits), dos módulos contadores con PLL, un generador de video, registro de E/S y otros registros que no se muestran en el diagrama. Ver la Tabla 1-3 para una lista completa de los registros del cog. Cada cog se diseña exactamente igual y puede correr tareas independientemente de los otros.

Los ocho cogs son manejados por la misma fuente de tiempo, el reloj de sistema, así que todos mantienen la misma referencia y todos los cogs activos ejecutan instrucciones simultáneamente. Ver Reloj del Sistema. También tienen acceso a los mismos recursos compartidos como pins E/S, RAM principal y el contador del sistema. Ver Recursos Compartidos.

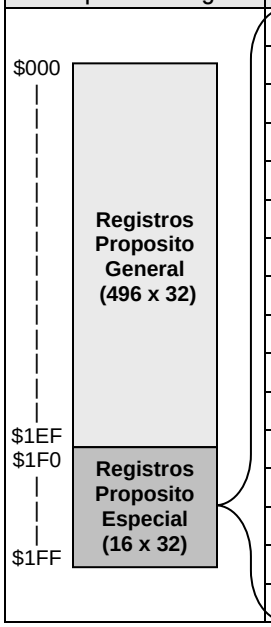
Los Cogs pueden iniciar o detenerse en tiempo real y pueden programarse para desarrollar tareas simultáneas, ya sea independientemente o en coordinación con otros cogs a través de la RAM principal. Sin importar la naturaleza de su uso el diseñador de la aplicación Propeller tiene control total sobre cómo y cuando se usa un cog; no hay control de compilador o control de sistemas operativos dividiendo tareas entre los múltiples cogs. Esto permite al

desarrollador a entregar tiempos determinados, consumos de potencia y respuesta a la aplicación desarrollada.

Cada cog tiene su propia RAM, llamada RAM cog, a cual contiene 512 registros de 32 bits cada uno. La RAM cog es de propósito general excepto los últimos 16 registros, los cuales son de propósito especial como se describe en la Tabla 1-3. La RAM cog se usa para código ejecutable, datos, variables y las ultimas 16 localidades sirven como interface al contador del sistema, Pins E/S y periféricos locales del cog.

Cuando se inicia un cog, las direcciones ($000) a 495 ($1EF) se cargan secuencialmente de la RAM / ROM principal y sus localidades de propósito especial 496 ($1F0) a 511 ($1FF) se limpian a cero. Después de cargarlas, el cog comienza a ejecutar instrucciones, empezando con la localidad 0 de la RAM del cog. Continuara ejecutando código hasta que se detiene o reinicia ya sea por el mismo o por otro cog o cuando una interrupción de corriente ocurre.

Tabla 1-3: RAM del Cog. Registros de propósito especial



Direcc.	Nombre	Tipo	Descripción
$1F0	PAR	Solo Lectura[1]	Parámetro Boot, p. 182, 342
$1F1	CNT	Solo Lectura[1]	Contador del Sistema, p. 76, 288
$1F2	INA	Solo Lectura[1]	Estado de Entradas P31–P0, p. 122, 304
$1F3	INB[3]	Solo Lectura[1]	Estado de Entradas P63–P32, p. 122, 304
$1F4	OUTA	LectoEscritura	Estado de Salidas P3–P0, p. 179, 340
$1F5	OUTB[3]	Lectoescritura	Estado de Salidas P63–P32, p. 179, 340
$1F6	DIRA	lectoescritura	Estado direcciones P31–P0, p. 107, 456
$1F7	DIRB[3]	Lectoescritura	Estado direcciones P63–P32, p. 107, 456
$1F8	CTRA	lectoescritura	Control Contador A, p. 98, 294
$1F9	CTRB	Lectoescritura	Control Contador B, p. 98, 294
$1FA	FRQA	lectoescritura	Contador Frecuencia A, p. 114, 300
$1FB	FRQB	Lectoescritura	Contador Frecuencia B, p. 114, 300
$1FC	PHSA	LectoEscritura[2]	Contador Fase A, p. 184, 344
$1FD	PHSB	LectoEscritura[2]	Contador Fase B, p. 184, 344
$1FE	VCFG	Lectoescritura	Configuración de Video, p. 218, 381
$1FF	VSCL	Lectoescritura	Escala de Video, p. 221, 382

Nota 1: Para ensamblador Propeller, solo accesible como registro fuente (ej., `mov` *dest*, *source*). Ver la sección de lenguaje ensamblador para PAR, pag 342; CNT, pag 288, e INA, INB, pag 304.

Note 2: Para ensamblador Propeller solo legible como registro fuente (ej., `mov` *dest, source*); leer modificar-escribir no es posible como registro destino. Ver lenguaje ensamblador sección `PHSA, PHSB` en pag 344.

Note 3: Reservado para uso futuro.

Cada registro de propósito especial puede accesarse vía:

1) Su dirección física de registro (Ensamblador Propeller),
2) Su nombre predefinido (Spin o Ensamblador Propeller), o
3) El registro del arreglo variable (`SPR`) con un índice de 0 a 15 (Spin).

Los siguientes son ejemplos de Ensamblador Propeller:

```
MOV   $1F4, #$FF        'Activa OUTA 7:0 alto
MOV   OUTA, #$FF        'Igual que arriba
```

Los siguientes son ejemplos de Spin:

```
SPR[$4] := $FF          'Activa OUTA 7:0 alto
OUTA := $FF             'Igual que arriba
```

Hub

Para mantener la integridad del sistema, los recursos mutuamente exclusivos no pueden accesarse por más de un cog al mimo tiempo. El Hub mantiene esta integridad controlando el acceso a los recursos mutuamente exclusivos dando a cada cog su turno para accesarlos de manera “round robin” desde el Cog 0 hasta el Cog 7 y de regreso al Cog 0 nuevamente. El Hub y el Bus que controla corren a la mitad del ritmo del reloj del sistema. Esto significa que el Hub le da al cog acceso a los recursos mutuamente exclusivos cada 16 ciclos de reloj del sistema. Las instrucciones del Hub y las instrucciones del ensamblador Propeller que accesan los recursos mutuamente exclusivos requieren 7 ciclos para ejecutarse, pero primero necesitan sincronizarse al inicio de la ventana de acceso del Hub. Toma hasta 15 ciclos (16 menos 1, si lo perdió) para sincronizar la ventana de acceso al Hub mas 7 ciclos para ejecutar la instrucción del Hub, por lo tanto las instrucciones del hub toman de 7 a 22 ciclos para completarse.

La Figura 1-3 y Figura 1-4 muestran ejemplos donde el Cog 0 tiene una instrucción para ejecutar. La Figura 1-3 muestra el mejor caso; la instrucción estaba lista al empezar el acceso al cog. La instrucción se ejecuta inmediatamente (7 ciclos) dejando 9 ciclos adicionales para otras instrucciones antes de que la ventana de acceso al siguiente Hub llegue.

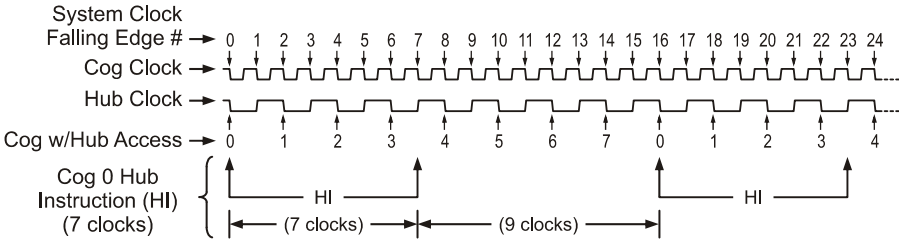


Figura 1-3: Interacción Cog-Hub – Mejor escenario

La Figura 1-4 muestra el peor escenario; la instrucción del hub estaba lista en el ciclo inmediato posterior después de iniciar la ventana de acceso al Cog 0; perdió el acceso. El Cog espera a la siguiente ventana de acceso al Hub (15 ciclos después) y entonces ejecuta la instrucción (7 ciclos) para un total de 22 ciclos para esa instrucción. Nuevamente hay 9 ciclos adicionales después de la instrucción del hub para ejecutar otras instrucciones antes de que llegue la nueva ventana de acceso. Para obtener la mayor eficiencia de las rutinas de Ensamblador Propeller que tienen que accesar a recursos mutuamente exclusivos puede ser de beneficio intercambiar instrucciones de hub con instrucciones de no-hub para reducir el número de ciclos en espera para el siguiente acceso. Debido a que la mayoría de las instrucciones propeller toman 4 ciclos de reloj pueden ejecutarse dos de estas instrucciones entre instrucciones contiguas de Hub.

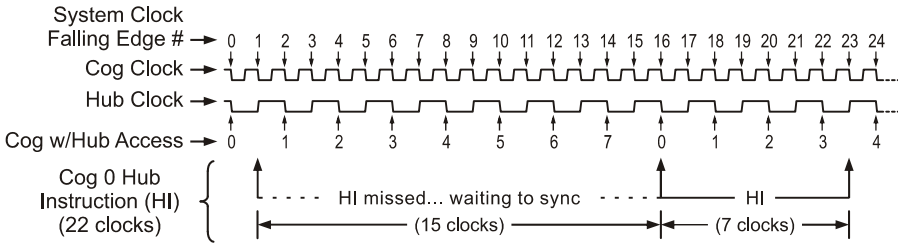


Figura 1-4: Interacción Cog-Hub – Peor escenario

Tenga en cuenta que una instrucción particular de un Cog no interfiere de ninguna manera con otras instrucciones de Cogs debido al mecanismo. El Cog 1 por ejemplo puede iniciar una instrucción durante el ciclo 2 del reloj del sistema, en estos ejemplos quizá se encimen las instrucciones con la del Cog 0 sin tener efectos. Mientras tanto los demás cogs continúan ejecutando instrucciones no-hub o esperando su ventana individual de acceso sin importar lo que otros estan haciendo.

Pins E/S

El Propeller tiene 32 pins de E/S, 28 de los cuales son enteramente de propósito general. Cuatro pins E/S (28-31) tienen un propósito especial al inicio y después estan disponibles para propósito general; ver la sección de Procedimiento de Inicio en la pagina 18. Después de iniciar cualquier pin E/S puede usarlo cualquier cog en cualquier momento ya que los pins de E/S son recursos comunes. Depende de la aplicación del desarrollador asegurar que dos cogs no traten de usar el mismo pin E/S para evitar conflictos durante la ejecución del programa.

Para detalles de hardware E/S, observe la parte interna de los cogs en la Figura 1-2 página 20 mientras lee la siguiente explicación.

Cada cog tiene sus propios Registros de Dirección 32-bit E/S y Registros de salida 32-bit E/S para influir en las direcciones y estados de salida del chip Propeller y corresponde a los 32 pins E/S. Las direcciones deseadas del cog y estados de salida se comunica a través del cog colectivamente para finalmente convertirse en lo que se llama “Dirección de Pin” y “Pin de Salida” en la esquina superior derecha de la Figura 1-2 de la pagina 20.

El cog colectivamente determina la Dirección de Pin y Salida de Pins como sigue:

1. La dirección del pin es el resultado de la OR de los estados de salida de los cogs.
2. Las salidas son el resultado de la OR de los estados de salida de todos los cogs. Un estado de salida del Cog consiste en los bits de sus módulos I/O (los contadores, el generador de video y el registro de salidas E/S) OR juntas y luego AND con los bits de dirección del registro.

En esencia cada dirección de pin E/S y estado de salida es el “OR” del cog entero. Esto permite a los cogs accesar e influir sobre los pins E/S simultáneamente sin la necesidad de cualquier recurso negociador y sin cualquier posibilidad de contención eléctrica entre los cogs.

El resultado de la configuración de este cableado de pins E/S puede describirse fácilmente con las siguientes reglas:

A. Un pin es una Entrada SOLO si un cog no lo activa como Salida.
B. Un pin es Salida Bajo SOLO si todos los cogs lo activan como Salida Baja
C. Un pin es Salida Alta si CUALQUIERA de los cogs lo activa como Salida Alta

La Tabla 1-4 demuestra algunas posibles combinaciones de la influencia colectiva de los cogs en un particular pin E/S, P12 en este ejemplo. Para simplificación estos ejemplos asumen que el bit 12 de cada hardware de E/S del Cog son puestos a cero (0).

Tabla 1-4: Ejemplos E/S				
	Bit 12 de Cogs' E/S Registro Dirección	Bit 12 de Cogs' E/S Registro Salida	Estado de E/S Pin P12	Regla Seguida
Cog ID	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7		
Ejemplo 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Entrada	A
Ejemplo 2	1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	Salida Baja	B
Ejemplo 3	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	Salida Alta	C
Ejemplo 4	1 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Salida Baja	B
Ejemplo 5	1 1 0 0 0 0 0 0	0 1 0 0 0 0 0 0	Salida Alta	C
Ejemplo 6	1 1 1 1 1 1 1 1	0 1 0 1 0 0 0 0	Salida Alta	C
Ejemplo 7	1 1 1 1 1 1 1 1	0 0 0 1 0 0 0 0	Salida Alta	C
Ejemplo 8	1 1 1 0 1 1 1 1	0 0 0 1 0 0 0 0	Salida Baja	B

Nota: Para el Registro de dirección un 1 en una localidad activa el pin correspondiente E/S a la dirección de salida mientras que un 0 lo active como dirección de entrada.

Cualquier cog que se apaga tiene sus registros de dirección y estados de salida puestos a cero, efectivamente removiéndolo de la influencia del estado final de los pins E/S que los restantes Cogs activos están controlando.

Cada cog tiene también sus propios Registros de Entrada de 32-bit. Este registro de entrada es realmente un pseudo registro; cada vez que se lee, el estado actual del registro es leído sin importar la dirección de entrada o salida.

Contador del Sistema

El contador del sistema es global, de solo lectura y 32-bit que se incrementa con cada ciclo del reloj. Los cogs pueden leer el contador del sistema (vía su registro `CNT`, Pág. 76) para desarrollar cálculos de tiempo usando el comando `WAITCNT` (Pág. 223) para crear retrasos efectivos en el proceso. El contador del sistema es un recurso común. Cada cog puede leerlo simultáneamente. El contador del sistema no se limpia una vez iniciado el sistema ya que tiene uso práctico para diferenciales de tiempo. Si un cog necesita mantener un tiempo específico solo necesita leer y guardar el valor del contador inicial en ese momento para luego compararlo con otros valores respecto a ese valor inicial.

Registro CLK

El registro CLK es el control de la configuración del reloj del sistema; determina la fuente y las características del reloj del sistema. El Registro CLK configura el oscilador RC, el Reloj PLL, el cristal oscilador y los circuitos selectores de reloj. (Ver Figura 1-2: Diagrama de Bloques del Chip Propeller en Pág. 20.) Se configura en la compilación por la constante _CLKMODE (Pág. 71) y puede escribirse en tiempo real a través del comando Spin CLKSET (Pág. 74) o con la instrucciones Ensamblador CLKSET (Pág. 277). Cada que el registro CLK se escribe, un retraso global de ≈75 µs ocurre como transición de fuentes de reloj.

Cada vez que se cambia el registro una copia del valor escrito deberá permanecer en el valor de la localidad Modo de Reloj (el cual es BYTE[4] en RAM principal) y el resultado de la frecuencia del reloj maestro deberá escribirse en el valor de la localidad Frecuencia del Reloj (el cual es LONG[0] en RAM principal) así los objetos a los cuales hace referencia tendrán disponible esta información para sus cálculos de tiempo. (Ver CLKMODE, Pág. 70, y CLKFREQ, Pág. 66.) Cuando sea posible se recomienda usar el comando Spin CLKSET (Pág. 74), ya que automáticamente actualice todas las localidades mencionadas.

Solo cierto patrón de bits en el registro CLK es valido en modos de reloj. Ver constante _CLKMODE en Pág. 71 y Tabla 2-4 en Pág. 72 para mas información. El objeto Clock en la librería Propeller puede ser útil ya que proporciona infamación de modificaciones de reloj y métodos de tiempo.

Tabla 1-5: Estructura del Registro CLK

Bit	7	6	5	4	3	2	1	0
Nombre	RESET	PLLENA	OSCENA	OSCM1	OSCM0	CLKSEL2	CLKSEL1	CLKSEL0

Tabla 1-6: RESET del Registro CLK (Bit 7)

Bit	Efecto
0	Siempre escribe ‘0’ aquí a menos que intente resetear el chip.
1	Mismo que el reinicio por hardware. El comando Spin REBOOT escribe ‘1’ al bit RESET.

Tabla 1-7: PLLENA del Registro CLK (Bit 6)	
Bit	**Efecto**
0	Deshabilita el circuito PLL. Los parámetros RCFAST y RCSLOW de la declaración _CLKMODE configura PLLENA de esta forma.
1	Habilita el circuito PLL. Cada uno de los parámetros PLLxx del la declaración _CLKMODE configura PLLENA de esta forma al momento de compilación. El Reloj PLL internamente multiplica la frecuencia del pin XIN por 16. OSCENA debe ser '1' para propagar la señal XIN al reloj PLL. La frecuencia interna del reloj PLL debe mantenerse entre 64 MHz a 128 MHz – esto transfiere a un XIN un rango de frecuencia de 4 MHz a 8 MHz. Permita 100 µs al reloj PLL para estabilizar antes de cambiar a una de sus salidas vía los bits CLKSELx. Una vez que l cristal oscilador y los circuitos del reloj PLL están estables y disponibles usted puede cambiar libremente entre las diversas fuentes de reloj para cambiar los bits CLKSELx.

Tabla 1-8: OSCENA del Registro CLK (Bit 5)	
Bit	**Efecto**
0	Deshabilita el circuito del cristal Oscilador. Los parámetros RCFAST y RCSLOW de la declaración _CLKMODE configura OSCENA de esta forma
1	Habilita el circuito del cristal oscilador para que la señal de reloj pueda ingresar a XIN o para que XIN y XOUT puedan funcionar juntos. Los parámetros XINPUT y XTALx de la declaración _CLKMODE configura OSCENA de esta forma. Los bits OSCMx seleccionan el modo de operación del circuito del cristal oscilador. Note que no hay resistencias externas o capacitares. Permita a un cristal o resonador 10ms para estabilizar antes de cambiar a un cristal oscilador o salida de reloj PLL vía los bits CLKSELx . Cuando habilita el circuito del cristal oscilador el reloj PLL puede estar habilitado al mismo tiempo y así pueden compartir el periodo de estabilización.

Tabla 1-9: OSCMx del Registro CLK (Bits 4:3)					
OSCMx		**_CLKMODE Parámetros**	**XOUT Resistencia**	**XIN/XOUT Capacitancia**	**Rango de Frecuencia**
1	**0**				
0	0	XINPUT	Infinita	6 pF (solo pad)	DC a 80 MHz Entrada
0	1	XTAL1	2000 Ω	36 pF	4 a 16 MHz Cristal/Resonador
1	0	XTAL2	1000 Ω	26 pF	8 a 32 MHz Cristal/Resonador
1	1	XTAL3	500 Ω	16 pF	20 a 60 MHz Cristal/Resonador

Tabla 1-10: CLKSELx del Registro CLK (Bits 2:0)						
CLKSELx			_CLKMODE Parámetros	Reloj Maestro	Fuente	Notas
2	1	0				
0	0	0	RCFAST	~12 MHz	Interno	Sin partes externas. Rango de 8 MHz a 20 MHz.
0	0	1	RCSLOW	~20 kHz	Interno	Baja potencia. Sin partes externas. Rango de 13 kHz a 33 kHz.
0	1	0	XINPUT	XIN	OSC	OSCENA debe ser '1'.
0	1	1	XTALx y PLL1X	XIN x 1	OSC+PLL	OSCENA y PLLENA deben ser '1'.
1	0	0	XTALx y PLL2X	XIN x 2	OSC+PLL	OSCENA y PLLENA deben ser '1'.
1	0	1	XTALx y PLL4X	XIN x 4	OSC+PLL	OSCENA y PLLENA deben ser '1'.
1	1	0	XTALx y PLL8X	XIN x 8	OSC+PLL	OSCENA y PLLENA deben ser '1'.
1	1	1	XTALx y PLL16X	XIN x 16	OSC+PLL	OSCENA y PLLENA deben ser '1'.

Seguros

Hay ocho bits “seguros” (conocidos como semáforos) disponibles para facilitar el acceso exclusivo de recursos entre múltiples cogs. Si un bloque de memoria se usa por dos o mas cogs a la vez y ese bloque consiste en mas de un long (4 bytes), los cogs tendrán que leer y escribir varias veces para obtener o actualizar el bloque de memoria. Esto lleva a la posibilidad de contenciones de lecto/escritura en ese bloque donde un cog podrá estar escribiendo mientras otro esta leyendo, resultando en lecturas o escrituras incorrectas.

Los seguros son bits globales que accesan a través del hub vía instrucciones de hub: LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR. Debido a que los seguros se accesan solo a través del hub solo un cog a la vez puede afectarlos haciendo esto un mecanismo efectivo de control. El Hub mantiene un inventario de cuales seguros se usan así como sus estados actuales y los cogs pueden verificarlos, regresar, activar y limpiar según lo necesite en su tiempo. Ver LOCKNEW, 126; LOCKRET, 129; LOCKSET, 130; y LOCKCLR, 124 para mayor información.

Memoria Principal

La memoria principal es un bloque de 64 KBytes (16 K largos) que accesan los cogs como recurso mutuamente exclusivo a través del hub. Son 32 KB de RAM y 32 KB de ROM. Los

32 KB de RAM principal es de propósito general y es el destino de la aplicación Propeller ya sea descargándolo de un host o subiéndolo de una memoria externa EEPROM de 32 KB. Los 32 KB de ROM principal incluye el código y datos de recursos vitales para las funciones del Propeller: definición de caracteres, Log, Anti-Log y tablas de seno así como el cargador de inicio y el interprete Spin. La organización de la memoria se muestra en la Figura 1-5.

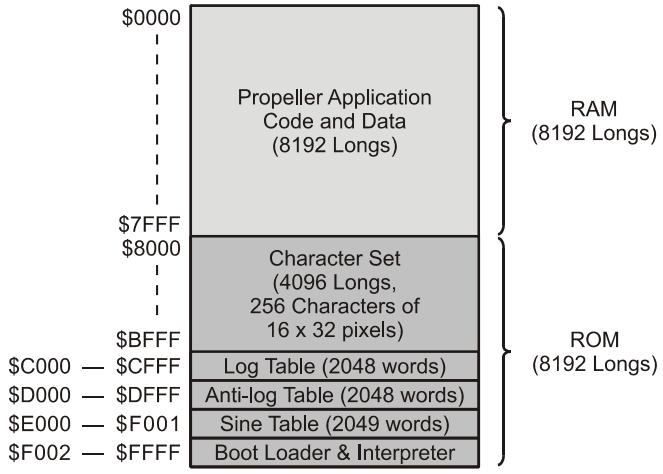


Figura 1-5: Mapa de Memoria Principal

RAM Principal

La primera mitad de la memoria principal es RAM. Este espacio se usa para el programa, datos, variables y pila(s); también conocida como la aplicación Propeller.

Cuando un programa se carga en el chip ya sea de un host o de EEPROM externa este espacio se escribe completamente. Las primeras 16 localidades, $0000 – $000F, tienen la inicialización de datos usados por el cargador de inicio e interprete. El código de programa ejecutable y los datos comenzaran en $0010 y se extenderá por algunos largos. El área posterior al código ejecutable se extiende hasta $7FFF, se usa como variable y pila.

Hay dos valores almacenados en el área de inicialización que pueden ser de interés al programa: un largo en $0000 contiene la frecuencia del reloj maestro inicial en Hertz y un byte siguiéndolo en $0004 contiene el valor inicial escrito en el registro CLK. Estos dos valores pueden ser leídos/escritos usando su dirección física (`LONG[$0]` y `BYTE[$4]`) y pueden leerse usando sus nombres predefinidos (**`CLKFREQ`** y **`CLKMODE`**). Si cambias el registro CLK sin usar el comando **`CLOCKSET`**, tendrás que actualizar estas dos localidades para que los objetos a los cuales hace referencia tengan la información actual.

ROM Principal

La segunda mitad de la memoria principal es ROM. Este espacio se usa para definición de caracteres, funciones matemáticas, el cargador de inicio y el interprete Spin.

Definiciones de Caracteres

La primera mitad de la ROM esta dedicada a un set de 256 caracteres. Cada definición de carácter es de 16 pixeles de ancho por 32 pixeles de altura. Estas definiciones de caracteres pueden usarse para desplegado de video, gráficos de LCD, impresión, etc. El set de caracteres se basa en la base de Norte America / Europa del este (Latín Básico y suplementos Latín-1), con muchos caracteres especiales insertados. Los caracteres especiales son conexiones de forma de onda y esquemáticos de construcción de bloques, símbolos griegos comúnmente usados en electrónica y varias flechas y punteros.

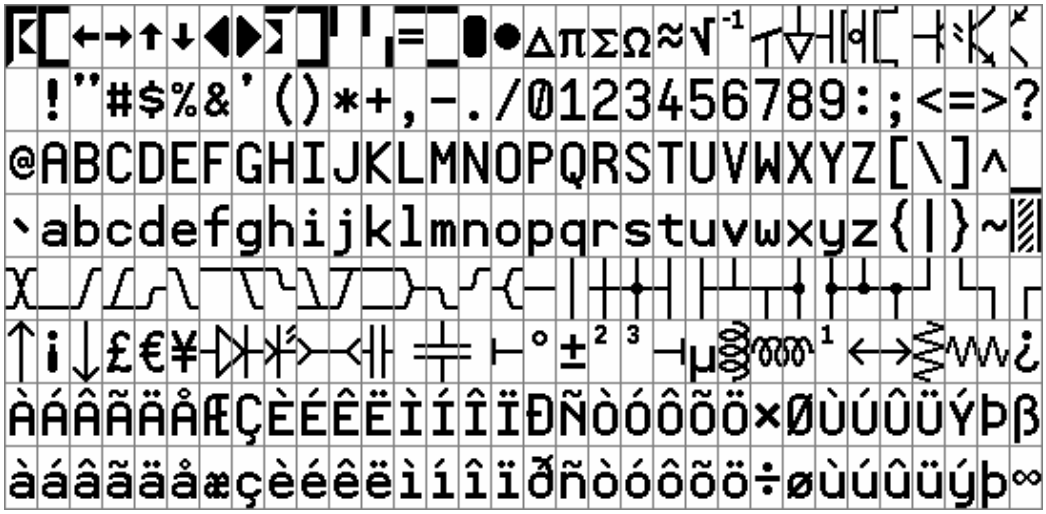


Figura 1-6: Fuente de Caracteres Propeller

La definición de caracteres están numerados de 0 a 255 de izquierda a derecha, de arriba abajo en la Figura 1-6. En ROM están acomodados con cada par de adyacentes par-impar caracteres mezclados para formar 32 longs. El primer par de caracteres esta localizado en los bytes $8000-$807F. El Segundo en $8080-$80FF, y así sucesivamente hasta el ultimo par $BF80-$BFFF. La herramienta propeller incluye un mapa de caracteres interactivo (Help → View Character Chart…) esta tiene un mapa de bit de la ROM el cual muestra donde y como reside cada carácter en ROM.

Los pares de caracteres se mezclan renglón por renglón de tal forma que cada 16 pixeles horizontales están espaciados con sus vecinos para que el carácter par tome los bits 0, 2, 4, ...30, y el carácter impar toma los bits 1, 3, 5, ...31. Los pixeles de la izquierda extrema están en los bits mas bajos mientras que los pixeles de la extrema derecha están en los bits mas altos como se muestra en la Figura 1-7. Esto forma un long (4 bytes) para cada renglón de pixeles en el par de caracteres. 32 de estos largos forman el renglón superior y así hasta el inferior logrando la definición completa de los caracteres mezclados. Las definiciones son codificadas de esta forma para que el hardware de video del cog pueda manejar los longs directamente, usando selección de color para desplegar los caracteres pares o impares. También tiene las ventajas de permitir pares de caracteres en tiempo real (ver siguiente párrafo) que son caracteres de cuatro colores usados para dibujar botones biselados, líneas e indicadores de enfoque.

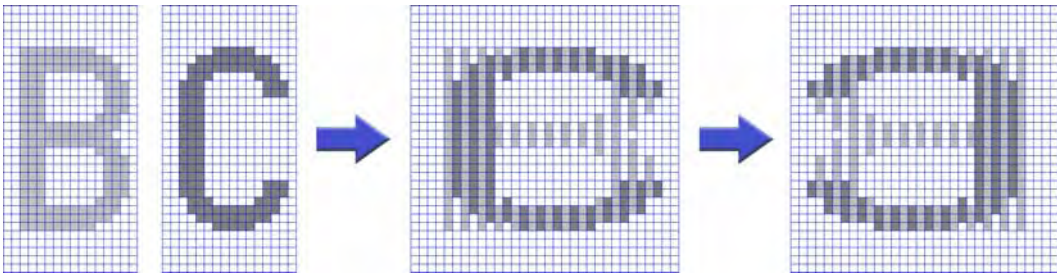


Figura 1-7: Intercalado de caracteres Propeller

Algunos códigos de caracteres tienen significados obvios tale como el 9 para Tab, 10 para línea y 13 para salto de renglón. Estos códigos de caracteres citan acciones y no son iguales a la definición de caracteres. Por esta razón, las definiciones de caracteres se han utilizado para los caracteres especiales de cuatro colores. Estos caracteres de cuatro colores son utilizados para dibujar líneas de cajas 3-D y son implementados como 16 x 16 pixeles contrario a los 16 x 32. Ellos ocupan pares de caracteres par-impar 0-1, 8-9, 10-11, y 12-13. La Figura 1-8 muestra un ejemplo de un botón con biselado 3D hecho de alguno de estos caracteres.

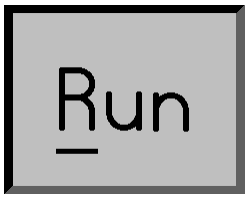


Figura 1-8: Botón con bordes biselados 3-D

La herramienta Propeller incluye y usa la fuente Parallax True Type® la cual sigue los diseños de la fuente Propeller integrada en el hardware. Con esta fuente y la herramienta Propeller usted puede incluir esquemáticos, diagramas de tiempo y otros diagramas justo en el código fuente de la aplicación.

Tablas Log y Anti-Log

Las tablas de Log y Anti-Log son útiles para convertir valores entre su forma numérica y su forma exponencial.

Cuando hay números codificados en forma exponencial las operaciones matemáticas toman efectos más complejos. Por ejemplo sumar y restar resulta en multiplicar y dividir, corrimiento a la izquierda se convierte en cuadrado y corrimiento a la derecha en raíz cuadrada, dividir por 3 produce raíz cúbica. Una vez que el exponente se convierte a un número el resultado es aparente.

Ver Apéndice B: Ejemplos Matemáticos y Tablas en pagina 397.

Tablas de Seno

La tabla de senos proporciona 2,049 ejemplos de seno de 16-bit desde 0° a 90°, (resolución de 0.0439°). Los valores de seno para todos los cuadrantes cubriendo > 90° a < 360° pueden calcularse de simples transformaciones de esta tabla. La tabla de seno puede usarse para cálculos relacionados con fenómenos angulares.

Ver Apéndice B: Ejemplos Matemáticos y Tablas en pagina 397.

Gestor de arranque e Interprete Spin

La última sección en la ROM principal contiene el cargador de arranque del chip Propeller y el intérprete de programas Spin.

El cargador de inicio es responsable de inicializar el Propeller una vez encendido o reseteado. Cuando un encendido se inicia el cargador se mueve a la RAM del cog 0 y el cog ejecuta el código iniciando en la localidad 0. El cargador de inicio verifica la comunicación con el host y EEPROM para cargar/descargar código/datos, procesa la información y finalmente inicia el interprete spin en la RAM del cog 0 (sobrescribiéndose) para correr la aplicación Propeller del usuario, o pone al Propeller en modo de apagado. Ver el Procedimiento de Inicio en pagina 18.

El interprete spin obtiene y ejecuta la aplicación Propeller desde la RAM principal. Esto puede guiar a iniciar cogs adicionales para correr mas código spin o código ensamblador Propeller según lo requiera la aplicación. Ver Procedimiento , pagina 18.

Capítulo 2: Referencia del Lenguaje Spin

Este capitulo describe todos los elementos del lenguaje Spin del chip Propeller y se puede usar como mejor referencia para elementos individuales de lenguaje Spin. Para un tutorial del uso del lenguaje use el tutorial de Lenguaje Spin de las herramientas en línea y luego regrese a esta parte para mas detalles.

La referencia del Lenguaje Spin se divide en tres secciones:

1) **La estructura de los Objetos Propeller**. Los objetos Propeller consisten en código Spin, ensamblador Propeller opcional y datos. Un objeto en código Spin se proporciona con una estructura que consiste de bloques de propósito especial. Esta sección enlista estos bloques y los elementos que pueden usarse en cada uno. Cada elemento listado tiene una página de referencia para más información.

2) **La lista Categórica del Lenguaje Spin Propeller** . Todos los elementos incluyendo operadores y símbolos de sintaxis están agrupados por una función relacionada. Esta es una forma de entender el lenguaje y sus características disponibles específicas. Cada elemento tiene una página de referencia para mayor información. Algunos elementos se marcan con un súper script “a” indicando que también están disponibles en ensamblador Propeller aunque la sintaxis puede variar. Estos elementos marcados también se incluyen en el Capitulo 3: Referencia Lenguaje Ensambla.

3) **Los Elementos del Lenguaje Spin** . La mayoría de los elementos tienen su propia sub-seccion dedicada acomodada en orden alfabético para una fácil búsqueda. Los elementos individuales sin una sub-sección dedicada tales como símbolos, y alguna constante están agrupados en otra sub-sección relacionada pero pueden identificarse fácilmente siguiendo las páginas de referencia de la lista categórica.

Estructura de Objetos/Spin

Cada objeto Propeller en un archive Spin que tiene una estructura inherente que consiste en hasta seis bloques diferentes de propósito especial: CON, VAR, OBJ, PUB, PRI, y DAT. Estos bloques se muestran abajo (en el orden que típicamente aparecen en los objetos) junto con el paquete de elementos que se usan en cada uno..

Para ejemplos detallados de la estructura del objeto (Spin) revise el tutorial de programación Propeller en la ayuda de la herramienta Propeller.

CON: Bloques Contantes definen constantes globales (Pág. 87).

_CLKFREQ	p 68	NEGX	p 96	PLL16X	p 71	XINPUT	p 71
_CLKMODE	p 71	Operadores*	p 147	POSX	p 96	XTAL1	p 71
_FREE	p 113	PI	p 96	RCFAST	p 71	XTAL2	p 71
_STACK	p 206	PLL1X	p 71	RCSLOW	p 71	XTAL3	p 71
_XINFREQ	p 241	PLL2X	p 71	ROUND	p 202		
FALSE	p 96	PLL4X	p 71	TRUE	p 96		
FLOAT	p 111	PLL8X	p 71	TRUNC	p 213		

*Operadores No Asignados únicamente.

VAR: Bloques Variables definen variables globales (Pág. 215).

BYTE	p 54	LONG	p 132	ROUND	p 202	TRUNC	p 213
FLOAT	p 111	Operadores*	p 147	WORD	p 232		

*Operadores No Asignados únicamente.

OBJ: Bloques de Objetos definen Objetos Referenciados (Pág. 145).

FLOAT	p 111	Operadores*	p 147	ROUND	p 202	TRUNC	p 213

*Operadores No Asignados únicamente.

PUB/PRI: Métodos Públicos y Privados definen rutinas Spin (Pág. 186/185).

DAT: Bloque de Datos definen datos y código ensamblador Propeller (Pág. 102).

*Operadores No Asignados únicamente.

Lista Categórica de Lenguaje Spin Propeller

Los elementos marcados con un súper script “a” también están disponibles en Ensamblador Propeller.

Asignación de Bloques

CON	Declara un Bloque Constante; p 87.
VAR	Declara un Bloque Variable; p 215.
OBJ	Declara un Bloque de Referencia de Objeto; p 145.
PUB	Declara un Bloque de Método Publico; p 186.
PRI	Declara un Bloque de Método Privado; p 185.
DAT	Declara un Bloque de Datos; p 102.

Configuración

CHIPVER	Número de versión del Chip Propeller; p 65.
CLKMODE	Ajuste de Modo de Reloj Actual; p 70.
_CLKMODE	Modo de Reloj Definida por aplicación (solo lectura); p 71.
CLKFREQ	Frecuencia Actual del Reloj; p 66.
_CLKFREQ	Frecuencia de Reloj Definida por aplicación (solo lectura); p 68.
CLKSET[a]	Ajuste de Modo y Frecuencia de Reloj; p 74.
_XINFREQ	Frec.Externa de Reloj Definida por aplicación (solo lectura); p 241.
_STACK	Espacio Reservado de pila Definida por aplicación (solo lectura); p 206.
_FREE	Espacio Libre Reservado Definida por aplicación (solo lectura); p 113.
RCFAST	Constante para _CLKMODE: Oscilador Interno Rápido; p 71.
RCSLOW	Constante para _CLKMODE: Oscilador Interno Lento; p 71.
XINPUT	Constante para _CLKMODE: Reloj/Osc Externo (pin XI); p 71.
XTAL1	Constante para _CLKMODE: Cristal Externo Baja Velocidad; p 71.
XTAL2	Constante para _CLKMODE: Cristal Externo Mediana Velocidad; p 71.
XTAL3	Constante para _CLKMODE: Cristal Externo Alta Velocidad; p 71.
PLL1X	Constante para _CLKMODE: Tiempo de Frecuencia Externa 1; p 71.
PLL2X	Constante para _CLKMODE: Tiempo de Frecuencia Externa 2; p 71.
PLL4X	Constant para _CLKMODE: Tiempo de Frecuencia Externa 4; p 71.

PLL8X	Constant para _CLKMODE: Tiempo de Frecuencia Externa 8; p 71.
PLL16X	Constant para _CLKMODE: Tiempo de Frecuencia Externa 16; p 71.

Control de Cog

COGID[a]	Identificación de Cog Actual (0-7); p 78.
COGNEW	Iniciar el Siguiente Cog Disponible; p 81.
COGINIT[a]	Iniciar o Reiniciar el Cog por Identificación; p 79.
COGSTOP[a]	Detener un Cog por Identificación; p 86.
REBOOT	Reiniciar el chip Propeller; p 191.

Control de Proceso

LOCKNEW[a]	Verificar un Nuevo seguro; p 126.
LOCKRET[a]	Desbloquear un seguro; p 129.
LOCKCLR[a]	Limpiar un seguro por Identificación; p 124.
LOCKSET[a]	Activar un seguro por Identificación; p 130.
WAITCNT[a]	Esperar Contador de Sistema para alcanzar un valor; p 223.
WAITPEQ[a]	Esperar por pin(s) a ser igual que un Valor; p 227.
WAITPNE[a]	Esperar por pin(s) a no ser igual que un Valor; p 229.
WAITVID[a]	Espera sincronización Video y entrega otro grupo de pixeles; p 230.

Control de Flujo

IF ...ELSEIF ...ELSEIFNOT ...ELSE	Ejecuta uno o mas Bloques de código Condicionalmente; p 115.
IFNOT ...ELSEIF ...ELSEIFNOT ...ELSE	Ejecuta uno o mas Bloques de código Condicionalmente; p 121.
CASE ...OTHER	Evalúa la expresión y Ejecuta un Bloque de código que satisface una condición; p 62.

REPEAT ...FROM ...TO ...STEP ...UNTIL ...WHILE	Ejecuta un Bloque de código repetidamente un numero finito o infinito con opción a contador opcional, intervalos, salidas y condiciones continuas; p 192.
NEXT	Salta el resto de bloque REPEAT hasta la siguiente instrucción; p 144.
QUIT	Salta el ciclo REPEAT; p 190.
RETURN	Sale de PUB/PRI con estatus normal y valor opcional de regreso; p 200.
ABORT	Sale de PUB/PRI abortando y valor opcional de regreso; p 49.

Memoria

BYTE	Declara símbolo byte o acceso byte de memoria principal; p 54.
WORD	Declara símbolo word o acceso word de memoria principal; p 232.
LONG	Declara símbolo long o acceso long de memoria principal; p 132.
BYTEFILL	Llena bytes de memoria principal con un valor; p 60.
WORDFILL	Llena words de memoria principal con un valor; p 239.
LONGFILL	Llena longs de memoria principal con un valor; p 138.
BYTEMOVE	Copia bytes de una región a otra en memoria principal; p 61.
WORDMOVE	Copia words de una región a otra en memoria principal; p 240.
LONGMOVE	Copia longs de una región a otra en memoria principal; p 139.
LOOKUP	Obtiene índice (1..N) de una lista; p 142.
LOOKUPZ	Obtiene índice base cero (0..N−1) de una lista; p 142.
LOOKDOWN	Obtiene índice (1..N) de un valor igual de una lista; p 140.
LOOKDOWNZ	Obtiene índice base cero (0..N−1) de un valor igual de una lista; p 140.
STRSIZE	Obtiene el tamaño de cadena en bytes; p 210.
STRCOMP	Compara una cadena de bytes contra otra cadena de bytes; p 207.

Instrucciones

STRING	Declara expresiones cadena; tiempo de compilación; p 209.
CONSTANT	Declara expresiones constantes; tiempo de compilación; p 94.
FLOAT	Declara expresiones punto flotante; tiempo de compilación; p 111.
ROUND	Redondea expresiones de punto flotante a enteros; p 202.
TRUNC	Abrevia expresiones de punto flotante a decimales; p 213.
FILE	Importa datos de un archivo externo; p 110.

Registros

DIRA[a]	Direcciona registros 32-bit en Puerto A; p 107.
DIRB[a]	Direcciona registros 32-bit en Puerto B (uso futuro); p 107.
INA[a]	Ingresa registros 32-bit en puerto A (solo lectura); p 121.
INB[a]	Ingresa registros 32-bit en puerto B (solo lectura) (uso futuro); p 122.
OUTA[a]	Salida de registros 32-bit Puerto A; p 179.
OUTB[a]	Salida de registros 32-bit Puerto B (uso futuro); p 181.
CNT[a]	32-bit System Counter Register (read only); p 76.
CTRA[a]	Registro de Control Contador A; p 98.
CTRB[a]	Registro de Control Contador B; p 98.
FRQA[a]	Registro de Frecuencia Contador A; p 114.
FRQB[a]	Registro de Frecuencia Contador B; p 114.
PHSA[a]	Registro (PLL) Ciclo Fase Cerrada Contador A; p 184.
PHSB[a]	Registro (PLL) Ciclo Fase Cerrada Contador B; p 184.
VCFG[a]	Registro de configuración de Video; p 218.
VSCL[a]	Registro de Escala de Video; p 221.
PAR[a]	Registro Parámetros de Boot del Cog (solo lectura); p 182.
SPR	Arreglo Registro Propósito Especial; acceso indirecto a Cog; p 204.

Constantes

TRUE[a]	Lógico verdadero: -1 ($FFFFFFFF); p 96.
FALSE[a]	Lógico Falso: 0 ($00000000) ; p 96.
POSX[a]	Máximo entero positivo: 2,147,483,647 ($7FFFFFFF); p 96.
NEGX[a]	Máximo entero negativo: -2,147,483,648 ($80000000); p 96.
PI[a]	Valor de punto flotante para PI: ~3.141593 ($40490FDB); p 96.

Variable

RESULT	Variable de resultado predeterminado para métodos PUB/PRI; p 198.

Operadores Unarios

+	Positivo (+X); forma unaria de suma; p 154.
-	Negación (-X); forma unaria de resta; p 154.
--	Pre-decremento (--X) o post-decremento (X--) y asignar; p 155.
++	Pre-incremento (++X) o post-incremento (X++) y asignar; p 156.
^^	Raíz cuadrada; p 160.
||	Valor Absoluto; p 160.
~	Signo-extendido de bit 7 (~X) o post-limpiar a 0 (X~); p 160.
~~	Signo-extendido de bit 15 (~~X) o post-activar a -1(X~~); p 161.
?	Numero aleatorio hacia adelante (?X) o en reversa (X?); p 163.
|<	Valor decodificado (módulos de 32; 0-31) bit-simple-alto long; p 164.
>|	Codificar long en magnitudes (0 - 32) como prioridad bit-alto; p 164.
!	Bit inteligente: NOT; p 170.
NOT	Booleano: NOT (mueve non-0 a -1); p 172.
@	Símbolo de dirección; p 177.
@@	Dirección de Objeto mas valor de símbolo; p 177.

Operadores Binarios

NOTA: Todos los operadores a la derecha son operadores de asignación.

=	--y--	=	Transferencia de constantes (Bloque CON); p 152.
:=	--y--	:=	Transferencia de variable (Bloque PUB/PRI); p 153.
+	--o--	+=	Suma; p 153.
-	--o--	-=	Resta; p 154.
*	--o--	*=	Multiplica y regresa los 32 bits bajos (firmado); p 157.
**	--o--	**=	Multiplica y regresa los 32 bits altos (firmado); p 157.
/	--o--	/=	División (firmado); p 158.
//	--o--	//=	Módulos (firmado); p 158.
#>	--o--	#>=	Limite mínimo (firmado); p 159.
<#	--o--	<#=	Limite máximo (firmado); p 159.
~>	--o--	~>=	Movimiento aritmético a la derecha; p 162.
<<	--o--	<<=	Bitwise: Movimiento a la izquierda; p 165.
>>	--o--	>>=	Bitwise: Movimiento a la derecha; p 165.
<-	--o--	<-=	Bitwise: Rotación a la izquierda; p 166.
->	--o--	->=	Bitwise: Rotación a la derecha; p 167.
><	--o--	>><=	Bitwise: Reversa; p 167.
&	--o--	&=	Bitwise: AND; p 168.
|	--o--	|=	Bitwise: OR; p 169.
^	--o--	^=	Bitwise: XOR; p 169.
AND	--o--	AND=	Booleano: AND (mueve non-0 a -1); p 171.
OR	--o--	OR=	Booleano: OR (mueve non-0 a -1); p 172.
==	--o--	===	Booleano: Igual a; p 173.
<>	--o--	<>=	Booleano: No igual a; p 174.
<	--o--	<=	Booleano: Menor que (firmado); p 174.
>	--o--	>=	Booleano: Mayor que (firmado); p 175.
=<	--o--	=<=	Booleano: Igual o menor que (firmado); p 175.
=>	--o--	=>=	Booleano: Igual o mayor que (firmado); p 176.

Símbolos sintaxicos

%	Indicador de número binario, como %1010; p 211.
%%	Indicador de numero cuaternario, como %%2130; p 211.
$	Indicador hexadecimal, como $1AF o ensamblador “aquí” ; p 211.
"	Indicador de Cadena "Hello"; p 211.
_	Delimitador grupo en valores constantes, o símbolo guión bajo; p 211.
#	Referencia Objeto-Constante: obj#constant; p 211.
.	Referencia Objeto-Método: obj.method(param) o punto decimal; p 211.
..	Indicador de rango, como 0..7; p 211.
:	Separador: PUB method : sym, o asignador de objeto, etc.; p 211.
|	Separador de Variable Local: PUB method | temp, str; p 212.
\	Aborta trampa, como \method(parameters); p 212.
,	Delimita lista, como method(param1, param2, param3); p 212.
( )	Designa parámetros de lista, como method(parameters); p 212.
[ ]	Designa índice de arreglo, como INA[2]; p 212.
{ }	Designa comentarios de código en línea/multilínea; p 212.
{{ }}	Designa comentarios de documento en línea/multilínea; p 212.
'	Designa comentario de código; p 212.
''	Designa comentario de documento; p 212.

Elementos de Lenguaje Spin

El resto de este capitulo describe los elementos del lenguaje spin, mostrados arriba, en orden alfabético. Algunos elementos se explican con el contexto de otros para mayor claridad; use las páginas de referencia de la lista categórica para ver los comentarios. Muchos elementos están disponibles en spin y ensamblador Propeller. Esos elementos se describen en detalle en esta sección, con referencias y diferencias marcadas en las áreas apropiadas del Capitulo 3: Referencia Lenguaje Ensambla comenzando en la pagina 243.

Reglas de Símbolos

Los símbolos son sensitivos a mayúsculas, nombres alfanumericos creados por el compilador (palabra reservada) o por el desarrollador del código (palabra definida por usuario). Representan valores (constantes o variables) para asegurar la facilidad del código fuente y mantenerlo. Los símbolos deben seguir las siguientes reglas:

1) Comenzar con una letra (a – z) o guión bajo '_'.
2) Contener solo letras, números o guión bajo (a – z, 0 – 9, _); no se permite espacios.
3) Debe ser de 30 caracteres o menos.
4) Único a objeto, no es palabra reservada (p. 395) o símbolo previo definido por usuario.

Representaciones de Valores

Los valores pueden ingresarse en binario (base-2), cuaternario (base-4), decimal (base-10), hexadecimal (base-16), o formato de caracteres. Valores numéricos pueden usar guión bajo, '_', como separador de grupo para clarificar. Vea los ejemplos siguientes de los formatos.

Tabla 2-1: Representación de Valores

Base	Tipo de Valor	Ejemplos				
2	Binario	%1010	–o–	%11110000_10101100		
4	Cuaternario	%%2130_3311	–o–	%%3311_2301_1012		
10	Decimal (entero)	1024	–o–	2_147_483_647	–o–	-25
10	Decimal punto flotante	1e6	–o–	1.000_005	–o–	-0.70712
16	Hexadecimal	$1AF	–o–	$FFAF_126D_8755		
n/a	Caracter	"A"				

Se pueden usar separadores en vez de comas (en valores decimales) o para formar grupos lógicos tales como nibbles, bytes, words, etc.

Definiciones Sintaxicas

Además de las descripciones detalladas, las siguientes páginas contienen definiciones de sintaxis para elementos y describe en términos cortos las opciones del elemento. Las definiciones de sintaxis usan símbolos especiales para indicar cuando y como se usan.

NEGRITAS	Negrita mayúsculas deben escribirse como se muestra.
Negritas Itálicas	Negritas itálicas deben remplazarse por texto de usuario, símbolos, operadores, expresiones, etc.
. .. : , # | \ [ ] ( )	Puntos, doble punto, dos puntos, comas, signo de números, diagonales, corchetes, paréntesis, deben escribirse como se muestra.
〈 〉	Corchetes de ángulo encierran puntos opcionales. Ingrese el punto encerrado. No ingrese los corchetes de ángulo.
(( ))	Símbolos de doble paréntesis encierran puntos mutuamente exclusivos, separados por una barra-guión. Ingrese uno y solo uno de los puntos codificados. No ingrese el doble paréntesis o la barra-guión..
…	símbolo de repetición indica que el punto anterior, o grupo puede repetirse varias veces. Repite el ultimo punto so lo desea. No ingrese el símbolo de repetición.
↳	Símbolo de nueva línea / Indentacion los siguientes puntos deben aparecer en otra línea, indenta al menos un espacio.
→|	Símbolo de indentacion indica que los siguientes puntos deberán estar indentados al menos por un espacio.
Single line	Separa varias opciones de sintaxis.
Double line	Separa instrucciones del valor de regreso.

Como los elementos son limitados a bloques específicos de spin, todas las definiciones de sintaxis comienzan con un indicador del tipo de bloque requerido. Por ejemplo, la siguiente sintaxis indica que el comando BYTEFILL y sus parámetros deben aparecer en el bloque PUB o PRI, pero puede ser uno de muchos comandos en ese bloque.

((PUB PRI))
  BYTEFILL (*StartAddress*, *Value*, *Count*)

ABORT

Instrucción: Sale de un método PUB/PRI abortando con un *Valor* opcional de regreso.

((PUB PRI))
ABORT ⟨*Valor*⟩

Regresa: Ya sea en RESULT actual, o *Valor* si se proporciona.

- *Valor* es una expresión opcional y se regresa cuando se aborta un método PUB o PRI.

Explicación

ABORT es una de dos instrucciones (ABORT y RETURN) que termina la ejecución de un método PUB o PRI.

ABORT causa un regreso de un método PUB o PRI a través del estado de aborto; significa que la llamada de pila es "jalada" repetidamente hasta que la pila esta vacía o alcanza una llamada con una trampa de abortar (\ ) y entrega un valor en el proceso.

ABORT es útil para casos donde un método necesita terminar e indicar un estado anormal o elevado al llamador o un llamador previo. Por ejemplo una aplicación puede ser cuando esta envuelta en una cadena complicada de eventos donde cualquiera de esos eventos podría llevar a una subdivisión diferente de la cadena o a una acción de decisión final. Puede ser más sencillo escribir la aplicación usando métodos pequeños y especiales que se llaman en una forma anidada, cada uno manejando diferentes eventos en la cadena. Cuando uno de esos métodos simples determina un curso de acción puede enviar un "abort" que colapsa completamente la cadena anidada y previene que continúen todos los métodos intermedios.

Cuando ABORT aparece sin *Valor* opcional regresa el valor actual de la variable RESULT del método PUB/PRI. Si se ingreso un *Valor* entonces cuando se aborta el método PUB o PRI regresa ese *Valor*.

Acerca de la llamada de Pila

Cuando se llaman métodos simplemente para referenciar a ellos desde otros métodos, debe existir un mecanismo para almacenar el lugar a donde regresar una vez que el método se completa. El mecanismo se llama pila pero aquí usaremos el termino "llamada de pila". Es simplemente memoria RAM usada para almacenar la dirección de regreso, valores de regreso, parámetros y resultados intermedios. Mientras mas y mas métodos se llamen la llamada de pila lógicamente crecerá. Mientras mas y mas métodos regresen de (a través de

RETURN o alcanzando el final del método) la pila de llamada disminuirá. A esto se le llama "empujar" a la pila y "jalar" de la pila respectivamente.

La instrucción RETURN jala la información mas reciente de la pila de llamada para facilitar el regreso al llamador inmediato; el que directamente llamo al método a donde regresara. La instrucción ABORT, sin embargo repetidamente jalara datos de la pila de llamada hasta que alcance un llamador con una trampa de abortar (ver abajo); regresando a algún llamador de nivel mas alto que puede tener solo una llamada o muchas llamadas hasta la anidación de una cadena de llamadas. Cualquier punto de regreso en el camino entre el método abortar y la trampa de método abortar se ignoran y esencialmente terminadas. De esta forma ABORT permite al código salir de un profundo y potencialmente complicada serie de lógicas complicadas que llevaran a un serio problema para poder manejar a un alto nivel.

Usando ABORT

Cualquier método puede proporcionar una instrucción ABORT. Es hasta el más alto nivel de código para verificar por un estado de abortar y manejarlo. El mas alto nivel de código puede ser ya sea el que llamo un método abort directamente o a través de algún otro grupo de métodos. Para proporcionar una instrucción ABORT, use algo como:

```
if <condición mala>
  abort                 'Una condición mala se detecta,
                             aborte
```

—o—

```
if <condición mala>
  abort <valor>         'Una condición mala se detecta,
                             aborte con un valor
```

...donde <mala condición> es una condición que determina si el método debe abortar y <valor> es un valor de regreso al abortar.

La Trampa Abort (\ )

Para atrapar un ABORT, la llamada al método o cadena de metodos que podría potencialmente abortar debe preceder con la trampa abortar, que es el símbolo diagonal invertida (\). Por ejemplos un método llamado MayAbort podría abortar posiblemente, o si sus llamadas a otros métodos pueden abortar, una llamada a método podrías atrapar esto con:

```
if \MayAbort            'Llama MayAbort con trampa abort
  abort <valor>         'Procesa abort
```

El tipo de salida que usa MayAbort, **ABORT** o **RETURN**, no es conocido automáticamente por la llamada de la trampa; podría suceder para ser el destino de una instrucción **RETURN**. Por lo tanto el código debe escribirse de tal forma que detecte que tipo se utilizo. Algunas posibilidades son: 1) El código puede ser designado para que el método de alto nivel es el único lugar donde se atrape un aborto y otro código de nivel medio procese las cosas normalmente sin permitir la propagación del **RETURN** mas alto, o 2) Abortando métodos puede regresar un valor especial que no puede ocurrir en una circunstancia normal, o 3) Una bandera global puede activarse por el método abort antes de abortar.

Ejemplos de uso de Abort

El siguiente es un ejemplo de una aplicación de un robot que se diseño para moverse lejos de un objeto que detecta con sus cuatro sensores (Izquierda, Derecha, Frente, Atrás). Asuma que CheckSensors, Beep, y MotorStuck son métodos definidos en otro lado.

```
CON
  #0, None, Left, Right, Front, Back  'Detalles de direccion

PUB Main | Direction
  Direction := None
  repeat
    case CheckSensors                 'Activa sensor
      Left  : Direction := Right      'Objecto a Izq? Mover a la
derecha
      Right : Direction := Left       'Objecto a Der? Mover a la
izquierda
      Front : Direction := Back       'Objecto enfrente? Mover hacia
atras
      Back  : Direction := Front      'Objecto atras? Mover al frente
      other : Direction := None       'Si no quedarse quieto
    if not \Move(Direction)           'Mover robot
      Beep                            'Detenido?  Bocina

PUB Move(Direction)
  result := TRUE                      'Asume exito
  if Direction == None
    return                            'Regresa si no hay dirección
  repeat 1000
    DriveMotors(Direction)            'Mueve motor 1000 veces
```

```
PUB DriveMotors(Direction)
  <código para manejar motores>
  if MotorStuck
    abort FALSE                         'Si el motor se detiene,
abortar
  <mas código>
```

El ejemplo de arriba muestra tres métodos de varios niveles lógicos, Main (“alto nivel”), Move (“nivel medio”) y DriveMotors (“bajo nivel”). El método de alto nivel, Main, es el que toma la decisión de la aplicación decidiendo como responder a los eventos como la activación de los sensores y movimientos del motor. El método de nivel medio, Move, es responsable de mover el robot una distancia corta. El método de bajo nivel, DriveMotors, maneja los detalles de control de los motores apropiadamente y verifica que se movieron correctamente.

En una aplicación como esta pueden ocurrir eventos críticos en el código de bajo nivel que necesitan direccionarse por el código de alto nivel. El comando ABORT puede ser de ayuda obteniendo el mensaje del código de alto nivel sin requerir el pase de mensajes complicados a través del código de nivel medio. En este caso solo tenemos un método de medio nivel pero podrían existir muchos códigos anidados de medio nivel entre el alto y bajo nivel.

El método Main tiene entradas de sensores y decide en que dirección mover el robot a través de la sentencia CASE. Entonces llama a Move de una forma especial con el símbolo de trampa de abortar, \ , precediéndolo. El método Move activa RESULT a TRUE y llama a un ciclo finito DriveMotors. Si se completa adecuadamente, Move regresa TRUE. El método DriveMotors maneja la forma de mover los motores del robot para lograr la dirección deseada, pero si determina que los motores están atorados y no se puede mover aborta con un valor FALSE. De otra forma simplemente regresa normalmente.

Si todo esta bien el método DriveMotors regresa normalmente, el método Move se mueve normalmente y eventualmente regresa un TRUE, y el método Main continua normalmente. Sin embargo si DriveMotors encuentra un problema ABORTA lo cual hace que el Propeller llame a la pila por todo el método Move hasta llegar al método Main donde se encuentra la trampa de abortar. El método Move esta completamente inconsciente de esto y se termina efectivamente. El método Main verifica el valor regresado por la llamada a Move (el cual es ahora un valor FALSE que se regreso el método abortado DriveMotors dentro de la llamada de pila) y decide generar un Beep como resultado de la falla.

Si no se hubiera puesto la trampa (\ ) frente a la llamada a Move cuando DriveMotors aborto la llamada de pila hubiera jalado toda la información hasta vaciarse y esta aplicación terminaría inmediatamente.

BYTE

instrucción: Declara un símbolo de tamaño byte, datos de tamaño alineado a byte, o lecto escritura de un byte en memoria principal.

```
VAR
  BYTE Symbol {[Count]}
```

```
DAT
  {Symbol} BYTE Data {[Count]}
```

```
((PUB   PRI))
  BYTE [BaseAddress] {[Offset]}
```

```
((PUB   PRI))
  Symbol.BYTE {[Offset]}
```

- *Symbol* es el nombre deseado para la variable (Sintaxis 1) o bloque de datos (Sintaxis 2) o el nombre existente de la variable (Sintaxis 4).
- *Count* es una expresión opcional indicando el nombre de elementos de tamaño byte para *Symbol* (Sintaxis 1), o el numero de entradas de *Data* de tamaño byte (Sintaxis 2) para almacenar en una tabla de datos .
- *Data* es una expresión constante o lista de coma separada de expresiones constantes. Extractos de cadenas de caracteres también se permiten; son tratados como lista de caracteres de coma separada.
- *BaseAddress* es una expresión que describe la dirección de la memoria principal para leer o escribir. Si se omite *Offset*, *BaseAddress* es la dirección actual para operar. Si se especifica *Offset*, *BaseAddress* + *Offset* es la dirección para operar.
- *Offset* es una expresión opcional que indica el balance de *BaseAddress* para operar, o el balance desde el byte 0 de *Symbol*.

Explicación

BYTE es una de tres declaraciones multipropósito (BYTE, WORD, y LONG) que opera o declara en memoria. BYTE puede usarse para:

1) declarar un símbolo de tamaño byte (8-bit) o un arreglo multi byte en un bloque VAR, o
2) declarar un byte alineado, y posiblemente de tamaño byte, datos en el bloque DAT, o
3) leer-escribir un byte a memoria principal en la dirección base con balance opcional, o
4) accesar un byte dentro de una variable de tamaño word o long.

Rango de Byte

La memoria que es tamaño byte (8 bits) puede contener un valor que es una posible combinación de 2^8 bits (Ej., uno de 256 combinaciones). Esto le da a los valores de byte un rango de 0-255. Como el lenguaje Spin desarrolla todas las operaciones matemáticas usando 32-bit, cualquier valor del tamaño byte se considera internamente como valor positivo de tamaño largo. Sin embargo el valor numérico actual contenido en un byte esta sujeto a como la computadora y el usuario lo interpretan. Por ejemplo, puedes escoger usar el operador Signo-Extendido 7 (~), Pág. 160, en una expresión para convertir un valor de byte que interpretas como "firmado" (-128 a +127) a un valor long firmado.

Declaración de la variable Byte (Sintaxis 1)

En bloques VAR, la sintaxis 1 de BYTE se usa para declarar global, variables simbólicas que son tamaño byte o un arreglo de bytes.

Por ejemplo:

```
VAR
  byte  Temp                  'Temp es un byte
  byte  Str[25]               'Str es un arreglo byte
```

El ejemplo de arriba declara dos variables (símbolos), Temp y Str. Temp es solo una variable de tamaño byte. La línea debajo de la declaración Temp usa el campo opcional *Count* para crear un arreglo de 25 variables de tamaño byte llamado Str. Ambos Temp y Str pueden accesarse de cualquier método PUB o PRI dentro del mismo objeto donde este bloque VAR se declaro; son globales al objeto. Un ejemplo de esto se muestra a continuación.

```
PUB SomeMethod
  Temp := 250                 'Activa Temp a 250
  Str[0] := "A"               'Activa el primer elemento de Str a "A"
  Str[1] := "B"               'Activa el Segundo elemento de Str a "B"
  Str[24] := "C"              'Activa el ultimo elemento de Str a "C"
```

Para mayor información de como usar BYTE de esta forma vea la sección VAR Declaraciones Variables (Sintaxis 1) en Pág. 215, y tenga en cuenta que BYTE se usa para el campo *Size* en esa descripción.

Declaración de Datos Byte (Sintaxis 2)

En bloques DAT, la sintaxis 2 de BYTE se usa para declarar alineación de byte, y/o datos de tamaño byte que es compilado como valores constantes en memoria principal. Los bloques DAT permiten a esta declaración tener un símbolo opcional que lo preceda, el cual puede ser usado para referencia posterior (Ver DAT, Pág. 102). Por ejemplo:

```
DAT
  MyData      byte  64, $AA, 55     'Byte alineado y datos tamaño
byte
  MyString    byte  "Hello",0       'Una cadena de bytes (caracteres)
```

Los ejemplos mencionados declaran dos símbolos de datos, MyData y MyString. Cada símbolo de datos apunta al inicio del byte alineado y datos de tamaño byte en memoria principal. Los valores de MyData en memoria principal son 64, $AA y 55, respectivamente. Los valores de MyString en memoria principal son "H", "e", "l", "l", "o", y 0, respectivamente. Este dato esta compilado en el objeto y aplicación resultante como parte de la sección del código ejecutable y puede accesarse usando la forma lecto/escritura, sintaxis 3, de **BYTE** (ver abajo). Para mas información acerca de cómo usar **BYTE** de esta firma vea la sección **DAT** de Declaración de Data(Sintaxis 1) en Pág. 103, y tenga en cuenta que **BYTE** se usa para el campo *Size* en esa descripción.

Los datos pueden repetirse usando el campo opcional *Count*. Por ejemplo:

```
DAT
  MyData    byte 64. $AA[8], 55
```

El ejemplo anterior declara un byte alineado, una tabla de datos tamaño byte llamada MyData, que consiste en los siguientes diez valores: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA, $AA, 55. Hay 8 repeticiones de $AA debido al [8] en la declaración inmediata después el.

Leyendo/Escribiendo Bytes de Memoria Principal (Sintaxis 3)

En bloques **PUB** y **PRI**, la sintaxis 3 de **BYTE** se usa para leer o escribir valores de tamaño byte en memoria principal. Esto se hace escribiendo expresiones que hacen referencia a la memoria principal usando la forma: byte[*BaseAddress*][*Offset*]. Un ejemplo.

```
PUB MemTest | Temp
  Temp := byte[@MyData][1]           'Lee valor byte
  byte[@MyStr][0] := "M"             'Escribe valor byte

DAT
  MyData   byte  64, $AA, 55         'Dato de tamaño byte alineado
  MyStr    byte  "Hello", 0          'Una cadena de bytes
(caracteres)
```

En este ejemplo el bloque **DAT** (parte baja del código) coloca la información en memoria como muestra la Figura 2-1. El primer elemento de MyData se coloca en la dirección de memoria $18. El ultimo elemento de MyData se coloca en la dirección $1A, con el primer

elemento de MyStr seguido inmediatamente a $1B. Note que la dirección de inicio ($18) es arbitraria y puede cambiar según se modifique el código o si el objeto mismo se incluye en otra aplicación.

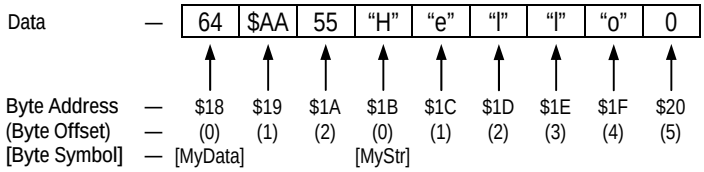


Figura 2-1: Estructura y dirección de datos tamaño byte en memoria principal

Al inicio del código la primer línea ejecutable del método MemTest, Temp := byte[@MyData][1], lee un valor de tamaño byte de la memoria principal. Activa la variable local Temp a $AA; el primer valor leído de la dirección $19 de la memoria principal. La dirección $19 se determina por la dirección del símbolo MyData ($18) mas el byte de balance 1. La siguiente simplificación progresiva lo demuestra.

```
byte[@MyData][1] ➡ byte[$18][1] ➡ byte[$18 + 1] ➡ byte[$19]
```

La siguiente línea, byte[@MyStr][0] := "M", escribe un valor de tamaño byte en la memoria principal. Activa el valor a la dirección de memoria principal $1B el caracter "M." La dirección $1B se calcula con la dirección de MyStr ($1B) mas el byte de balance 0.

```
byte[@MyStr][0] ➡ byte[$1B][0] ➡ byte[$1B + 0] ➡ byte[$1B]
```

Direccionando Memoria Principal

Como se muestra en la Figura 2-1, la memoria principal es realmente solo un grupo de bytes continuos y las direcciones son calculadas en forma de bytes. Este concepto es un sistema confiable para cualquier comando que use direcciones.

La memoria principal es finalmente direccionada en términos de bytes sin importar el tamaño del valor que se esta accesando; byte, word, o long. Esto es una ventaja cuando piensas acerca de cuantos Bytes, Word y Longs se relacionan entre si, pero puede ser problemático cuando se piensa en múltiples eventos de un tamaño sencillo como word o long. Ver las discusiones de Sintaxis 3 de WORD (Pág. 234) y LONG (Pág. 134) para ejemplos de accesos de Word y Long.

Para mayor explicación de como se acomodan los datos en memoria, ver la sección de DAT Declaración de Data(Sintaxis 1) en Pág. 103.

Una Referencia de Memoria Alternativa

Hay aun otra forma de accesar los datos del código mostrado arriba; usted puede referenciar los símbolos de datos directamente. Por ejemplo, esta sentencia lee el primer byte de la lista MyData:

```
Temp := MyData[0]
```

Y estas sentencias leen el Segundo y tercer byte de MyData:

```
Temp := MyData[1]
Temp := MyData[2]
```

Otro Fenómeno de Direccionamiento

Ambas técnicas BYTE y referencia de símbolo directo demostradas pueden usarse para accesar cualquier localidad en memoria principal sin importar como se relaciona a los datos definidos, aquí se muestran unos ejemplos:

```
Temp := byte[@MyStr][-1]   'Lee ultimo Byte de MyData (antes MyStr)
Temp := byte[@MyData][3]   'Lee primer Byte de MyStr (despues MyData)
Temp := MyStr[-3]          'Lee primer byte de MyData
Temp := MyData[-2]         'Lee el byte que esta dos bytes antes de
MyData
```

Estos ejemplos van mas allá de las fronteras lógicas (punto inicial y punto final) de la lista de datos a los que referencia. Puede ser un truco útil, pero con mayor frecuencia se hace por error; sea cuidadoso al direccionar memoria, especialmente si esta escribiendo a esa memoria.

Accesando Bytes de Símbolos de Mayor Tamaño (Sintaxis 4)

En bloques PUB y PRI, la sintaxis 4 de BYTE se usa para leer o escribir componentes variables de tamaño Byte de tamaño Word o tamaño Long. Por ejemplo:

```
VAR
  word  WordVar
  long  LongVar

PUB Main
  WordVar.byte := 0          'Activa el primer byte de WordVar a 0
  WordVar.byte[0] := 0       'Igual que arriba
```

```
  WordVar.byte[1] := 100     'Activa el Segundo byte de WordVar a 100
  LongVar.byte := 25         'Activa el primer byte de LongVar a 25
  LongVar.byte[0] := 25      'Igual que arriba
  LongVar.byte[1] := 50      'Activa el Segundo byte de LongVar a 50
  LongVar.byte[2] := 75      'Activa el tercer byte de LongVar a 75
  LongVar.byte[3] := 100     'Activa el cuarto byte de LongVar a 100
```

Este ejemplo acceso los componentes de tamaño byte de ambos `WordVar` y `LongVar`, individualmente. El comentario indica lo que hace cada línea. Al final del método `Main`, `WordVar` equivale a 25,600 y `LongVar` equivale a 1,682,649,625.

Las mismas técnicas pueden usarse para referenciar datos de símbolos de componentes de tamaño byte de tamaño word o tamaño.

```
PUB Main | Temp
  Temp := MyData.byte[0]            'Lee el byte bajo de MyData word
0
  Temp := MyData.byte[1]            'Lee el byte alto de MyData word
0
  MyList.byte[3] := $12             'Escribe el byte alto de MyList
long 0
  MyList.byte[4] := $FF             'Escribe el byte bajo de MyList
long 1

DAT
  MyData  word  $ABCD, $9988        'dato alineado tamaño word
  MyList  long  $FF998877, $EEEE    'dato alineado tamaño long
```

La primera y segunda línea ejecutable de `Main` lee los valores $CD y $AB, respectivamente de `MyData`. La tercer línea escribe $12 al byte alto del long en el elemento 0 de `MyList`, resultando en un valor de $12998877. La cuarta línea escribe $FF al byte al índice 4 en `MyList` (el byte bajo del long en elemento 1), resultando en un valor de $EEFF.

BYTEFILL

instrucción: Llena los bytes de la memoria principal con un valor.

((PUB PRI))
BYTEFILL (*StartAddress*, *Value*, *Count*)

- *StartAddress* es una expresión indicando la localidad del primer byte de memoria para llenar con un *Value*.
- *Value* es una expresión indicando el valor con el cual llenar los bytes.
- *Count* es una expresión indicando el numero de bytes a llenar, empezando con *StartAddress*.

Explicación

BYTEFILL es una de las tres instrucciones (BYTEFILL, WORDFILL, y LONGFILL) que se usan para llenar bloques de memoria principal con un valor especifico. BYTEFILL llena *Count* bytes de memoria principal con *Value*, empezando en la localidad *StartAddress*.

Usando BYTEFILL

BYTEFILL es una gran forma de limpiar bloques grandes de memoria tamaño byte. Por ejemplo:

```
VAR
  byte   Buff[100]

PUB Main
  bytefill(@Buff, 0, 100)     'Limpia Buff a 0
```

La primera línea del método Main limpia completamente el arreglo de 100 bytes Buff a cero. BYTEFILL es mas rápido en esta tarea que un ciclo dedicado REPEAT.

BYTEMOVE

instrucción: Copia bytes de una región a otra en memoria principal.

((PUB PRI))
BYTEMOVE (*DestAddress*, *SrcAddress*, *Count*)

- *DestAddress* es una expresión especificando la localidad de memoria principal en donde copiar el primer byte de la fuente..
- *SrcAddress* es una expresión especificando la localidad de memoria principal del primer byte a copiar de la fuente.
- *Count* es una expresión indicando el numero de bytes de la fuente que se copiaran en el destino.

Explicación

BYTEMOVE es uno de tres comandos (BYTEMOVE, WORDMOVE, y LONGMOVE) que se usan para copiar bloques de memoria principal de un área a otra. BYTEMOVE copia el numero *Count* de bytes de memoria principal empezando en *SrcAddress* a la memoria principal *DestAddress*.

Usando BYTEMOVE

BYTEMOVE es una excelente forma de copiar largos bloques de memoria de tamaño byte. Por ejemplo:

```
VAR
  byte   Buff1[100]
  byte   Buff2[100]

PUB Main
  bytemove(@Buff2, @Buff1, 100)     'Copia Buff1 a Buff2
```

La primer línea del método `Main` copia los 100-bytes del arreglo `Buff1` al arreglo `Buff2`. BYTEMOVE es mas rápido en esta tarea que un ciclo dedicado REPEAT.

CASE

instrucción: Compara una expresión contra otra expresión y ejecuta el bloque si encuentra que ambas expresiones coinciden.

((PUB PRI))
  CASE *CaseExpression*
    →| *MatchExpression* :
      →| *Statement(s)*
  〈→| *MatchExpression* :
      →| *Statement(s)* 〉
  〈→| OTHER :
      →| *Statement(s)* 〉

- *CaseExpression* es la expresión a comparar.
- *MatchExpression* es un set de valores de coma delimitados y/o un rango de valores para comparar contra *CaseExpression*. Cada *MatchExpression* debe seguirse por dos puntos (:).
- *Statement(s)* es un bloque de una o mas líneas de código para ejecutar cuando *CaseExpression* coincide con su asociado *MatchExpression*. La primera o sola declaración en *Statement(s)* puede aparecer a la derecha de los dos puntos en la línea *MatchExpression* o debajo de esta y ligeramente indentada de *MatchExpression*.

Explicación

CASE es una de las tres instrucciones condicionales (IF, IFNOT, y CASE) que condicionan la ejecución de bloques de código. CASE es la estructura favorita para usar, opuesto a IF..ELSEIF..ELSE, cuando necesitas comparar la igualdad de *CaseExpressio*n a un numero de diferentes valores.

CASE compara *CaseExpression* contra los valores de cada *MatchExpression* en orden y si se encuentra un equivalente se ejecuta el *Statement(s)* asociado. Si no se encuentran igualdades el *Statement(s)* asociado puede ejecutar el código con la instrucción opcional OTHER.

El Indentado es Critico

IMPORTANTE: El indentado es critico. El lenguaje Spin El lenguaje Spin se apoya en la indentacion (de un espacio o mas) de líneas siguiendo comandos opcionales para determinar si pertenecen al comando o no. Para hacer que el la herramienta Propeller indique estos grupos lógicos de bloques de código en pantalla usted puede presionar Ctrl + I para observar

los indicadores. Presionando Ctrl + I nuevamente deshabilitara esta función. Vea la ayuda de la herramienta Propeller para una lista completa de accesos rápidos.

Usando CASE

CASE es practico cuando una de muchas acciones necesitan desarrollarse dependiendo del valor de una expresión. El siguiente ejemplo asume que A, X y Y son variables definidas anteriormente.

```
case X+Y                    'Prueba X+Y
  10, 15: !outa[0]          'X+Y = 10 o 15? activa P0
  A*2   : !outa[1]          'X+Y = A*2? activa P1
  30..40: !outa[2]          'X+Y es 30 a 40? activa P2
X += 5                      'Agrega 5 a X
```

Como las líneas de *MatchExpression* están indentadas de la línea **CASE**, pertenecen a la estructura **CASE** y se ejecutan basadas en la comparación de resultados *CaseExpression*. La siguiente línea `X += 5`, no esta indentada de **CASE**, así que se ejecuta sin importar el resultado de **CASE**.

Este ejemplo compara el valor de `X + Y` contra `10 o 15, A*2` y el rango de `30` a `40`. si `X + Y` es igual a 10 o 15, se activa P0 . Si `X + Y` es igual a `A*2`, se activa P1. Si `X + Y` esta en el rango de `30` a `40`, entonces se activa P2. Sin importar si alguno coincide la línea `X += 5` se ejecuta posteriormente.

Usando OTHER

El componente opcional **OTHER** de **CASE** es similar al componente opcional **ELSE** de una estructura **IF**. Por ejemplo:

```
case X+Y                    'Prueba X+Y
  10, 15: !outa[0]          'X+Y = 10 o 15? activa P0
  25    : !outa[1]          'X+Y = 25? activa P1
  30..40: !outa[2]          'X+Y es 30 a 40? activa P2
  OTHER : !outa[3]          'Si ninguno cumple activa P3
X += 5                      'Agrega 5 a X
```

Este ejemplo es similar al ultimo excepto que posterior al tercer *MatchStatement* que verifica por el rango de 20 a 30 existe un componente **OTHER**. Si `X + Y` no son iguales a 10, 15, 25, o no esta en el rango de 20 a 30, el siguiente bloque *Statement(s)* **OTHER** se ejecuta. Posteriormente se ejecuta la línea `X += 5`.

Existe un importante concepto respecto a este ejemplo. Si X + Y es 10 o 15, se activa P0 o si X + Y es 25, se activa P1, o se X + Y es 20 a 30, se activa P2, etc. Esto es porque se verifica *MatchExpressions* una a la vez en el orden que fueron listadas y solamente se ejecuta el código de la primer expresión que coincide; las demás expresiones no se prueban posteriormente. Esto significa que si teníamos reacomodado las líneas 25 y 20..30 para el rango de 20..30 verifique primero si tenemos un problema en el código. Observe el ejemplo:

```
case X+Y                'Prueba X+Y
  10, 15: !outa[0]      'X+Y = 10 o 15? Activa P0
  20..30: !outa[2]      'X+Y entre 20 a 30? Activa P2
  25    : !outa[1]      'X+Y = 25? Activa P1  <-- ESTO NUNCA CORRE
```

El ejemplo anterior contiene un error porque X + Y podría ser igual a 25 sin embargo esa expresión nunca será probada porque la anterior, 20..30 se probara primero y como es cierta se ejecuta su bloque y ninguna expresión se verifica posteriormente.

Variaciones de las instrucciones

El ejemplo mostrado arriba solo usa una línea por bloque *Statement(s)*, pero cada bloque puede tener varias líneas por supuesto. Además el bloque *Statement(s)* puede aparecer debajo, y ligeramente indentado de la misma *MatchExpression*. Los ejemplos siguientes muestran las variaciones.

```
case A                  'Prueba A
  4     : !outa[0]      'A = 4? Activa P0
  Z+1   : !outa[1]      'A = Z+1? Activa P1
          !outa[2]      'y activa P2
  10..15: !outa[3]      'A entre 10 y 15? Activa P3


case A                  'Prueba A
  4:                    'A = 4?
    !outa[0]            'Activa P0
  Z+1:                  'A = Z+1?
    !outa[1]            'Activa P1
    !outa[2]            'y activa P2
  10..15:               'A entre 10 y 15?
    !outa[3]            'Activa P3
```

CHIPVER

instrucción: Obtener el numero de la Versión del chip Propeller .

((PUB PRI))
  CHIPVER

Regresa: Numero de la versión del chip Propeller.

Explicación

La instrucción CHIPVER lee y regresa el numero de la versión del chip Propeller. Por ejemplo:

```
V := chipver
```

Este ejemplo activa V al numero de la version del chip Propeller, 1 en este caso. Aplicaciones futuras del Propeller pueden usar esto para determinar la versión y tipo de chip Propeller que están corriendo y/o hacer modificaciones a las operaciones según sea necesario.

CLKFREQ

instrucción: Frecuencia de reloj del sistema; frecuencia en la que esta corriendo cada cog.

((PUB PRI))
CLKFREQ

Regresa: La frecuencia actual del reloj de sistema en Hz.

Explicación

El valor de regreso de CLKFREQ es la frecuencia actual del reloj de sistema determinada por el modo de reloj actual (tipo oscilador, ganancia y parámetros PLL) y el pin de frecuencia externa X1 en su caso. Los objetos usan CLKFREQ para determinar el tiempo adecuado de retraso para operaciones sensitivas al tiempo. Por ejemplo:

```
waitcnt(clkfreq / 10 + cnt)  'espera .1 segundos (100 ms)
```

Esta sentencia divide CLKFREQ por 10 y suma el resultado a CNT (el valor actual del contador del sistema) luego espera (WAITCNT) hasta que el contador del sistema alcanza el valor del resultado. Como CLKFREQ es el numero de ciclos por segundo, una división por 10 resulta en el numero de ciclos de reloj cada 0.1 segundos o 100 ms. De esta forma independientemente del tiempo que toma procesar la expresión, esta sentencia detiene la ejecución del programa por 100ms. La tabla de abajo muestra mas ejemplos del reloj del sistema contra cálculos de tiempo.

Tabla 2-2: Ciclos de reloj de sistema vs. Calculo de tiempo

expresión	Resultado
`clkfreq / 10`	Ciclo de reloj por 0.1 segundos (100 ms)
`clkfreq / 100`	Ciclo de reloj por 0.01 segundos (10 ms)
`clkfreq / 1_000`	Ciclo de reloj por 0.001 segundos (1 ms)
`clkfreq / 10_000`	Ciclo de reloj por 0.0001 segundos (100 µs)
`clkfreq / 100_000`	Ciclo de reloj por 0.00001 segundos (10 µs)
`clkfreq / 9600`	Ciclo de reloj por periodo de bit a 9,600 baud (~ 104 µs)
`clkfreq / 19200`	Ciclo de reloj por periodo de bit a 19,200 baud (~ 52 µs)

El valor que regresa CLKFREQ se lee del long 0 (la primera localidad en RAM) y ese valor puede cambiar cada que la aplicación cambia el modo de reloj, ya sea manual o a través de la

instrucción CLKSET. Los objetos que son sensitivos al tiempo deberán verificar CLKFREQ en puntos estratégicos para ajustar a los nuevos parámetros automáticamente.

CLKFREQ vs. _CLKFREQ

CLKFREQ esta relacionado a, pero no es el mismo que _CLKFREQ. CLKFREQ es una instrucción que regresa la frecuencia actual del reloj del sistema mientras _CLKFREQ es una constante de aplicación definida que contiene la aplicación de la frecuencia del reloj del sistema al inicio. En otras palabras, CLKFREQ es la frecuencia actual del reloj y _CLKFREQ es la frecuencia del reloj original; pueden llegar a tener el mismo valor pero también pueden ser diferentes.

_CLKFREQ

Constante: Pre-definido, constante activada una vez para especificar la frecuencia del reloj de sistema.

```
CON
  _CLKFREQ = expresión
```

- *expresión* es una expresión entera que indica la frecuencia del reloj del sistema al inicio de la aplicación.

Explicación

_CLKFREQ especifica la frecuencia del reloj del sistema al inicio. Es un símbolo constante pre definido que su valor esta determinado por el objeto superior de una aplicación. _CLKFREQ se active ya sea directamente por la aplicación misma o indirectamente como resultado de los parámetros _CLKMODE y _XINFREQ.

El objeto superior en una aplicación (en el que comienza la compilación) puede especificar un parámetro para _CLKFREQ en su bloque CON. Esto define la frecuencia inicial del reloj del sistema para la aplicación y es la frecuencia que el reloj del sistema usara tan pronto como la aplicación inicie y la ejecución comience.

La aplicación puede especificar _CLKFREQ o _XINFREQ en el bloque CON; son mutuamente exclusivos y el no especificado es automáticamente calculado y activado como resultado de la especificación del otro..

Los siguientes ejemplos asumen que están contenidos en el archivo superior. Cualquier parámetro de _CLKFREQ en objetos hijos simplemente es ignorado por el compilador.

Por ejemplo:

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _CLKFREQ = 32_000_000
```

La primera declaración en el bloque CON active el modo de clock para un cristal de baja velocidad externo y un Clock PLL multiplicador de 8. La segunda declaración activa la frecuencia del reloj del sistema a 32 MHz, lo cual significa que la frecuencia del cristal externo debe ser 4 MHz porque 4 MHz * 8 = 32 MHz. El valor _XINFREQ se activa automáticamente a 4 MHz por estas declaraciones.

```
CON
  _CLKMODE = XTAL2
  _CLKFREQ = 10_000_000
```

Estas dos declaraciones activan el modo clock para un cristal externo de media velocidad, no se activa multiplicador Clock PLL, y la frecuencia del reloj del sistema a 10 MHz. El valor de **_XINFREQ** se activa automáticamente a 10 MHz, también por estas declaraciones.

CLKFREQ vs. _CLKFREQ

CLKFREQ esta relacionado a, pero no es el mismo que **_CLKFREQ**. **CLKFREQ** es una instrucción que regresa la frecuencia actual del reloj del sistema mientras **_CLKFREQ** es una constante de aplicación definida que contiene la aplicación de la frecuencia del reloj del sistema al inicio. En otras palabras, **CLKFREQ** es la frecuencia actual del reloj y **_CLKFREQ** es la frecuencia del reloj original; pueden llegar a tener el mismo valor pero también pueden ser diferentes.

CLKMODE

instrucción: Activación del modo clock actual.

((PUB PRI))
  CLKMODE

Regresa: Modo clock actual.

Explicación

La activación del modo clock es un valor de tamaño byte, determinado por la aplicación al tiempo de compilación desde el registro CLK. Ver Registro CLK, Pág. 28, para una explicación de las activaciones posibles. Por ejemplo:

```
Mode := clkmode
```

Esta sentencia puede usarse para activar una variable Mode, al parámetro actual del modo clock. Muchas aplicaciones mantienen un modo de activación estático de clock; sin embargo algunas aplicaciones cambiaran el modo clock durante tiempo real para ajustar velocidades del reloj, modos de bajo consumo, etc. Puede ser necesario para algunos objetos poner atención a la dinámica potencial de los modos clock para mantener tiempos apropiados y funcionalidad.

CLKMODE vs._CLKMODE

CLKMODE esta relacionado, pero no es lo mismo que, _CLKMODE. CLKMODE es una instrucción que regresa el valor actual del modo clock (en la forma de un patrón de bit de CLK) mientras _CLKMODE es una constante de aplicación definida que contiene el modo requerido clock al inicio (en la forma de activación de la constante clock que son OR). Ambas pueden describir el mismo modo lógico de clock pero sus valores no son iguales.

_CLKMODE

Constante: Una constante de tiempo pre definida ajustable para especificar el nivel de aplicación en los parámetros del modo clock .

```
CON
  _CLKMODE = expresión
```

- *expresión* es un expresión entero hecha para uno o dos parámetros de modo de clock constantes mostrados en la Tabla 2-3. Esto definirá el modo clock cuando inicie la aplicación.

Explicación

_CLKMODE se usa para especificar la naturaleza del reloj del sistema. Es un símbolo constante pre definido cuyo valor esta determinado por el objeto superior de una aplicación. La activación del modo clock es un byte y su valor esta descrito por una combinación de constantes RCxxxx, XINPUT, XTALx y PLLxx al tiempo de compilación. La Tabla 2-3 ilustra el modo de la programación del modo de reloj. Note que no todas las combinaciones son validas; La Tabla 2-4 muestra todas las combinaciones validas.

Tabla 2-3: Constantes de Activación del Modo Clock

Constante[1] Activación Modo Clock	Resistencia[2] XO	Capacitancia[2] XI/XO	Descripción
RCFAST	Infinita	n/a	Oscilador interno rápido (~12 MHz). Puede ser 8 MHz a 20 MHz. (Default)
RCSLOW	Infinita	n/a	Oscilador interno lento (~20 KHz). Puede ser 13 KHz a 33 KHz.
XINPUT	Infinita	6 pF (solo pad)	Oscilador-Reloj externo (DC a 80 MHz); XI pin solo, XO desconectado
XTAL1	2 kΩ	36 pF	Cristal externo baja velocidad (4 MHz a 16 MHz)
XTAL2	1 kΩ	26 pF	Cristal externo velocidad media (8 MHz a 32 MHz)
XTAL3	500 Ω	16 pF	Cristal externo alta velocidad (20 MHz a 80 MHz)
PLL1X	n/a	n/a	Multiplica la frecuencia externa 1 vez
PLL2X	n/a	n/a	Multiplica la frecuencia externa 2 veces
PLL4X	n/a	n/a	Multiplica la frecuencia externa 4 veces
PLL8X	n/a	n/a	Multiplica la frecuencia externa 8 veces
PLL16X	n/a	n/a	Multiplica la frecuencia externa 16 veces

1. Todas las constantes están también disponibles en Ensamblador Propeller.
2. Todas las resistencias / capacitores necesarios están incluidos en el chip Propeller.

Tabla 2-4: Expresiones Validas de Modo Clock y Valores de Registro CLK			
expresión Valida	Valor de Registro CLK	expresión Valida	CLK Register Value
RCFAST	0_0_0_00_000	XTAL1 + PLL1X	0_1_1_01_011
RCSLOW	0_0_0_00_001	XTAL1 + PLL2X	0_1_1_01_100
XINPUT	0_0_1_00_010	XTAL1 + PLL4X	0_1_1_01_101
		XTAL1 + PLL8X	0_1_1_01_110
		XTAL1 + PLL16X	0_1_1_01_111
XTAL1	0_0_1_01_010	XTAL2 + PLL1X	0_1_1_10_011
XTAL2	0_0_1_10_010	XTAL2 + PLL2X	0_1_1_10_100
XTAL3	0_0_1_11_010	XTAL2 + PLL4X	0_1_1_10_101
		XTAL2 + PLL8X	0_1_1_10_110
		XTAL2 + PLL16X	0_1_1_10_111
XINPUT + PLL1X	0_1_1_00_011	XTAL3 + PLL1X	0_1_1_11_011
XINPUT + PLL2X	0_1_1_00_100	XTAL3 + PLL2X	0_1_1_11_100
XINPUT + PLL4X	0_1_1_00_101	XTAL3 + PLL4X	0_1_1_11_101
XINPUT + PLL8X	0_1_1_00_110	XTAL3 + PLL8X	0_1_1_11_110
XINPUT + PLL16X	0_1_1_00_111	XTAL3 + PLL16X	0_1_1_11_111

El objeto superior en una aplicación (donde comienza la compilación) puede especificar un parámetro para **_CLKMODE** en su bloque **CON**. Esto define el modo inicial de clock para la aplicación y es el modo que el Reloj de Sistema usara tan pronto como se inicie la aplicación y comience la ejecución. Los siguientes ejemplos asumen que están contenidos dentro del objeto superior. Cualquier parámetro **_CLKMODE** en objetos hijos es ignorado por el compilador. Por ejemplo:

```
CON
  _CLKMODE = RCFAST
```

Esta instrucción activa el modo clock para el circuito Reloj/Oscilador RC a modo rápido. El reloj del sistema correrá aproximadamente a 12 MHz con este parámetro. El parámetro **RCFAST** es el parámetro por defecto, por lo tanto si no se define **_CLKMODE** este es el parámetro que se utilizara. Note que el Clock PLL no puede usarse con el Reloj Oscilador RC interno. Este es un ejemplo con el reloj externo:

```
CON
  _CLKMODE = XTAL1 + PLL8X
```

Esta instrucción active el modo clock para un cristal externo de baja velocidad (**XTAL1**), habilita el circuito PLL y active el reloj del sistema para usar el 8x multiplicador del Clock PLL (**PLL8X**). Si se conecta un cristal externo de 4 MHz a XI y XO, por ejemplo, su señal se

multiplicara por 16 (el clock PLL siempre multiplica por 16) pero se usara el bit resultado de 8x; el reloj de sistema deberá ser 4 MHz * 8 = 32 MHz.

```
CON
  _CLKMODE = XINPUT + PLL2X
```

Esta instrucción active el modo clock para un oscilador externo conectado a X1 solamente y habilita el circuito clock PLL para activar el reloj del sistema usando el resultado 2x. Si se anexo un reloj-oscilador externo de 8 MHz en XI, el reloj del sistema correrá a 16 MHz; lo cual es 8 MHz * 2.

Note que el clock PLL no es requerido y puede deshabilitarse simplemente sin especificar el multiplicador, por ejemplo:

```
CON
  _CLKMODE = XTAL1
```

Esto active el modo clock para un cristal externo de baja velocidad pero deja el clock PLL deshabilitado; el reloj del sistema será igual a la frecuencia del cristal externo.

Los Parámetros _CLKFREQ y _XINFREQ

Por simplicidad los ejemplos solo muestran parámetros **_CLKMODE**, pero cualquier parámetro **_CLKFREQ** o **_XINFREQ** se requiere seguirlo para que el objeto pueda determinar la frecuencia actual del reloj. La siguiente es la versión completa del segundo ejemplo con un cristal externo de 4MHZ (**_XINFREQ**).

```
CON
  _CLKMODE = XTAL1 + PLL8X     'cristal baja velocidad x 8
  _XINFREQ = 4_000_000         'cristal externo de 4 MHz
```

El ejemplo es igual al segundo ejemplo de arriba pero **_XINFREQ** indica la frecuencia del cristal externo de 4 MHz. El Propeller usa este valor con el parámetro de **_CLKMODE** para determinar la frecuencia del reloj del sistema (como se reporta por la instrucción **CLKFREQ**) para que los objetos puedan ajustar apropiadamente su tiempo. Ver **_XINFREQ**, Pág. 241.

CLKMODE vs._CLKMODE

CLKMODE esta relacionado, pero no es lo mismo que, **_CLKMODE**. **CLKMODE** es una instrucción que regresa el valor actual del modo clock (en la forma de un patrón de bit de CLK) mientras **_CLKMODE** es una constante de aplicación definida que contiene el modo requerido clock al inicio (en la forma de activación de la constante clock que son OR). Ambas pueden describir el mismo modo lógico de clock pero sus valores no son iguales.

CLKSET

instrucción: Activa ambos modos clock y frecuencia del reloj del sistema en tiempo real.

((PUB PRI))
CLKSET (*Mode*, *Frequency*)

- *Mode* es una expresión entero que se escribirá al registro CLK para cambiar el modo clock.
- *Frequency* es una expresión entero que indica el resultado de la frecuencia del reloj de sistema.

Explicación

Una de las características mas potentes del chip Propeller es la habilidad de cambiar el comportamiento del reloj en tiempo real. Una aplicación puede escoger uno u otro modo entre baja velocidad (para bajo consumo) y una alta velocidad (para operaciones de banda ancha), por ejemplo. CLKSET se usa para cambiar el modo clock y frecuencia mientras corre el programa. Esto es equivalente a definir el equivalente de _CLKMODE y la constante _CLKFREQ definida por la aplicación al momento de compilación. Por ejemplo:

```
clkset(%01101100, 4_000_000)            'Activa a XTAL1 + PLL2x
```

Esto active el modo clock a un cristal de baja velocidad externo y un multiplicador clock PLL de 2, indica el resultado de la frecuencia del reloj del sistema (CLKFREQ) que es 4 MHz. Después de ejecutar este comando los comandos CLKMODE y CLKFREQ reportaran los parámetros actualizados para que los objetos los utilicen..

En general es seguro cambiar entre modos clock usando un comando CLKSET, sin embargo si el circuito del cristal oscilador esta habilitado (bit OSCENA del registro CLK) es importante desarrollar el cambio de modo clock con un proceso de tres partes:

1) Primero activar los bits de PLLENA, OSCENA, OSCM1 y OSCM0 del registro CLK como sea necesario. Ver Registro CLK en Pág. 28 para mas información.
2) Esperar 10 ms para darle tiempo de estabilización al cristal externo.
3) Activar los bits CLKSELx del registro CLK como sea necesario para cambiar el reloj del sistema a la nueva fuente.

El proceso mencionado solo es necesario cuando se esta cambiando el circuito del cristal oscilador encendido. Ningún otro cambio de clock requiere este proceso si el circuito del

cristal oscilador se deja en el estado actual, sin importar si esta encendido o apagado. Ver el objeto Clock en la librería Propeller para métodos y tiempos de modificación de clock.

NOTA: Toma aproximadamente 75 µs al Propeller desarrollar la acción de cambio entre fuentes de reloj.

CNT

Registro: Registro Contador del Sistema.

((PUB PRI))
  CNT

Regresa: El valor actual del contador del sistema 32-bit.

Explicación

El registro CNT contiene el valor actual en el contador del sistema global de 32 bits. El contador del sistema sirve como referencia central de tiempo para todos los cogs; incrementa su valor de 32 bits cada ciclo del reloj del sistema.

Comenzando con un inicio o reinicio el contador del sistema comienza con un valor arbitrario y de ahí cuenta incrementando con cada ciclo del reloj del sistema. Como el contador del sistema es un recurso de solo lectura cada cog puede leerlo simultáneamente y puede usar el valor de regreso para sincronizar eventos, contar ciclos y medir tiempo.

Usando CNT

Leer CNT para obtener el valor actual del contador del sistema. El valor actual en si no significa nada para ningún propósito, pero la diferencia en lecturas sucesivas es muy importante. Con mayor frecuencia el registro CNT se usa para retrasar la ejecución por un periodo especifico o para sincronizar un evento al iniciar una ventana de tiempo. El siguiente ejemplo usa la instrucción WAITCNT para lograr esto.

En código Spin, cuando se usa CNT dentro de un comando WAITCNT como se muestra, asegúrese de escribir la expresión en la forma "offset + cnt" no "cnt + offset" y asegúrese que offset es al menos 381 para contar por el interprete Spin y evitar retrasos de tiempo inesperados. Ver el comando Vea la sección de Pausas Fijas del comando WAITCNT en Pág. 223 para mayor información.

```
waitcnt(3_000_000 + cnt)    'Espera 3 millones de ciclos de reloj
```

El código de arriba es un ejemplo de un retraso fijo. Retrasa la ejecución del cog por 3 millones de ciclos de reloj (cerca de ¼ de segundo cuando se esta usando el oscilador interno en modo rápido).

El siguiente es un ejemplo de un retraso sincronizado. Nota el contador actual en un lugar y desarrolla una acción (activa un pin) cada milisegundo subsecuentemente con una precisión tan buena como la del oscilador que este manejando el chip Propeller.

```
PUB Toggle | TimeBase, OneMS
  dira[0]~~                       'Acyiva P0 a salida
  OneMS := clkfreq / 1000         'Calcula ciclos por 1 milisegundo
  TimeBase := cnt                 'Obtiene el contador actual
  repeat                          'Ciclo sin fin
    waitcnt(TimeBase += OneMS)    'Espera para iniciar el siguiente
milisegundo
    !outa[0]                      'Activa P0
```

Aquí el pin 0 de E/S se active como salida. Entonces la variable local `OneMS` obtiene el valor actual de la frecuencia del reloj dividida por 1000; Ej. El numero de ciclos de reloj del sistema por 1 milisegundo de tiempo. Después la variable local `TimeBase` obtiene el valor actual del contador del sistema. Finalmente las dos ultimas líneas del código repiten sin fin cada vez esperando hasta que inicie el siguiente milisegundo y luego activan el estado de P0.

Para mayor información vea la sección de Pausas Fijas de **`WAITCNT`** en la Pág. 223 y Pausas Sincronizadas en Pág. 224.

El registro CNT es de solo lectura así que no deberá asignarse en Spin un valor (Ej. No debe tener a la izquierda un `:=` o cualquier operador de asignación) y cuando se use en Ensamblador Propeller deberá accesarse como valor fuente (campo-s) (Ej. **`mov`** *dest*, *source*).

COGID

instrucción: Numero de Cog Actual (0-7).

((PUB PRI))
COGID

Regresa: El numero actual del cog (0-7).

Explicación

El valor de regreso de COGID es la identificación del cog que ejecuto el comando. Normalmente el cog actual en el cual esta corriendo el código no importa, sin embargo para algunos objetos puede ser importante mantener el rastreo. Por ejemplo:

```
PUB StopMyself  'Detiene el cog donde esta corriendo el código
  cogstop(cogid)
```

Este método ejemplo, StopMyself, tiene una línea de código que simplemente llama a COGSTOP con COGID como parámetro. Como COGID regresa el numero del cog que esta corriendo el código esta rutina ocasiona que el cog se termine así mismo.

COGINIT

Instrucción: Inicia o reinicia un Cog por numero para correr Spin o Ensamblador Propeller.

((PUB PRI))
COGINIT (*CogID*, *SpinMethod* 〈(*ParameterList*)〉, *StackPointer*)

((PUB PRI))
COGINIT (*CogID*, *AsmAddress*, *Parameter*)

- *CogID* es el número de cog a iniciar (0 – 7) o reiniciar. Un numero mayor a 7 dará como resultado que el siguiente cog disponible inicie (si es posible).
- *SpinMethod* es el método Spin PUB o PRI que el cog afectado deberá correr. Opcionalmente puede definirse una lista de parámetros entre paréntesis.
- *ParameterList* es una lista opcional de coma delimitada de uno o mas parámetros para *SpinMethod*. Se debe incluir solo si *SpinMethod* requiere parámetros.
- *StackPointer* es un apuntador de memoria, reservado para espacio de pila para el cog afectado. El cog afectado usa este espacio para almacenar datos temporalmente durante llamadas complementarias y evaluación de expresiones. Si no se proporciona suficiente espacio la aplicación fallara o correrá dando resultados extraños.
- *AsmAddress* es la dirección de una rutina de Ensamblador Propeller del bloque DAT.
- *Parameter* se usa opcionalmente para pasar un valor al nuevo cog. Este valor termina en el registro del parámetro de inicio (PAR) de solo lectura del cog afectado. *Parameter* puede pasar ya sea un valor sencillo de 14-bit o la dirección de un bloque de memoria y usarlo en una rutina Ensamblador. *Parameter* se requiere en COGINIT, si no se necesita para la rutina simplemente se puede colocar un valor como cero (0).

Explicación

COGINIT trabaja exactamente igual a COGNEW (Pág. 81) con dos excepciones: 1) inicia el código en un cog especifico donde el numero es *CogID*, y 2) no regresa un valor. Como COGINIT opera en un especifico cog como lo indica el parámetro *CogID* puede usarse para detener y reiniciar un cog activo en un solo paso. Esto incluye el cog actual; Ej. Un cog puede usar COGINIT para detener y reiniciarse a si mismo para correr quizá un código completamente diferente.

Note que cada cog en el cual se correo código spin debe tener su propio espacio de pila. Los cogs no pueden compartir el mismo espacio de pila.

Además debe cuidar cuando reinicia un cog con código spin y especificando el espacio de pila que se esta usando en otro cog. El Propeller siempre construye un marco de pila inicial

del cog en e; espacio especificado de pila antes de iniciar el cog. Reiniciar un cog con una instrucción COGINIT especificando el mismo espacio de pila que esta usando producirá una nueva imagen del marco que golpeara antes de que el cog reinicie. Para evitar que esto suceda primero indique un COGSTOP en ese cog seguido del deseado COGINIT.

código Spin (Sintaxis 1)

Para correr un método spin en un cog especifico la instrucción COGINIT necesita el numero de cog, el nombre del método, sus parámetros y un apuntador a un espacio de pila. Por ejemplo:

```
coginit(1, Square(@X), @SqStack)  'Inicia Square en Cog 1
```

Este ejemplo inicia el método Square en el cog 1, pasando la dirección de X a Square y la dirección de SqStack como apuntador de pila de COGINIT. Ver COGNEW, Pág. 81, para mayor información.

código Ensamblador Propeller (Sintaxis 2)

Para correr código Ensamblador Propeller en un cog especifico el comando COGINIT necesita el numero de cog, la dirección de la rutina ensamblador, y un valor que puede usarse opcionalmente para la rutina ensamblador. Por ejemplo:

```
coginit(2, @Toggle, 0)
```

Este ejemplo inicia la rutina Ensamblador Propeller Toggle, en el Cog 2 con un parámetro PAR de 0. Vea el ejemplo en la descripción de la sintaxis de COGNEW, Pág. 81, para un ejemplo mas detallado teniendo en cuenta que la instrucción COGINIT de arriba puede usarse en vez de COGNEW en ese ejemplo.

El Campo *Parameter*

Es importante notar que el campo *Parameter* pretende pasar una dirección long, así solo 14 bits (bits 2 al 15) se pasan en el registro PAR del cog; los bits mas bajos se ponen a ceros siempre para asegurar que es una dirección con numero long alineado. Un valor diferente a dirección long se puede pasar a través del campo *Parameter* pero tiene que limitarse a 14-bits, debe recorrerse a la izquierda dos bits (por instrucción COGNEW/COGINIT), y tendrá que moverse a la derecha dos bits por el programa ensamblador que lo recibe.

COGNEW

instrucción: Inicia el siguiente cog disponible para correr Spin o Ensamblador Propeller.

((PUB PRI))
COGNEW (*SpinMethod* 〈(*ParameterList*)〉, *StackPointer*)

((PUB PRI))
COGNEW (*AsmAddress*, *Parameter*)

Regresa: El numero del cog iniciado con éxito (0-7), o -1 si no se inicio.

- *SpinMethod* es el método spin PUB o PRI que el Nuevo cog debe correr. Opcionalmente puede seguirlo una lista de parámetros entre paréntesis.
- *ParameterList* es una lista opcional de coma delimitada de uno o mas parámetros para *SpinMethod*. Debe incluirse solo si *SpinMethod* requiere parámetros.
- *StackPointer* es un apuntador de memoria reservado para espacio de pila para el Nuevo cog. El nuevo cog usa este espacio para almacenar datos temporales durante llamadas y evaluaciones de expresiones. Si no hay suficiente espacio la aplicación fallara o correrá con resultados extraños.
- *AsmAddress* es la dirección de una rutina de Ensamblador Propeller de un bloque DAT.
- *Parameter* se usa opcionalmente para pasar un valor al nuevo cog. Este valor termina en el registro del parámetro de inicio (PAR) de solo lectura del cog afectado. *Parameter* puede pasar ya sea un valor sencillo de 14-bit o la dirección de un bloque de memoria y usarlo en una rutina Ensamblador. *Parameter* se requiere en COGINIT, si no se necesita para la rutina simplemente se puede colocar un valor como cero (0).

Explicación

COGNEW inicia un nuevo cog y corre en Spin o Ensamblador Propeller. Si tiene éxito, COGNEW regresa el numero del Nuevo cog iniciado. Si no hay mas cogs disponibles COGNEW regresa -1. COGNEW trabaja exactamente igual que COGINIT (Pág. 79) con dos excepciones: 1) inicia código en el siguiente cog disponible y 2) regresa el numero de cog en el que se inicio.

código Spin (Sintaxis 1)

Para correr un método spin en otro cog la instrucción **COGNEW** necesita el nombre del método, sus parámetros y un apuntador a un espacio de pila. Por ejemplo:

```
VAR
  long SqStack[6]               'Espacio de pila para Square cog


PUB Main | X
  X := 2                        'Inicia X
  cognew(Square(@X), @SqStack)  'Inicia square cog
  <check X here>                'Ciclo aqui y checa X


PUB Square(XAddr)               'Cuadrado del valor en XAddr
  repeat                        'Repite el siguiente infinito
    long[XAddr] *= long[XAddr]  'Valor cuadrado almacenado
    waitcnt(2_000_000 + cnt)    'Espera 2 millones de ciclos
```

Este ejemplo muestra dos métodos, `Main` y `Square`. `Main` inicia un cog que corre `Square` infinitamente, entonces `Main` monitorea el resultado en `X`. `Square`, que lo corre otro cog toma el valor de `XAddr`, lo eleva al cuadrado y almacena el resultado de regreso en `XAddr`, espera 2 millones de ciclos antes de hacerlo nuevamente. Hay mas explicaciones pero el resultado es que `X` inicia como 2, y el segundo cog que corre `Square`, continuamente activa `X` a 4, 16, 256, 65536 y finalmente a 0 (sobre flujo de 32 bits), todo esto independientemente de que el primer cog puede estar verificando el valor de `X` o desarrollando otras tareas.

El método `Main` declara una variable local, `X`, que se active en 2 en su primer línea. `Main` inicia un nuevo cog usando **COGNEW**, para correr el método `Square` en un cog separado. El primer parámetro de **COGNEW**, `Square(@X)`, es el método Spin a correr y se requiere el parámetro; en este caso pasamos la dirección de la variable `X`. El segundo parámetro de **COGNEW**, `@SqStack`, es la dirección del espacio de pila reservado para el nuevo cog. Cuando un cog inicia para correr código spin necesita espacio de pila donde almacenar datos. Este ejemplo solo requiere 6 longs de pila para la operación (ver La Necesidad de Espacio de Pila, para mayor información).

Después de ejecutar **COGNEW**, hay dos cogs corriendo, el primero aun corriendo en el método `Main` y el segundo se inicia en el método `Square`. Aun con el hecho de que están usando código del mismo objeto spin están corriendo independientemente. La línea "`<check X here>`" puede reemplazarse con código que usa el valor de `X` de alguna forma.

La Necesidad de Espacio de Pila

Un cog ejecutando código spin a diferencia de uno ejecutando ensamblador Propeller necesita un espacio temporal llamado espacio de pila para retener datos tales como llamadas de pila, parámetros y resultados intermedios. Sin esto características sofisticadas como llamadas a métodos, valores y expresiones complejas no serian posibles sin tener limitaciones.

El compilador Spin automáticamente proporciona espacio de pila para la aplicación del código inicial, el código superior de la aplicación spin. El "espacio libre" siguiendo la memoria de la aplicación se usa para este propósito. Sin embargo el compilador no puede distinguir entre bloques de pila para el código spin que la aplicación esta corriendo por si misma, por lo tanto la aplicación debe proveer el espacio de pila a si mismo.

Típicamente este espacio de pila se proporciona en la forma de variables globales declaradas solo para ese uso, tales como la variable `SqStack` en el ejemplo. Desafortunadamente es difícil determinar cuanto espacio debe proporcionarse, así que cuando desarrolle un objeto se sugiere que inicialmente proporcione un monto largo de memoria long (120 longs o mas) y una vez que el objeto se completo utilice un objeto como el objeto Stack Lenght de la librería Propeller para determinar el tamaño optimo. Ver el objeto Stack Lenght para mayor explicación

código Spin solo puede ser iniciado por su propio Objeto

En lenguaje spin, por diseño, los objetos deben inteligentemente controlar su propia información, los métodos que operan, los cogs que ejecutan esos métodos y la interfase que otros objetos usan para afectarlo. Estos son todos los aspectos que sirven para mantener la integridad del objeto e incrementar su uso y consistencia.

Por estas razones el objeto y su diseñador son notablemente el mejor equipo para proveer el apropiado espacio de pila que se requiere para que el código spin pueda iniciarse en otro cog.

Para reforzar este principio la instrucción **`COGNEW`** y **`COGINIT`** no pueden iniciar código spin fuera de su objeto contenido. Esto significa que la sentencia como la que sigue no trabajara como se espera.

```
cognew(SomeObject.Method, @StackSpace)
```

En vez de iniciar `SomeObject.Method` en otro cog, el Propeller ejecutara `SomeObject.Method` dentro del cog actual y si ese método regresa un valor el Propeller tomara el valor y lo usara como la dirección de código a iniciar con la instrucción cognew. Esto no representa la intención de lo que el escritor del código quería realizar.

Si Method se determina para ser código que es verdaderamente importante para correr en otro cog en vez de escribir el código como el ejemplo de arriba, SomeObject deberá reescribirse de la siguiente forma.

```
VAR
  long StackSpace[8]                    'Espacio pila para nuevo cog
  byte CogID                            'Almacena Numero del Nuevo
cog

PUB Start
  Stop                                  'Previene multiples inicios
  CogID := cognew(Method, @StackSpace)  'Inicia método en otro cog

PUB Stop
  if CogID > -1
    cogstop(CogID)                      'Detiene el cog previamente
iniciado

PRI Method
  <Mas código aqui>
```

El ejemplo incluye dos métodos de interfase publicas, Start y Stop, los cuales un objeto puede usar para iniciar apropiadamente el código de objeto en otro cog. El principio importante es que el objeto mismo esta proporcionando esta capacidad, y haciendo eso, esta controlándola memoria de pila requerida para la operación apropiada. Note también que Method se cambio a método privado (PRI) para descartar llamadas directas desde afuera.

Código Ensamblador Propeller (Sintaxis 2)

Ensamblador Propeller en otro cog, la instrucción COGNEW necesita direccionar la rutina ensamblador y un valor que puede opcionalmente ser usado por la rutina ensamblador, por ejemplo:

```
PUB Main
  cognew(@Toggle, 0)                    'Inicia código Toggle

DAT
                org 0                   'Reinicia el apuntador
ensamblador
```

```
Toggle              rdlong   Delay, #0          'Obtiene frecuencia de
reloj
                    shr      Delay, #2          'Divide en 4
                    mov      Time,  cnt         'Obtiene el tiempo
actual
                    add      Time,  Delay       'Ajusta por 1/4 de
segundo
                    mov      dira,  #1          'Activa pin 0 a salida
Loop                waitcnt  Time,  Delay       'Espera 1/4 de segundo
                    xor      outa,  #1          'activa pin
                    jmp      #Loop              'ciclo

Delay     res       1
Time      res       1
```

La instrucción **COGNEW** en el método `Main` le dice al Propeller que inicie el código ensamblador `Toggle` en un cog nuevo. El Propeller encuentra un cog disponible y copia 496 longs del contenido del bloque **DAT**, empezando en `Toggle`, en la RAM del cog. Luego el registro del cog **PAR** se inicia, los demás registros de propósito especial se limpian, y el cog comienza a ejecutar el código ensamblador empezando en la localidad 0 de la RAM.

El campo *Parameter*

Es importante notar que el campo *Parameter* pretende pasar una dirección long, así solo 14 bits (bits 2 al 15) se pasan en el registro **PAR** del cog; los bits mas bajos se ponen a ceros siempre para asegurar que es una dirección con numero long alineado. Un valor diferente a dirección long se puede pasar a través del campo *Parameter* pero tiene que limitarse a 14-bits, debe recorrerse a la izquierda dos bits (por instrucción **COGNEW**/**COGINIT**), y tendrá que moverse a la derecha dos bits por el programa ensamblador que lo recibe.

COGSTOP

instrucción: Detiene un cog por su numero.

((PUB PRI))
COGSTOP (*CogID*)

- *CogID* es el numero (0 – 7) del cog a detener.

Explicación

COGSTOP detiene un cog de numero *CogID* y coloca al cog en un estado durmiente. En este estado el cog deja de recibir pulsos del reloj del sistema de tal forma que el consumo de potencia se reduce de manera significante.

Para detener un cog coloque la instrucción COGSTOP y el numero de cog. Por ejemplo:

```
VAR
  byte Cog    'Usa para almacenar numero del cog iniciado


PUB Start(Pos) : Pass
  'Inicia un Nuevo cog para correr actualizado con Pos,
  'regresa TRUE si tiene éxito
  Pass := (Cog := cognew(@Update, Pos) + 1) > 0


PUB Stop
  'Detiene el cog iniciado anteriormente, si existe.
  if Cog
    cogstop(Cog~ - 1)
```

Este ejemplo usa **COGSTOP en el método publico** Stop para detener el cog que inicio previamente en el método Start. Ver **COGNEW**, Pág. 81, para mas información de este ejemplo.

CON

Denominador: Declara un Bloque Constante .

```
CON
  Symbol = Expression ⟨(( , ↳)) Symbol = Expression⟩...
```

```
CON
  #Expression (( , ↳)) Symbol ⟨[Offset]⟩ ⟨(( , ↳)) Symbol ⟨[Offset]⟩ ⟩...
```

```
CON
  Symbol ⟨[Offset]⟩ ⟨(( , ↳)) Symbol ⟨[Offset]⟩ ⟩...
```

- *Symbol* es el nombre deseado por la constante.
- *expresión* es cualquier entero valido, o de punto flotante, o expresión constante algebraica. La expresión puede incluir otros símbolos constantes como long según hayan sido definidos previamente.
- *Offset* es una expresión opcional por la cual se ajusta el valor detallado para *Symbol* siguiendo a este. Si *Offset* no se especifica, se aplica por defecto 1. Use *Offset* para influenciar el siguiente valor detallado de *Symbol* para algo mas que el valor de *Symbol* mas 1.

Explicación

El Bloque constante es una sección de código fuente que declara símbolos constantes globales y parámetros de configuración globales del Propeller. Este es uno de seis denominadores especiales (`CON`, `VAR`, `OBJ`, `PUB`, `PRI`, y `DAT`) que proporcionan una estructura que no se puede separar del lenguaje Spin.

Las constantes son valores numéricos que no pueden cambiar en tiempo real. Pueden definirse en términos de valores simples (1, $F, 65000, %1010, %%2310, “A”, etc.) o como expresiones, llamadas expresiones constantes (25 + 16 / 2, 1000 * 5, etc.) que siempre resulta en un numero especifico.

El bloque constante es un área de código usado específicamente para asignar símbolos (nombres útiles) a constantes de forma que esos símbolos pueden usarse en cualquier lugar del código donde se requiere un valor constante. Esto hace el código mas legible y fácil de mantener porque se puede cambiar posteriormente el valor de una constante que aparece en varios lugares. Estas constantes son globales al objeto así que cualquier método dentro del mismo podrá usarlas. Hay muchas formas de definir constantes, se describen abajo.

Declaraciones Constantes Comunes (Sintaxis 1)

La forma mas común de declaraciones constantes comienza con una línea **CON** sola seguida por una o mas declaraciones. **CON** debe empezar en la columna 1 (la columna de la extrema izquierda) de la línea donde se encuentra y se recomienda que la siguientes líneas se indenten por al menos un espacio. Las expresiones pueden ser combinaciones de números, operadores, paréntesis y caracteres entre comillas. Ver Operadores , Pág. 147, para ejemplo de expresiones.

Ejemplo:

```
CON
  Delay = 500
  Baud = 9600
  AChar = ''A''
```

—o—

```
CON
  Delay = 500, Baud = 9600, AChar = ''A''
```

Ambos ejemplos creados con el símbolo `Delay` el cual es igual a 500, un símbolo `Baud` igual a 9600, y un símbolo `AChar` igual al caracter “A”. Para la declaración `Delay`, por ejemplo podríamos usar una expresión algebraica tal como:

```
  Delay = 250 * 2
```

La sentencia de arriba resulta en igualar `Delay` a 500, como antes, pero la expresión puede hacer el código mas sencillo de entender si el numero resultante no es solo un numero arbitrario.

El bloque **CON** también se usa para especificar parámetros globales, tales como parámetros del reloj del sistema. El ejemplo de abajo muestra como activar el modo de reloj a un cristal baja velocidad, el PLL a 8x, y especificar que la frecuencia del pin XIN es 4 MHz.

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _XINFREQ = 4_000_000
```

Ver **_CLKMODE**, Pág. 71, y **_XINFREQ**, Pág. 241, para descripciones de estos parámetros.

también pueden definirse valores de punto flotante como constantes. Valores de punto flotante son números reales (con componentes fraccionales) y están codificados en 32 bits a

diferencia de las constantes enteros. Para especificar una constante de punto flotante se debe dar una clara indicación que el valor es de punto flotante; la expresión puede hacerse de punto flotante sencillo o completamente de punto flotante (no enteros).

Los valores de punto flotante deben escribirse como:

1) Dígitos decimales seguidos por punto decimal y al menos un digito decimal mas,
2) Dígitos decimales seguido por “e” (para exponente) y un valor exponente entero o,
3) Una combinación de 1 y 2.

A continuación ejemplos de contantes validas:

`0.5`	Valor de punto flotante
`1.0`	Valor de punto flotante
`3.14`	Valor de punto flotante
`1e16`	Valor de punto flotante
`51.025e5`	Valor de punto flotante
`3 + 4`	expresión Entera
`3.0 + 4.0`	expresión de punto flotante
`3.0 + 4`	expresión invalida, ocasiona error de compilación
`3.0 + FLOAT(4)`	expresión de punto flotante

Este es un ejemplo declarando un entero contante y dos constantes de punto flotante.

```
CON
  Num1 = 20
  Num2 = 127.38
  Num3 = 32.05 * 18.1 - Num2 / float(Num1)
```

El código de arriba activa `Num1`, `Num2` y `Num3` a 20, 127.38 y 573.736, respectivamente. Note que la ultima expresión requiere que `Num1` este encerrado en la declaración **`FLOAT`** para que el compilador lo trate como valor de punto flotante.

El compilador Propeller maneja las constantes de punto flotante como un numero real de precisión simple como describe el Standard IEEE-754. números reales de precisión sencilla se almacenan en 32 bits con un bit de signo, un exponente de 8-bit, y una mantisa de 23-bit (la parte fraccional). Esto proporciona aproximadamente 7.2 dígitos decimales significativos.

En operaciones de punto flotante en tiempo real, los objetos FloatMath, FloatString, Float32, y Float32Full ofrecen funciones matemáticas compatibles con números de precisión sencilla.

Vea FLOAT en Pág. 111, ROUND en Pág. 202, TRUNC en Pág. 213, y los objetos FloatMath, FloatString, Float32, y Float32Full para mayor información.

Enumeración (Sintaxis 2 y 3)

Los bloques constantes también pueden declarar símbolos constantes enumerados. Las enumeraciones son símbolos agrupados lógicamente que tienen incrementos de valores constantes enteros asignados para que sean únicos para el grupo. Por ejemplo, un objeto puede necesitar ciertos modos de operación. Cada uno de estos modos puede identificarse por un numero, 0, 1, 2 y 3 por ejemplo. Los números por si mismos no significan nada, solo tienen que ser únicos dentro del contexto del modo de operación. Como los números en si no son descriptivos, puede ser difícil recordar que hace el modo 3, pero es mas sencillo recordar significa el modo si tiene un nombre. Vea el siguiente ejemplo.

```
CON
  'Declara modos de operación
  RunTest    = 0
  RunVerbose = 1
  RunBrief   = 2
  RunFull    = 3
```

El código de arriba seria suficiente para nuestro propósito; ahora los usuarios de nuestro objeto pueden indicar "RunFull" en vez de "3" para especificar el modo de operación. El problema es que definiendo un grupo lógico de eventos de esta manera puede ocasionar problemas de mantenimiento ya que si un numero se cambio (a propósito o por accidente) sin cambiar el resto de acuerdo a, puede causar falla del programa. también imagine un caso donde hay 20 modos de operación. Eso podría ser un numero mayor de constantes y mas oportunidades de falla en mantenimiento.

Las enumeraciones resuelven este problema incrementando automáticamente valores por símbolos. Podemos reescribir el ejemplo de arriba con sintaxis de enumeración como sigue:

```
CON    'Declara modos de operación
  #0, RunTest, RunVerbose, RunBrief, RunFull
```

Aquí #0, le dice al compilador que inicie contando desde 0 y que active el siguiente símbolo igual a ese valor. Después si hay símbolos adicionales que no especifican su propio valor (vía una '= expresión') automáticamente se le asigna el valor anterior mas 1. El resultado es que RunTest iguala a 0, RunVerbose iguala a 1, RunBrief iguala a 2 y RunFull iguala a 3. Para la

mayoría de los casos, los valores en si no importan; todo lo que importa es que tienen asignado un numero único. Definiendo valores de enumeración de esta forma tiene la ventaja de asegurar que los valores asignados son únicos y contiguos dentro del grupo.

Usando el ejemplo de arriba los métodos que usan pueden hacer cosas como lo que sigue (Asumiendo que `Mode` es un símbolo activado por una llamada de objeto):

```
  case Mode
    RunTest    : <código de prueba aqui >
    RunVerbose : <código abundante aqui >
    RunBrief   : <código breve aqui >
    RunFull    : <código completo aqui >
—o—

  if Mode > RunVerbose
    <brief and run mode code here>
```

Observe que estas rutinas no dependen del valor exacto del modo, pero si en la enumeración del modo de símbolo para comparaciones así como la posición del símbolo en relación a otros símbolos en la misma enumeración. Es importante escribir el código de esta forma para reducir posibles Bugs en cambios futuros.

Las enumeraciones no tienen que ser coma separada tampoco. Los siguientes también trabajan y dejan espacio para comentarios acerca de cada modo.

```
CON    'Declara modos de operación
  #0
  RunTest          'Corre en modo de prueba
  RunVerbose       'Corre en modo abundante
  RunBrief         'Corre con breves anuncios
  RunFull          'Corre en modo completo de producción
```

El ejemplo de arriba hace lo mismo que el ejemplo anterior, pero ahora se cuenta con espacio para comentarios que describen el propósito de cada modo sin perder la ventaja del incremento automático. Después, si hay necesidad de agregar un quinto modo simplemente se agrega a la lista en la posición que se necesite. Si hay necesidad de que la lista comience con un numero determinado solo se cambia el `#0` al valor que se necesite: `#1`, `#20`, etc.

Mejor aun, es posible enumerar el valor en medio de la lista.

```
CON
  'Declara modos de operación
  #1, RunTest, RunVerbose, #5, RunBrief, RunFull
```

Aquí, RunTest y RunVerbose son 1 y 2, respectivamente y RunBrief y RunFull son 5 y 6, respectivamente. Mientras esta característica puede ser útil, para mantener buenas practicas de programación deberá usarse solo en casos especiales.

Una forma mas recomendada para lograr el mismo resultado que en el ejemplo anterior es incluir campos *Offset* opcionales. El código previo pudo escribirse de la siguiente forma:

```
CON
  'Declara modos de operación
  #1, RunTest, RunVerbose[3], RunBrief, RunFull
```

Igual que antes, RunTest y RunVerbose son 1 y 2 respectivamente. El [3] seguido de RunVerbose hace que la enumeración actual de valor (2) se incremente en 3 antes del siguiente símbolo enumerado. El efecto de esto es también como antes, RunBrief y RunFull son 5 y 6, respectivamente. La ventaja de esta técnica sin embargo, es que los símbolos enumerados están activados relativamente entre ellos. Al cambiar la línea del valor inicial se produce un cambio relativo entre ellos. Por ejemplo cambiando el #1 a #4 genera que RunTest y RunVerbose sean 4 y 5, respectivamente y RunBrief y RunFull 8 y 9. En contraste si el ejemplo original el #1 se cambia a #4, ambos RunVerbose y RunBrief estarán activos en 5, causando posiblemente que el código use esos símbolos para comportarse diferente.

El valor *Offset* puede ser un valor con signo, pero solo afecta el valor inmediato; el valor enumerado se incrementa siempre en 1 después de un *Symbol* que no especifica un *Offset*. Si se desea empalmar valores se puede especificar un *Offset* de 0 o menor para lograr este efecto.

La sintaxis 3 es una variación de la sintaxis de enumeración. No especifica ningún valor inicial. Cualquier elemento definido de esta forma siempre comenzara con el primer símbolo igual a 0 (para bloques nuevos **CON**) o al siguiente valor enumerado relativo al previo (dentro del mismo bloque **CON**).

Alcance de Constantes

Las constantes simbólicas definidas en bloques constantes son globales al objeto en el cual se definieron pero no fuera de ese objeto. Esto quiere decir que las constantes pueden accesarse directamente desde cualquier punto dentro del objeto pero su nombre no entrara en conflicto con símbolos definidos en otros objetos padres o hijos.

Las constantes simbólicas pueden ser accesadas indirectamente por objetos padres, usando la sintaxis referente a constante.

Ejemplo:

```
OBJ
  Num : "Numbers"

PUB SomeRoutine
  Format := Num#DEC     'Activa formato a numero constante decimal
```

Aquí se declara un objeto, "Numbers," como el símbolo Num. Después un método se refiere a la constante numbers **DEC** con Num#DEC. Num es el objeto referencia, # indica que necesitamos accesar las contantes de ese objeto, y DEC es la constante que necesitamos dentro del objeto. Esta característica permite al objeto definir constantes para uso con ellos mismos y por objetos padres para accesar aquellas constantes libremente sin interferir con los símbolos que crearon ellos mismos.

CONSTANT

Directiva: Declara una expresión constante en línea para resolver en tiempo de compilación.

((PUB PRI))
CONSTANT (*ConstantExpression*)

Regresa: Valor resuelto de una expresión constante.

- *ConstantExpression* es la expresión constante deseada.

Explicación

El bloque CON puede usarse para crear constantes desde expresiones que son referenciadas desde diversos lugares en el código, pero hay ocasiones en que la expresión constante se necesita temporalmente, para una sola vez. La instrucción CONSTANT se usa para resolver un método en línea, expresión constante en tiempo de compilación. Sin el uso de la instrucción CONSTANT, una expresión en línea del método se resuelve siempre en tiempo real, aun si la expresión es siempre un valor constante.

Usando CONSTANT

La instrucción CONSTANT puede crear expresiones de uso de una vez que ahorran espacio de código y velocidad en el tiempo de ejecución. Note en dos de los ejemplos siguientes:

Ejemplo 1, usando expresiones Standard de tiempo real:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(X+200 + cnt)                  'expresión de tiempo real
Standard
  !outa[0]
  waitcnt((X+Y)/2 + cnt)                'expresión de tiempo real
Standard
```

Ejemplo 2, el mismo de arriba pero con instrucción CONSTANT, con expresiones de tiempo real:

```
CON
  X = 500
  Y = 2500

PUB Blink
  !outa[0]
  waitcnt(constant(X+200) + cnt)  'exp c/partes en tiempo real
  !outa[0]
  waitcnt(constant((X+Y)/2) + cnt)'exp c/partes en tiempo real
```

Los dos ejemplos de arriba hacen exactamente lo mismo: sus métodos `Blink` cambian P0, esperan `X+200` ciclos, cambian P0 nuevamente y esperan `(X+Y)/2` ciclos antes de regresar. Mientras puede ser necesario que los símbolos `X` y `Y` del bloque **CON** se usen en diversos lugares dentro del objeto, las expresiones **WAITCNT** utilizadas en cada ejemplo del método `Blink` pueden ser usadas solo en un lugar. Por esta razón puede que no tenga sentido definir constantes adicionales en el bloque **CON** para partes como `X+200` y `(X+Y)/2`. No hay nada malo en colocar las expresiones en código de tiempo real, como en el ejemplo 1, pero la expresión completa se evalúa desafortunadamente en tiempo real y necesita tiempo extra y espacio de código.

La instrucción **CONSTANT** es perfecta para esta situación, porque resuelve completamente cada expresión constante de un solo uso, valor estático, ahorrando espacio de código y acelerando la ejecución. En el ejemplo 1 el método `Blink` consume 33 bytes de espacio de código mientras que el ejemplo 2 del método `Blink`, agregando la instrucción **CONSTANT** solo necesita 23 bytes. Observe que la parte "`+ cnt`" de las expresiones no están incluidas dentro de los paréntesis de la instrucción **CONSTANT**; esto es porque el valor de `cnt` es variable (`cnt` es el registro contador del sistema; ver **CNT**, Pág. 76) por lo que el valor no se puede resolver en tiempo de compilación.

Si una constante necesita ser utilizada en mas de un lugar del código, es mejor definirla en el bloque **CON** para que ese definida solo una vez y el símbolo que lo representa pueda usarse en diversas ocasiones.

Constantes (pre-definidas)

Las siguientes constantes son predefinidas por el compilador:

TRUE	Lógico Verdadero:	-1	($FFFFFFFF)
FALSE	Lógico Falso:	0	($00000000)
POSX	Máximo Entero Positivo:	2,147,483,647	($7FFFFFFF)
NEGX	Máximo Entero Negativo:	-2,147,483,648	($80000000)
PI	Valor de punto flotante para PI:	≈ 3.141593	($40490FDB)
RCFAST	Oscilador Interno Rápido:	$00000001	(%00000000001)
RCSLOW	Oscilador Interno Lento:	$00000002	(%00000000010)
XINPUT	Oscilador Reloj Externo:	$00000004	(%00000000100)
XTAL1	Cristal Externo Baja Velocidad:	$00000008	(%00000001000)
XTAL2	Cristal Externo Media Velocidad:	$00000010	(%00000010000)
XTAL3	Cristal Externo Alta Velocidad:	$00000020	(%00000100000)
PLL1X	Frecuencia Externa 1 vez:	$00000040	(%00001000000)
PLL2X	Frecuencia Externa 2 veces:	$00000080	(%00010000000)
PLL4X	Frecuencia Externa 4 veces:	$00000100	(%00100000000)
PLL8X	Frecuencia Externa 8 veces:	$00000200	(%01000000000)
PLL16X	Frecuencia Externa 16 veces:	$00000400	(%10000000000)

(Todas estas constantes también están disponibles en Ensamblador Propeller.)

TRUE y FALSE

TRUE y FALSE se usan normalmente para propósitos de comparación booleanas:

```
if (X = TRUE) o (Y = FALSE)
  <código si el total de condiciones es verdadero >
```

POSX y NEGX

POSX y NEGX se usan para propósitos de comparación o como bandera en evento especifico:

```
if Z > NEGX
  <código a ejecutar si Z no ha alcanzado el negativo mas pequeño
>
```

—o—

```
PUB FindListItem(Item) : Index
  Index := NEGX                 'Default a "not found" response
  <código para encontrar un punto en la lista>
  if <item found>
    Index := <items index>
```

PI

PI puede usarse para calcular de punto flotante, constantes de punto flotante o variables de punto flotante usan los objetos FloatMath y FloatString.

RCFAST a través de PLL16X

RCFAST a través de PLL16X son parámetros contantes del modo clock. Se explican en detalle en la sección _CLKMODE que comienza en la pagina 71.

Note que hay constantes enumeradas y no son equivalentes al valor correspondiente del registro CLK. Ver Registro CLK en Pág. 28 para información de como la constante se relaciona a los bits del registro CLK.

CTRA, CTRB

Registro: Registros de Control Contador A y Contador B.

((PUB PRI))
  CTRA

((PUB PRI))
  CTRB

Regresa: Valor actual del Registro de Control del Contador A o Contador B, si se usa como una fuente variable.

Explicación

CTRA y CTRB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores de los Cog. Cada cog tiene dos módulos contadores idénticos (A y B) que pueden desarrollar tareas repetitivas. Los registros CTRA y CTRB contienen la configuración de los módulos Contadores A y B respectivamente.

La siguiente discusión usa CTRx, FRQx y PHSx para referirse a los dos pares de cada registro A y B.

Cada uno de los dos módulos contadores puede controlar o monitorear hasta dos pines de E/S y desarrollar acumulaciones condicionales de 32-bit del valor en el registro FRQx en el registro PHSx por cada ciclo de reloj. Cada modulo contador tiene su propio ciclo de fase cerrada (PLLx) el cual puede usarse para sintetizar frecuencias de 64 MHz a 128 MHz.

Con solo un poco de configuración y en algunos casos un poco de mantenimiento del cog los módulos contadores pueden usarse para:

- Síntesis de Frecuencia
- Frecuencia de Mediciones
- Conteo de Pulsos
- Medición de Pulsos
- Medición de estado de Multi Pins
- Modulación ancho de pulso (PWM)
- Medición de Ciclo de Tarea
- Conversión Digital Análoga(DAC)
- Conversión Análoga Digital (ADC)
- Y mas.

Para alguna de estas operaciones el cog puede activar la configuración del contador a través de CTRA o CTRB, y desarrollara sus tareas completamente independiente. Para otras, el cog puede usar WAITCNT para alinear en tiempo la lecto escritura del contador y escribir en un ciclo; creado el efecto de un estado mas complejo de maquina. Como el periodo de

actualización del contador puede ser breve (12.5 ns a 80 MHz), la generación de una señal muy dinámica y su medición es posible.

Control de Campos de Registros

Cada registro CTRA y CTRB contiene cuatro campos mostrados en la tabla.

Tabla 2-5: Registros CTRA y CTRB

31	30..26	25..23	22..15	14..9	8..6	5..0
-	CTRMODE	PLLDIV	-	BPIN	-	APIN

APIN

El campo APIN de CTRA selecciona un pin E/S primario para ese contador. Puede ignorarse si no se usa. %0xxxxx = Puerto A, %1xxxxx = Puerto B (reservado para uso futuro). En ensamblador Propeller el campo APIN puede escribirse usando la instrucción MOVS.

Note que al escribir un cero a CTRA inmediatamente deshabilitara el contador A y detendrá todos los pins de salida relacionados y la acumulación PHSA.

BPIN

El campo BPIN de CTRx selecciona un pis E/S secundario para ese contador. Puede ignorarse si no se usa. %0xxxxx = Puerto A, %1xxxxx = Puerto B (reservado para uso futuro). En Ensamblador Propeller el campo BPIN puede escribirse usando la instrucción MOVD.

PLLDIV

El campo PLLDIV de CTRx selecciona una salida PLLx, ver tabla. Esto determina cual división “power-of-two” de la frecuencia VCO se usara como la salida final PLLx (un rango de 500 KHz a 128 MHz). Este campo puede ignorarse si no se usa. En ensamblador Propeller el campo PLLDIV puede escribirse junto con CTRMODE, usando la instrucción MOVI.

Tabla 2-6: Campo PLLDIV

PLLDIV	%000	%001	%010	%011	%100	%101	%110	%111
Output	VCO ÷ 128	VCO ÷ 64	VCO ÷ 32	VCO ÷ 16	VCO ÷ 8	VCO ÷ 4	VCO ÷ 2	VCO ÷ 1

CTRMODE
El campo CTRMODE de CTRA y CTRB selecciona uno de 32 modos de operación, mostrados en la Tabla 2-7, para el correspondiente Contador A o Contador B. En ensamblador Propeller el campo CTRMODE puede escribirse junto con PLLDIV, usando la instrucción MOVI.

Los modos %00001 a %00011 hacen que la acumulación FRQx-a-PHSx, suceda cada ciclo de reloj. Esto hace un oscilador controlado numéricamente (NCO) en PHSx[31], el cual alimenta la entrada de referencia de PLLx. El PLLx multiplicara esta frecuencia por 16 usando su oscilador de voltaje controlado (VCO).

Para una operación estable es recomendable que la frecuencia VCO se mantenga dentro de 64 MHz a 128 MHz. Esto se convierte en una frecuencia NCO de 4 MHz a 8 MHz.

Usando CTRA y CTRB

En Spin, CTRx puede leer y escribirse igual que cualquier otro registro o variable pre definida. Tan pronto como se escribe el registro el Nuevo modo de operación tiene efecto en el contador. Por ejemplo:

```
CTRA := %00100 << 26
```

El código activa el campo CTRA de CTRMODE a modo NCO (%00100) y todos los demás bits a cero.

Tabla 2-7: Modos Contadores (CTRMODE Valores de Campo)				
CTRMODE	**Descripción**	**Acumulado FRQx a PHSx**	**APIN Salida***	**BPIN Salida***
%00000	Contador deshabilitado (apagado)	0 (never)	0 (none)	0 (none)
%00001 %00010 %00011	PLL interno (modo de video) PLL terminación sencilla PLL diferencial	1 (always) 1 1	0 PLLx PLLx	0 0 !PLLx
%00100 %00101	NCO terminación sencilla NCO diferencial	1 1	PHSx[31] PHSx[31]	0 !PHSx[31]
%00110 %00111	DUTY terminación sencilla DUTY diferencial	1 1	PHSx-Carry PHSx-Carry	0 !PHSx-Carry
%01000 %01001 %01010 %01011	POS detector POS detector con retroalimentación POSEDGE detector POSEDGE detector c/retroalimentación	A^1 A^1 A^1 & $!A^2$ A^1 & $!A^2$	0 0 0 0	0 $!A^1$ 0 $!A^1$
%01100 %01101 %01110 %01111	NEG detector NEG detector con retroalimentación NEGEDGE detector NEGEDGE detector c/retroalimentación	$!A^1$ $!A^1$ $!A^1$ & A^2 $!A^1$ & A^2	0 0 0 0	0 $!A^1$ 0 $!A^1$
%10000 %10001 %10010 %10011 %10100 %10101 %10110 %10111 %11000 %11001 %11010 %11011 %11100 %11101 %11110 %11111	LOGIC nunca LOGIC !A y !B LOGIC A y !B LOGIC !B LOGIC !A y B LOGIC !A LOGIC A <> B LOGIC !A | !B LOGIC A y B LOGIC A == B LOGIC A LOGIC A | !B LOGIC B LOGIC !A | B LOGIC A | B LOGIC siempre	0 $!A^1$ & $!B^1$ A^1 & $!B^1$ $!B^1$ $!A^1$ & B^1 $!A^1$ A^1 <> B^1 $!A^1$ | $!B^1$ A^1 & B^1 A^1 == B^1 A^1 A^1 | $!B^1$ B^1 $!A^1$ | B^1 A^1 | B^1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

***Debe Activar el bit DIR correspondiente para afectar al pin**
A^1 = APIN entrada retrasada por 1 clock
A^2 = APIN entrada retrasada por 2 clock
B^1 = BPIN entrada retrasada por 1 clock

DAT

Denominador: Declara un bloque Data

DAT
〈*Symbol*〉 *Alignment* 〈*Size*〉 〈*Data*〉 〈[*Count*]〉 〈, 〈*Size*〉 *Data* 〈[*Count*]〉〉...

DAT
〈*Symbol*〉 〈*Condition*〉 *Instruction Operands* 〈*Effect(s)*〉

- *Symbol* es un nombre opcional para el dato, espacio reservado o instrucción que sigue.
- *Alignment* es la alineación deseada y tamaño por defecto (BYTE, WORD, o LONG) de los elementos data que le siguen.
- *Size* es el tamaño deseado (BYTE, WORD, o LONG) del elemento data que lo sigue inmediatamente; la alineación no cambia.
- *Data* es una expresión constante o lista de coma separada de expresiones constantes. también son permitidas cadenas de caracteres; son tratados como lista de caracteres de coma separada.
- *Count* es una expresión opcional indicando el numero de entradas de tamaño byte-, word-, o long- de *Data* para almacenar en la tabla data.
- *Condition* es una condición de lenguaje ensamblador, IF_C, IF_NC, IF_Z, etc.
- *Instruction* es una instrucción de lenguaje ensamblador, ADD, SUB, MOV, etc.
- *Operands* es cero, uno o dos operandos según requiere la instrucción.
- *Effect(s)* es/son uno dos o tres efectos de lenguaje ensamblador que hacen escribir o no el resultado de una instrucción, NR, WR, WC, o WZ.

Explicación

Un bloque DAT (Data) es una sección de código fuente que contiene datos pre definidos, memoria reservada para uso en tiempo de ejecución y código ensamblador Propeller. Este es uno de los seis denominadores (CON, VAR, OBJ, PUB, PRI, y DAT) que proporcionan una estructura inherente al lenguaje spin.

Los bloques de datos son secciones multi propósito de código fuente que se usan para tablas de datos, espacio en tiempo de ejecución y código ensamblador Propeller. Datos y código pueden mezclarse si es necesario, los datos se cargan en un cog con el código ensamblador.

Declaración de Data(Sintaxis 1)

Data se declara con una alineación especifica y tamaño (BYTE, WORD, o LONG) para indicar como debe almacenarse en memoria. La dirección donde se almacenan los datos depende de la estructura del objeto y la aplicación donde se compila ya que los datos se incluyen como parte del código compilado.

Por ejemplo:

```
DAT
  byte 64, "A", "String", 0
  word $FFC2, 75000
  long $44332211, 32
```

En las primeras dos líneas de este ejemplo, BYTE, indica que el dato que lo sigue debe ser alineado byte y tamaño byte. En tiempo de compilación el dato seguido de BYTE, 64, "A", etc., se almacena en memoria de programa un byte a la vez empezando en la siguiente localidad disponible. La línea tres especifica una alineación Word y datos de tamaño word. Los datos, $FFC2 y 75000, comenzaran en la siguiente frontera de la posición word siguiendo a los datos que aparecen antes; con cualquier byte no usados de los datos previos llenados con ceros para ajustar a la siguiente frontera word. La cuarta línea especifica alineación long y datos de tamaño long; sus datos se almacenaran en la siguiente frontera de long siguiendo a los datos word que aparecen anteriormente, llenando con ceros para completar hasta la frontera. La Tabla 2-8 muestra como s ve esto en memoria (mostrado en hexadecimal).

Tabla 2-8: Ejemplo de Datos en Memoria

L	0				1				2				3				4				5			
W	0		1		2		3		4		5		6		7		8		9		10		11	
B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
D	40	41	53	74	72	69	6E	67	00	00	C2	FF	F8	24	00	00	11	22	33	44	20	00	00	00

L = longs, W = words, B = bytes, D = data

Los primeros nueve bytes (0 – 8) son los datos byte de la primer línea; $40 = 64 (decimal), $41 = "A", $53 = "S", etc. El byte 9 se ajusta con ceros para alinear la primer palabra word de los datos word, $FFC2, en el byte 10. Los bytes 10 y 11 (word 5) contiene el primer valor de tamaño word, $FFC2, almacenado en formato bit-bajo-primero como $C2 y $FF. Los bytes 12 y 13 (word 6) son la parte baja de 75000; mayor explicación después. Los Bytes 14 y 15 (word 7) se ajustan con ceros para alinear al primer dato long, $44332211. Los bytes 16 a 19 (long 5) contienen el valor en formato bit-bajo-primero. Finalmente los bytes 20 al 23 (long 6) contienen el Segundo dato long, 32, en formato bit-bajo-primero.

Quizá noto que el valor 75000 se especifico en tamaño word. El numero 75000 es $124F8 en hexadecimal, pero como es mayor que un word, solo los datos del word mas bajo ($24F8) se almaceno. Esto resulta en word 6 (bytes 12 y 13) que contienen $F8 y $24, y en word 7 (bytes 14 y 15) que contiene $00 y $00 debido al ajuste a ceros para el siguiente valor alineado long.

Este fenómeno, intencional o no, sucede de igual forma para alineación byte – tamaño byte también, por ejemplo:

```
DAT
  byte $FFAA, $BB995511
```

...el resultado es que solo los bytes bajos de cada valor, $AA y $11 se almacenaran en localidades consecutivas.

Ocasionalmente, sin embargo, se desea almacenar un valor mayor entero como unidades elementales mas pequeñas que no están necesariamente alineadas de acuerdo al tamaño del valor mismo. Para hacer esto especifique el tamaño del valor justo antes del valor.

```
DAT
  byte word $FFAA, long $BB995511
```

Este ejemplo especifica datos alineados como bytes, pero con un valor de tamaño word seguido de un valor long. El resultado es que en la memoria contiene $AA y $FF, consecutivamente y después de eso, $11, $55, $99 y $BB.

Si modificamos la línea 3 del primer ejemplo de arriba como sigue:

```
  word $FFC2, long 75000
```

... tendremos al final $F8, $24, $01, y $00 ocupando los bytes 12 al 15. El byte 15 es el byte alto del valor y se colocara a la izquierda inmediatamente después de la siguiente frontera long así que no hay ceros adicionales para completar para la alineación a long.

Opcionalmente el campo *Symbol* de la sintaxis 1 puede incluirse al “nombre” del dato. Con esto se puede referenciar el dato de un bloque `PUB` o `PRI` fácilmente. Por ejemplo:

```
DAT
  MyData    byte $FF, 25, %1010

PUB GetData | Temp
  Temp := MyData[0]             'Obtiene el primer byte de una tabla
de datos
```

Este ejemplo crea una tabla de datos llamada MyData que consiste en bytes $FF, 25 y %1010. El método publico GetData lee el primer byte de MyData desde memoria principal y lo almacena en su variable local Temp.

también puede usar declaraciones **BYTE**, **WORD**, y **LONG** para leer localidades de memoria principal. Por ejemplo:

```
DAT
  MyData    byte $FF, 25, %1010

PUB GetData | Temp
  Temp := BYTE[@MyData][0]      Obtiene el primer byte de una tabla
de datos
```

Este ejemplo es similar al anterior excepto que usa la declaración **BYTE** para leer el valor almacenado en la dirección de MyData. Vea **BYTE**, Pág. 54; **WORD**, Pág. 232; y **LONG**, Pág. 132, para mayor información sobre lectura y escritura en memoria principal.

Declarando repetición de Datos (Sintaxis 1)

Pueden repetirse listas de datos usando al campo opcional *Count*. Por ejemplo:

```
DAT
  MyData      byte  64, $AA[8], 55
```

El ejemplo de arriba declara una tabla de datos de tamaño byte, alineación byte llamada MyData, consiste en diez valores: 64, $AA, $AA, $AA, $AA, $AA, $AA, $AA, $AA, 55. Hay ocho repeticiones de $AA debido al [8] en la declaración inmediata después de $AA.

Escribiendo código Ensamblador Propeller (Sintaxis 2)

Además de los datos numéricos y de cadena, el bloque **DAT** también se usa para código ensamblador Propeller. El siguiente ejemplo cambia el pin 0 cada ¼ de segundo.

```
DAT
                org 0                   'Reactiva el apuntador
ensamblador
Toggle          rdlong  Delay, #0       'Obtiene frecuencia de
reloj
                shr     Delay, #2       'Divide por 4
                mov     Time,  cnt      'Obtiene el tiempo actual
                add     Time,  Delay    'Ajusta por 1/4 de segundo
                mov     dira,  #1       'Activa pin 0 a salida
```

```
Loop              waitcnt  Time,  Delay    'Espera   de segundo
                  xor      outa,  #1       'cambia pin
                  jmp      #Loop           'regresa ciclo

Delay     res     1
Time      res     1
```

Cuando una aplicación Propeller arranca inicialmente solo se ejecuta código Spin. En cualquier tiempo, sin embargo, el código Spin puede escoger iniciar código ensamblador en un cog por si mismo. Los comandos **COGNEW** (Pág. 81) y **COGINIT** (Pág. 79) se usan para este propósito. El siguiente ejemplo de código Spin inicia el código ensamblador `Toggle` mostrado arriba.

```
PUB Main
  cognew(@Toggle, 0)                          'Inicia código Toggle
```

La instrucción **COGNEW**, de arriba le dice al chip Propeller que inicie el código ensamblador `Toggle` en un cog nuevo. El Propeller entonces puede encontrar un cog disponible, copia el código del bloque **DAT** iniciando `Toggle` en la RAM del cog, y luego inicia el cog el cual comienza a ejecutar el código de la localidad 0 de la Memoria.

Un bloque **DAT** puede contener múltiples programas en ensamblador Propeller, o múltiples bloques **DAT** pueden contener programas ensamblador individuales, pero en ambos casos, cada programa ensamblador deberá comenzar con una directiva **ORG** (Pág. 338) para reiniciar el apuntador ensamblador apropiadamente.

Comandos Dobles

Los lenguajes Spin y Ensamblador Propeller comparten un numero de instrucciones llamadas comandos dobles. Estos comandos dobles desarrollan tareas similares pero cada uno tiene diferente sintaxis o estructura que es similar al lenguaje en el que se escribe; Spin vs. Ensamblador Propeller. Cualquier comando doble que se usa en un bloque **DAT** se considera instrucción ensamblador. Por otro lado cualquier comando doble escrito en un bloque **PUB** y **PRI** se considera instrucción Spin.

DIRA, DIRB

Registro: Registro de Dirección para puertos A y B 32-Bit.

((PUB PRI))
DIRA ⟨[*Pin(s)*]⟩

((PUB PRI))
DIRB ⟨[*Pin(s)*]⟩ (Reserved for future use)

Regresa: El valor actual de la dirección de bits para pin(s) E/S en puertos A o B, si se usan como fuente variable.

- *Pin(s)* es una expresión opcional o una expresión de rango que especifica el pin E/S o pins para accesar en Puerto A (0-31) o Puerto B (32-63). Si se da como una expresión sencilla solo se acceso el pin especificado. Si se da como una expresión de rango (dos expresiones en formato de rango; x..y) se accesan los pins marcados del inicio al final de las expresiones.

Explicación

DIRA y DIRB son uno de seis registros (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pins E/S. El registro DIRA mantiene el estado de la dirección para cada uno de los 32 pins E/S en el Puerto A; los bits 0 al 31 corresponden de P0 a P31. El registro DIRB mantiene el estado de la dirección para cada uno de los 32 pins E/S en el Puerto B; los bits 0 a 31 corresponden de P32 a P63.

NOTA: DIRB esta reservado para uso futuro; El Propeller P8X32A no incluye pins E/S del Puerto B por lo que en adelante solo se discutirá DIRA.

DIRA se usa para activar y obtener el estado actual de la dirección de uno o mas pins E/S del puerto A. Un bit bajo (0) activa el pin correspondiente como dirección de entrada. Un bit alto (1) activa el pin E/S correspondiente como dirección de salida. Todos los bits de registros DIRA están puestos a cero por defecto al arrancar el cog; todos los pins E/S se especifican como entradas por ese cog hasta que el código indica lo contrario.

Cada cog tiene acceso a todos los pins E/S en cualquier tiempo. Esencialmente todos los pins E/S están conectados directamente a cada cog de esta forma no hay un hub relacionado mutuamente con acceso mutuamente exclusivo. Cada cog mantiene su propio registro DIRA que le da la habilidad de activar cualquier dirección del pin E/S. Cada registro DIRA del cog es OR con el registro DIRA de otros cogs y el resultante valor de 32-bit se convierte en las direcciones E/S del puerto A del pin P0 a P31. El resultado es que el estado de cada dirección

del pin de E/S es el “cableado OR” del cog entero. Vea Pins E/S en la Pág. 26 para mas información.

Esta configuración puede describirse fácilmente en las siguientes reglas:

A. Un pin es solo entrada si un cog no lo activa como salida.
B. Un pin es una salida si cualquier cog activo lo activa como salida.

Si se deshabilita un cog su registro de dirección se trata como si se limpiara a 0, ocasionando que no se haga uso de su influencia en la dirección del pin E/S y el estado.

Note que debido a la naturaleza “wired-OR” de los pins de E/S no es posible una contención eléctrica entre los pins, ellos pueden accesar los pins E/S simultáneamente. Depende del desarrollador de la aplicación asegurar que no hay dos cogs ocasionando una contención lógica en el mismo pin E/S durante el tiempo de ejecución.

Usando DIRA

Activa o limpia los bits en **DIRA** afectando la dirección deseada de los pins E/S. Por ejemplo:

```
DIRA := %00000000_00000000_10110000_11110011
```

El código de arriba active el registro **DIRA** (los 32 bits al mismo tiempo) para un valor que activa los pins E/S 15, 13, 12, 7, 6, 5, 4, 1 y 0 a salidas y el resto a entradas.

Usando los operadores unarios post-clear (**~**) y post-set (**~~**), el cog puede activar todos los pins E/S a entradas o salidas respectivamente; sin embargo no es usual o deseado activar todos los pins E/S a salidas. Por ejemplo:

```
DIRA~                   'Limpia registro DIRA (todas E/S son
entradas)
```

—y—

```
DIRA~~                  'Activa registro DIRA (todas E/S son salidas)
```

El primer ejemplo de arriba limpia el registro entero **DIRA** a cero (los 32 bits al mismo tiempo) todas las E/S de P0 a P31 son entradas. El segundo ejemplo activa el registro **DIRA** a unos (los 32 bits al mismo tiempo); todas las E/S de P0 a P31 son salidas.

Para afectar solo un pin de E/S (un bit), incluya el campo del pin(s) opcional. Esto trata al registro **DIRA** como un arreglo de 32 bits.

```
DIRA[5]~~               'Bit 5 de DIRA (P5 a salida)
```

Esto active P5 a salida. Todos los demás bits de **DIRA** (y por lo tanto todos los demás pins E/S) se quedan en el estado previo.

El registro **DIRA** soporta una forma especial de expresión llamada expresión de rango, el cual permite afectar un grupo de pins E/S al mismo tiempo sin afectar a otros fuera del rango especificado. Para afectar pins E/S múltiples o seguidos al mismo tiempo use una expresión de rango (como x..y) en el campo de pin(s).

```
DIRA[5..3]~~          'Activa bits 5 a 3 de dira (P5-P3 a salida)
```

Esto active P5, P4 y P3 a salidas y los otros bits del registro **DIRA** permanecen en su estado previne. Aquí otro ejemplo:

```
DIRA[5..3] := %110    'Activa P5 y P4 a salida, P3 a entrada
```

El ejemplo de arriba activa 5, 4 y 3 de **DIRA** a 1, 1 y 0 respectivamente, dejando todos los demás bits en el estado previo. Consecuentemente P5 y P4 ahora son salidas y P3 es entrada.

IMPORTANTE: El orden de los valores en la expresión de rango afecta según es utilizado. Por ejemplo, en los siguientes ejemplos el orden de la expresión de rango intercambia el orden del ejemplo anterior.

```
DIRA[3..5] := %110    'Activa P3 y P4 a salida, P5 a entrada
```

Aquí los bits 3, 4 y 5 se igualan a 1, 1, y 0 en el registro **DIRA**, hacienda P3 y P4 salidas y P5 una entrada.

Las expresiones de rango son una característica ponderosa, pero si no se tiene cuidado, también puede causar extraños comportamientos y resultados no intencionados.

Normalmente **DIRA** solo se escribe pero también puede leer para obtener la lectura de los estados actuales de E/S. El siguiente ejemplo asume que `Temp` es una variable creada en cualquier otro lado:

```
Temp := DIRA[7..4]    'Obtiene la dirección de P7 a P4
```

El ejemplo de arriba active `Temp` igual a los bits 7, 6, 5 y 4 de **DIRA**; por ejemplo, los 4 bits mas bajos de `Temp` ahora son iguales a `DIRA7:4` y los otros bits de `Temp` se limpian a cero.

FILE

Directiva: Importa un archivo externo como datos.

```
DAT
  FILE "FileName"
```

- *FileName* es el nombre sin extensión del archivo de datos deseado. Durante la compilación un archivo con este nombre se busca en el tabulador del editor, el directorio de trabajo y el directorio de la librería. *FileName* puede contener cualquier carácter valido de nombre de archivo; no se permiten \, /, :, *, ?, ", <, >, y |.

Explicación

La directiva FILE se usa para importar un archiva de datos externo (normalmente un archivo binario) en el bloque DAT de un objeto. Los datos pueden accesarse por el objeto justo como cualquier bloque de datos DAT.

Usando FILE

FILE se usa en bloques DAT similar a como se usaría BYTE excepto que el nombre del archivo va entre comillas en vez de los datos de valores. Por ejemplo:

```
DAT
  Str    byte  "Esta es la cadena de datos.", 0
  Data   file  "Datafile.dat"
```

En este ejemplo el bloque DAT se compone de una cadena byte seguida por los datos de un archive llamado Datafile.dat. Durante la compilación la herramienta Propeller buscara a través del editor, el directorio de trabajo o el directorio de la librería por un archivo llamado Datafile.dat y cargara sus datos en el primer byte seguid de la terminación cero de la cadena Str. Los métodos pueden accesar lo datos importados usando la declaración BYTE, WORD o LONG como lo harían en datos normales. Por ejemplo:

```
PUB GetData | Index, Temp
  Index := 0
  repeat
    Temp := byte[Data][Index++] 'Lee datos en Temp 1 byte a la vez
    <hace algo con Temp >     'desarrolla tareas con el valor Temp
  while Temp > 0              'cicla hasta encontrar fin
```

Este ejemplo leerá los datos importados, un byte a la vez hasta encontrar un byte igual a 0.

FLOAT

Directiva: Convierte un entero constante a valor punto flotante en tiempo de compilación.

((CON VAR OBJ PUB PRI DAT))
FLOAT (*IntegerConstant*)

Regresa: Valor resuelto de la expresión entera a numero de punto flotante.

- *IntegerConstant* es el entero que se desea convertir para usarse como valor de punto flotante.

Explicación

FLOAT es una de las tres directivas (FLOAT, ROUND y TRUNC) usadas para expresiones de constantes de punto flotante. La directiva FLOAT convierte una constante entera en un valor de punto flotante.

Usando FLOAT

Mientras la mayoría de las constantes son enteros de 32 bits, el compilador Propeller soporta valores de punto flotante de 32-bit y expresiones constantes para uso en tiempo de compilación. Note que esto es para expresiones constantes únicamente, no para expresiones variables en tiempo de ejecución.

Para declaraciones típicas de punto flotante la expresión debe mostrarse como un valor de punto flotante en una de tres formas: 1) Como un entero seguido de un punto decimal con al menos un digito, 2) como un entero con una E seguida de un valor exponencial o 3) ambos 1) y 2). Por ejemplo:

```
CON
  OneHalf = 0.5
  Ratio   = 2.0 / 5.0
  Miles   = 10e5
```

Los códigos mostrados arriba crean tres valores de punto flotante. OneHalf es igual a 0.5, Ratio es igual a 0.4 y Miles es igual a 1,000,000.

Note que en el ejemplo los componentes de cada expresión se muestran como valor de punto flotante. Ahora observe el siguiente ejemplo:

```
CON
  Two     = 2
  Ratio   = Two / 5.0
```

Aquí, Two se define como una constante entera y Ratio aparece definida como una constante de punto flotante. Esto ocasiona un error en la línea Ratio porque para expresiones constantes de punto flotante cada valor en la expresión debe ser de punto flotante; no se puede mezclar enteros con valores de punto flotante como Ratio = 2 / 5.0.

Sin embargo se puede usar la directiva FLOAT para convertir un entero a valor de punto flotante como se muestra a continuación:

```
CON
  Two     = 2
  Ratio   = float(Two) / 5.0
```

La directiva FLOAT en este ejemplo convierte la constante entera Two en valor de punto flotante para que se pueda usar en una expresión de punto flotante.

Acerca de Punto Flotante

El compilador Propeller maneja constantes de punto flotante como numero real de precisión simple como se describe por el estándar IEEE-754. números reales de precisión simple se almacenan en 32 bits, con 1 bit de signo un exponente de 8 bits y una mantisa de 23 bits (la parte fraccional). Esto proporciona aproximadamente 7.2 dígitos decimales significativos.

Las expresiones de punto flotante pueden definirse y usarse para muchos propósitos de compilación pero no para tiempo de ejecución, los objetos FloatMath, FloatString, Float32, y Float32Full dan funciones matemáticas compatibles con números de precisión simple

Vea Asignación Constante ‘=’ en la sección de Operadores de la Pág. 152, ROUND en Pág. 202, y TRUNC en Pág. 213, así como los objetos FloatMath, FloatString, Float32, y Float32Full para mayor información.

_FREE

Constante: Pre-definida, constante ajustable de una sola vez para especificar el tamaño de espacio libre en la aplicación.

```
CON
  _FREE = expresión
```

- *expresión* es un expresión entera que indica el numero de longs a reservar de espacio libre.

Explicación

_FREE es una constante opcional pre definida ajustable una vez que especifica el espacio de memoria libre requerida por una aplicación. Este valor se suma a _STACK, si es especificado, para determinar el monto total de espacio en memoria libre para reservar en una aplicación Propeller. Use _FREE si una aplicación requiere un mínimo monto de memoria libre para correr apropiadamente. Si el resultado de la aplicación compilada es muy grande para permitir la memoria libre se marcara un mensaje de error, por ejemplo::

```
CON
  _FREE    = 1000
```

La declaración _FREE en el bloque CON indica que la aplicación necesita 1,000 longs de memoria libre después de la compilación. Si el resultado de la compilación no tiene ese espacio libre un mensaje de error indicara por cuanto se excedió. Esta es una buena forma de prevenir compilaciones correctas de una aplicación pero que fallaran al correr por falta de memoria.

Note que solamente el objeto superior puede activar el valor de _FREE. Cualquier objeto hijo que declare _FREE será ignorado.

FRQA, FRQB

Registro: Registros de frecuencia del contador A y contador B .

((PUB PRI))
FRQA

((PUB PRI))
FRQB

Regresa: Valor actual del Registro de Frecuencia del Contador A o Contador B, si se usa como fuente variable.

Explicación

FRQA y FRQB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores del Cog. Cada cog tiene dos módulos contadores idénticos (A y B) que pueden desarrollar muchas tareas repetitivas. El registro FRQA contiene el valor que se acumula en el registro PHSA. El registro FRQB contiene el valor que se acumula en el registro PHSB. Ver CTRA, CTRB en Pág. 98para mayor información.

Usando FRQA y FRQB

FRQA y FRQB pueden leer y escribir al igual que otro registro o variable pre definida. Por ejemplo:

```
FRQA := $00001AFF
```

El código de arriba activa FRQA a $00001AFF. Dependiendo del campo CTRMODE del registro CTRA este valor en FRQA puede agregarse al registro PHSA a una frecuencia determinada por el reloj del sistema y el pin E/S primario y/o secundario. Ver CTRA, CTRB en Pág. 98 para mayor información.

IF

instrucción: Prueba condiciones y ejecuta un bloque de código si es valido (lógica positiva).

((PUB PRI))
IF *Condition(s)*
→ *IfStatement(s)*
〈 ELSEIF *Condition(s)*
→ *ElseIfStatement(s)* 〉...
〈 ELSEIFNOT *Condition(s)*
→ *ElseIfNotStatement(s)* 〉...
〈 ELSE
→ *ElseStatement(s)* 〉

- *Condition(s)* es una o mas expresiones booleanas a probar.
- *IfStatement(s)* es un bloque de una o mas líneas de código para ejecutar cuando la condición IF es verdadera.
- *ElseIfStatement(s)* es un bloque opcional de una o mas líneas a ejecutar cuando todas las condiciones previas son invalidas y la condición ELSEIF es verdadera.
- *ElseIfNotStatement(s)* es un bloque opcional de una o mas líneas a ejecutar cuando todas las condiciones previas son invalidas y la condición ELSEIFNOT es falsa.
- *ElseStatement(s)* es un bloque opcional de una o mas líneas a ejecutar cuando todas las condiciones previas son invalidas.

Explicación

IF es uno de tres comandos condicionales mayores (IF, IFNOT, y CASE) que condicionalmente ejecutan bloques de código. IF puede opcionalmente combinarse con uno o mas comandos ELSEIF, uno o mas comandos ELSEIFNOT, y/o un comando ELSE para formar estructuras condicionales sofisticadas.

IF prueba la condición y si es verdadera ejecuta *IfStatement(s)*.Si la condición es falsa se prueban las siguientes condiciones opcionales ELSEIF y/o la condición ELSEIFNOT en ese orden hasta que se encuentra una línea con una condición valida, entonces el bloque asociado *ElseIfStatement(s)* o *ElseIfNotStatement(s)* se ejecuta. El bloque opcional *ElseStatement(s)* se ejecuta si no se encontraron condiciones validas previas.

Una condición “valida” es una que evalúa a TRUE para una declaración condicional positiva (IF o ELSEIF) o evalúa a FALSE para una declaración condicional negativa (ELSEIFNOT).

Indentacion es Critica

IMPORTANTE: La indentacion es critica. El lenguaje Spin usa la indentacion (de un espacio o mas) de líneas que siguen instrucciones condicionales para determinar si corresponden a la instrucción o no. Para ver estos bloques de código lógicamente agrupados puede presionar Ctrl + I para ver los indicadores de grupos. Con Ctrl + I nuevamente deshabilitara la pantalla. Vea la ayuda en la herramienta Propeller para una lista completa de accesos rápidos.

instrucción Simple IF

La forma mas común de la instrucción condicional **IF** desarrolla una acción si una condición en verdadera. Esto se escribe como una instrucción **IF** seguida por uno o mas líneas indentadas de código. Por ejemplo:

```
if X > 10                   'Si X es mayor que 10
  !outa[0]                  'cambia P0
!outa[1]                    'cambia P1
```

Este ejemplo prueba si `X` es mayor que 10; si es, el pin E/S se cambia. Aun si la condición **IF** no es verdadera el pin de E/S se cambia en la siguiente instrucción.

Como la línea `!outa[0]` esta indentada de la línea **IF**, pertenece al bloque *IfStatement(s)* y se ejecuta solo si la condición **IF** es verdadera. La siguiente línea `!outa[1]` no esta indentada de la línea **IF**, por lo tanto se ejecuta aun cuando la condición **IF** sea o no verdadera. Esta es otra versión del mismo ejemplo:

```
if X > 10                   'si X es mayor que 10
  !outa[0]                  'cambia P0
  !outa[1]                  'cambia P1
waitcnt(2_000 + cnt)        'espera 2,000 ciclos
```

Este ejemplo es muy similar al primero excepto que ahora las dos líneas de código están indentadas de la instrucción **IF**. En este caso si `X` es mayor que 10, P0 cambia, luego P1 cambia y finalmente se ejecuta la línea `waitcnt`. Si por el contrario, `X` no fue mayor que 10, las líneas `!outa[0]` y `!outa[1]` se brincan (como están indentadas son parte del bloque *IfStatement(s)*) y se ejecuta la línea `waitcnt` (como no esta indentada no es parte del bloque *IfStatement(s)*).

Combinando Condiciones

El campo *Condition(s)* se evalúa como una condición booleanas simple, pero puede hacerse de mas de una expresión booleanas al combinarla con los operadores **AND** y **OR**; ver Pág. 171-172. Por ejemplo:

```
if X > 10 AND X < 100     'Si X es mayor que 10 y menor que 100
```

Esta instrucción **IF** seria verdadera si y solo si X es mayor que 10 y X es también menor que 100. En otras palabras es verdaderos X esta en el rango del 11 al 99. Algunas veces instrucciones como estas pueden ser un poco difíciles de leer. Para hacerlas mas sencillas se puede usar paréntesis para agrupar cada sub condicion tal como se hace a continuación:

```
if (X > 10) AND (X < 100)'Si X es mayor que 10 y menor que 100
```

Usando IF con ELSE

La segunda forma mas común de la instrucción condicional **IF** desarrolla una acción si una condición es verdadera o una acción diferente si la condición es falsa. Esto se escribe como una instrucción **IF** seguida por su bloque *IfStatement(s)*, después un **ELSE** seguido por su bloque *ElseStatement(s)* como se muestra a continuación:

```
if X > 100                'Si X es mayor que 100
  !outa[0]                'cambia P0
else                      'de lo contrario, X <= 100
  !outa[1]                'cambia P1
```

Aquí si X es mayor que 100, se cambia el pin 0 de E/S, de otra forma X debe ser menor o igual a 100 y el pin 1 E/S se cambia. Esta construcción **IF**...**ELSE** como esta escrita siempre realice un cambio de P0 o P1 nunca ambos y nunca ninguno.

Recuerde, el código lógicamente perteneciente al *IfStatement(s)* o al *ElseStatement(s)* debe estar indentado del **IF** o del **ELSE** respectivamente al menos un espacio. también note que **ELSE** debe estar alineado horizontalmente con la instrucción **IF**; ambos deben comenzar en la misma columna o el compilador no sabrá que el **ELSE** va con el **IF**.

Para cada instrucción **IF** puede existir cero o un componente **ELSE**. **ELSE** debe ser el ultimo componente en una instrucción **IF**, apareciendo después de cualquier potencial **ELSEIF**.

Usando IF con ELSEIF

La tercera forma de la instrucción condicional **IF** desarrolla una acción si una condición es verdadera o una acción diferente si la condición es falsa pero otra condición es verdadera, etc. Esto se escribe como una instrucción **IF** seguida por su bloque *IfStatement(s)*, luego uno o mas instrucciones **ELSEIF** seguidas por sus respectivos bloques *ElseIfStatement(s)*. Aquí un ejemplo:

```
if X > 100                'Si X es mayor que 100
  !outa[0]                'cambia P0
```

```
elseif X == 90              'de lo contrario si X = 90
  !outa[1]                  'cambia P1
```

Aquí si X es mayor que 100, el Pin 0 de E/S se cambia, de otra forma si X es igual a 90, el Pin 1 de E/S se cambia, y si ninguna de esas condiciones es verdad ninguno de los dos P0 y P1 se cambia. Esto es ligeramente un camino mas corto de escribir el siguiente código:

```
if X > 100                  'Si X es mayor que 100
  !outa[0]                  'cambia P0
else                        'de otra forma
  if X == 90                'si X = 90
    !outa[1]                'cambia P1
```

Ambos ejemplos desarrollan las mismas acciones, pero el primero es mas corto y usualmente mas sencillo de leer. Note que el **ELSEIF**, igual que el **ELSE**, debe estar alineado (iniciar en la misma columna) que el **IF** al que esta asociado.

Cada instrucción condicional **IF** puede tener cero o mas instrucciones **ELSEIF** asociadas. Observe el siguiente ejemplo:

```
if X > 100                  'Si X es mayor que 100
  !outa[0]                  'cambia P0
elseif X == 90              'de otra forma si X = 90
  !outa[1]                  'cambia P1
elseif X > 50               'de otra forma si X > 50
  !outa[2]                  'cambia P2
```

Tenemos tres condiciones y tres posibles acciones aqui. Igual que el ejemplo anterior si X es mayor que 100, P0 se cambia, de lo contrario si X es igual a 90, P1 se cambia, pero si ninguna de esas condiciones fue verdad X es mayor que 50, P2 se cambia. Si ninguna de esas condiciones fue verdadera entonces ninguna acción ocurre.

Aquí hay un concepto importante a notar. Si X es 101 o mayor, P0 se cambia, o si X es 90, P1 se cambia, o si X es 51 a 89, o 91 a 99, P2 se cambia. Esto es porque la condición **IF** y **ELSEIF** se pruebas una a la vez en el orden que se enlistaron y solo una, la primera condición que es verdadera, es la que ejecutara su código, ninguna otra acción se probara después de esto. Esto significa que si reacomodamos los dos **ELSEIF** para que "X > 50" se verifique primero tendríamos un error en nuestro código.

Hicimos esto abajo:

```
  if X > 100                'Si X es mayor que 100
    !outa[0]                'cambia P0
  elseif X > 50             'de otra forma si X > 50
    !outa[2]                'cambia P2
  elseif X == 90            'de otra forma si X = 90 <-- ERROR, POR
LA CONDICION DE ARRIBA.
    !outa[1]                'cambia P1  <-- SI REEMPLAZA
                             'ESTO EL CODIGO NUNCA
                             'CORRERA
```

El ejemplo de arriba contiene un error porque mientras `X` podría ser igual a 90, la condición `elseif X == 90` nunca se probara porque el anterior, `elseif X > 50`, se probara primero y como es verdad su bloque se ejecuta y ninguna otra condición de esa estructura **IF** se probara. Si `X` fuera 50 o menor, la ultima condición **ELSEIF** se prueba, pero por supuesto esto nunca será verdad.

Usando IF con ELSEIF y ELSE

Otra forma de la instrucción condicional **IF** es que desarrolla una o diferentes acciones si una o diferentes condiciones es verdadera o una acción diferente si ninguna de las condiciones previas fueron verdaderas. Esto se escribe con un **IF**, uno o mas **ELSEIFs**, y finalmente un **ELSE**. Aquí se muestra un ejemplo:

```
  if X > 100                'Si X es mayor que 100
    !outa[0]                'cambia P0
  elseif X == 90            'de lo contrario si x = 90
    !outa[1]                'cambia P1
  elseif X > 50             'de lo contrario si X > 50
    !outa[2]                'cambia P2
  else                      'de lo contrario
    !outa[3]                'cambia P3
```

Esto hace lo mismo que el ejemplo anterior a diferencia que si ninguna de las condiciones **IF** o **ELSEIF** son verdaderas P3 se cambia.

La condición ELSEIFNOT

La condición **ELSEIFNOT** se comporta exactamente como **ELSEIF** excepto que usa lógica negativa; ejecuta su bloque *ElseIfNotStatement(s)* solo si su expresión *Condition(s)* evalúa a

FALSO. Múltiples condiciones ELSEIFNOT y ELSEIF se pueden combinar en una instrucción sencilla condicional IF en cualquier orden entre el IF y el opcional ELSE.

IFNOT

instrucción: Prueba una condición y ejecuta un bloque de código si es válido (lógica negativa)

((PUB PRI))
IFNOT *Condition(s)*
→ *IfNotStatement(s)*
〈 ELSEIF *Condition(s)*
→ *ElseIfStatement(s)* 〉...
〈 ELSEIFNOT *Condition(s)*
→ *ElseIfNotStatement(s)* 〉...
〈 ELSE
→ *ElseStatement(s)* 〉

- *Condition(s)* es una o mas expresiones booleanas a probar
- *IfNotStatement(s)* es un bloque de uno o mas líneas de código a ejecutar cuando la condición IFNOT es falsa.
- *ElseIfStatement(s)* es un bloque opcional de código a ejecutar cuando todas las condiciones previas son invalidas y la condición ELSEIF es verdadera.
- *ElseIfNotStatement(s)* es un bloque opcional de código a ejecutar cuando todas las condiciones previas son invalidas y la condición ELSEIFNOT es falsa.
- *ElseStatement(s)* es un bloque opcional de código a ejecutar cuando todas las condiciones previas son invalidas.

Explicación

IFNOT es uno de las tres instrucciones mayores (IF, IFNOT, y CASE) que ejecutan condicionalmente un bloque de código. IFNOT es la forma complementaria de IF.

IFNOT prueba la condición y si es falsa ejecuta *IfNotStatement(s)*. Si la condición es verdadera la siguiente condición opcional ELSEIF y/o ELSEIFNOT *Condition(s)*, se prueban en orden hasta que una condición valida se encuentra, así se ejecuta el bloque asociado *ElseIfStatement(s)*, o *ElseIfNotStatement(s)*. El bloque opcional *ElseStatement(s)* se ejecuta si no se encuentran condiciones validas previamente.

Una condición valida evalúa FALSE para una instrucción condicional negativa (IFNOT, o ELSEIFNOT) o evalúa a TRUE para una instrucción condicional positiva (ELSEIF).

Ver IF en Pág. 115 para información de los componentes opcionales IFNOT.

INA, INB

Registro: Ingresa registros 32-bit para Puerto A y Puerto B.

((PUB PRI))
INA ⟨[*Pin(s)*]⟩

((PUB PRI))
INB ⟨[*Pin(s)*]⟩ (Reservado para uso futuro)

Regresa: Estado actual de pins E/S para Puerto A o B

- *Pin(s)* es una expresión opcional o una expresión de rango que especifica el pin E/S o los pins para accesar el Puerto A (0-31) o puerto B (32-63). Si se da como expresión sencilla solo se acceso el pin especificado. Si se da una expresión de rango (dos expresiones en un formato de rango x.y) se accesan los pins entre el inicio y final de la expresión.

Explicación

INA e INB son dos de seis registros (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pins de E/S. El registro INA contiene el estado actual de cada uno de los 32 pins E/S en el puerto A; los bits 0 a 31 corresponden de P0 a P31. El registro INB contiene el estado actual de cada uno de los 32 pins E/S en puerto B; los bits 0 al 31 corresponden de P32 a P63.

NOTA: INB esta reservado para uso futuro, el Propeller P8X32A no incluye pins de E/S para el Puerto B por lo que solo INA se considera en la discusión.

INA es solo lectura y no es realmente implementado como registro, es solo una dirección que cuando se acceso como una fuente en una expresión lee los pins de E/S del Puerto A directamente en ese momento. En el resultado un bit bajo (0) indica que el pin esta detectando tierra y un bit alto (1) indica que esta detectando voltaje VDD (3.3 volts). Como el Propeller es un componente CMOS los pins E/S sensan todo arriba de ½ VDD como alto, de esta forma significa que un alto es puede ser 1.65 volts o mas alto.

Cada cog tiene acceso a todos los pins E/S en cualquier momento. Esencialmente todos los pins E/S están directamente conectados a cada cog así que no hay acceso mutuamente exclusivo. Cada cog tiene su propio pseudo registro INA que le da la habilidad de leer el estado de los pins E/S (alto o bajo) en cualquier momento. EL valor actual del pin E/S se lee sin importar si su dirección es entrada o salida.

Note que debido a la naturaleza “wired-OR” de los pins E/S no hay conflicto eléctrico entre los cogs, todos ellos pueden accesar lo pins E/S simultáneamente. Depende del desarrollador de la aplicación asegurar que no hay dos cogs que ocasionan conflicto lógico en el mismo pin E/S durante el tiempo de ejecución. Como todos los cogs comparten todos los pins E/S, un cog puede usar **INA** para leer pins que esta usando así como los pins que están en uso por otro o por los demás cogs.

Usando INA

Leer **INA** para obtener el estado de los pins E/S en ese momento. Los siguientes ejemplos asumen que Temp se creo en otra parte.

```
Temp := INA                 'Obtiene estado de P0 a P31
```

Este ejemplo lee el estado de todos los 32 pins E/S del puerto A en Temp.

Usando el campo opcional *Pin(s)* el cog puede leer un pin E/S (un bit) a la vez. Por ejemplo:

```
Temp := INA[16]             'Obtiene el estado de P16
```

La línea de arriba lee el estado del Pin de E/S y almacena su estado (0 o 1) en el bit mas bajo de Temp; todos los otros bits de Temp se limpian.

En spin el registro **INA** soporta una forma de expresión especial que se llama expresión de rango, la cual permite leer un grupo de pins E/S al mismo tiempo sin leer otros fuera del rango especificado. Para leer múltiples pins seguidos de E/S a la vez use una expresión de rango (como x.y) en el campo *Pin(s)*.

```
Temp := INA[18..15]         'Obtiene el estado de P18:P15
```

Aquí los cuatro bits mas bajos de Temp (3, 2, 1, y 0) se activan a los estados de los pins E/S 18, 17, 16, y 15, respectivamente y todos los demás bits de Temp se limpian a 0.

IMPORTANTE: El orden de los valores en las expresiones de rango afecta su uso. Por ejemplo, el siguiente ejemplo intercambia el oren del rango del ejemplo previo.

```
Temp := INA[15..18]         'Obtiene el estado de P15:P18
```

Aquí los bits 3, 2, 1, y 0 de Temp se activan al estado de los pins de E/S 15, 16, 17, y 18, respectivamente. Esta es una característica poderosa para las expresiones de rangos, pero si no se tiene cuidado puede causar extraños comportamientos y resultados no deseados.

LOCKCLR

instrucción: Limpia el seguro a falso y obtiene su estado previo.

((PUB PRI))
LOCKCLR (*ID*)

Regresa: Estado previo de lock (TRUE o FALSE).

- *ID* es el ID (0 – 7) del lock para limpiar a falso.

Explicación

LOCKCLR es uno de cuatro comandos (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) que se usan para administrar recursos que son definidos por usuario y se consideran mutuamente exclusivos. LOCKCLR limpia el ID de Lock a FALSE y recupera el estado previo de ese lock (TRUE o FALSE).

Vea Acerca de los Seguros, Pág. 126, y Reglas Sugeridas para Seguros, Pág. 127 para información de los típicos usos de locks e instrucciones LOCKxxx.

Lo siguiente da por entendido que un cog (ya sea este u otro) utilizo un lock usando LOCKNEW y compartió el ID con este cog el cual se grabo como SemID. también asume que este cog tiene un arreglo de longs que se llama LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID) 'espera hasta asegurar el recurso
  repeat Idx from 0 to 9          'lee los 10 longs del recurso
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                  'desbloquea el recurso

PUB WriteResource | Idx
  repeat until not lockset(SemID) 'espera hasta bloquear el recurso
  repeat Idx from 0 to 9          'esribe 0 longs al recurso
    long[Idx] := LocalData[Idx]
  lockclr(SemID)                  'desbloquea el recurso
```

Ambos métodos, ReadResource y WriteResource, siguen las mismas reglas antes y después de accesar el recurso. Primero esperan indefinidamente al primer ciclo REPEAT hasta que se bloquea el recurso; ejemplo: activo exitosamente el candado asociado. Si LOCKSET regresa TRUE, la condición "until not lockset…" es FALSE, lo cual significa que algún otro cog esta actualmente accesando el recurso, así que el primer ciclo repeat intenta otra vez. Si LOCKSET regresa FALSE, la condición "until not lockset…" es verdadera lo que significa que hemos

"bloqueado el recurso" y el primer ciclo repeat termina. El segundo ciclo **REPEAT** en cada método lee o escribe el recurso a través de las instrucciones `long[Idx]` y `LocalData[Idx]`. La última línea de cada método, `lockclr(SemID)`, limpia los seguros asociados de los recursos a **FALSE**, lógicamente desbloqueando o liberando los recursos para que otros los usen.

Vea **LOCKNEW**, Pág. 126; **LOCKRET**, Pág. 129; y **LOCKSET**, Pág. 130 para mas información.

LOCKNEW

instrucción: Verifica un nuevo seguro y obtiene su ID.

```
((PUB   PRI))
  LOCKNEW
```

Regresa: ID (0-7) del seguro que se verificó o -1 si ninguno esta disponible.

Explicación

LOCKNEW es uno de cuatro comandos de seguros (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) utilizado para administrar recursos que son definidos por usuario y se consideran mutuamente exclusivos. LOCKNEW verifica un seguro único del Hub y obtiene el ID de ese seguro. Si no hay seguros disponibles LOCKNEW regresa -1.

Acerca de los Seguros

Un seguro es un mecanismo de semáforo que se usa para comunicarse entre dos o mas entidades. En el chip Propeller un seguro es simplemente uno de ocho bits globales en un registro protegido dentro del Hub. El hub mantiene un inventario de cuales seguros están en uso y sus estados actuales. Los Cogs pueden verificar, activar, limpiar y regresar seguros según se necesite durante el tiempo de ejecución para indicar si un elemento compartido, tal como un bloque de memoria, esta disponible o no. Como los seguros se administran a través del Hub solamente un Cog puede afectarlos a la vez, haciendo esto un mecanismo efectivo de control.

En aplicaciones donde dos o mas cogs están compartiendo la misma memoria, una herramienta como un seguro puede ser requerido para prevenir una colisión catastrófica. El hub previene que ocurran colisiones en datos elementales (tales como byte, word o long) en cada momento en el tiempo, pero no puede prevenir colisiones "lógicas" en bloques de múltiples elementos (tales como bloques de bytes, words y longs o una combinación de estos). Por ejemplo si dos o mas cogs están compartiendo un byte sencillo de memoria principal , cada uno tiene garantizado el acceso exclusivo a ese byte por naturaleza del hub. Pero si esos dos cogs comparten múltiples bytes de memoria principal el hub no puede prevenir que un cog escriba alguno de esos bytes mientras el otro esta leyendo todos; todas las interacciones del los cogs con esos bytes puede interrumpirse en el tiempo. En este caso el desarrollador debe diseñar cada proceso (en cada cog que comparte esta memoria) para que cooperativamente compartan el bloque de memoria de una forma no destructiva. Los seguros sirven como banderas para notificar a cada cog cuando es seguro manipular un bloque de memoria o cuando no.

Usando LOCKNEW

Un recurso definido por usuario mutuamente exclusivo debe iniciarse por un cog, entonces el mismo cog deberá usar **LOCKNEW** para verificar un seguro único en el cual administre ese recurso y pase el ID de ese seguro a cualquier otro cog que lo requiera. Por ejemplo:

```
VAR
  byte SemID

PUB SetupSharedResource
  <código para activar el recurso compartido definido por usuario>
  if (SemID := locknew) == -1
    <error, no hay seguros disponibles>
  else
    <comparte  el valor de SemID con otros cogs>
```

El ejemplo de arriba llama a **LOCKNEW** y almacena el resultado en SemID. Si ese resultado es -1 ocurrirá un error. Si SemID no es -1 entonces un seguro valido fue extraído y ese SemID necesita compartirse con otros cogs junto con la dirección del recurso para el que SemID es usado. El método usado para comunicar el SemID y la dirección del recurso depende de la aplicación , pero típicamente los dos se pasan como parámetros al método spin que se inicia en un cog o como el parámetro **PAR** cuando inicia una rutina ensamblador en un cog. Ver **COGNEW**, Pág. 81.

Reglas Sugeridas para Seguros

Los siguientes son unas reglas sugeridas para usar los seguros.

- Los objetos que necesitan un seguro para administrar un recurso mutuamente exclusivo definido por el usuario deberá activar un seguro usando **LOCKNEW** y guardar el ID de regreso, aqui lo llamaremos SemID. Solamente un cog deberá activar este seguro. El cog que activo el seguro debe comunicar SemID a todos los demás cogs que usaran el recurso.
- Cualquier cog que necesite el recurso debe activar primero con éxito el seguro SemID. Un exitoso “activado” es cuando LOCKSET(SemID) regresa **FALSE**. Si **LOCKSET** regreso **TRUE**, entonces otro cog debe estar accesando el recurso y deberá esperar e intentar después hasta obtener un exitoso “activo”.
- El cog que logro un "activo" con éxito puede manipular el recurso como sea necesario. Cuando esta hecho debe limpiar el seguro a través de LOCKCLR(SemID) así otro cog puede tener acceso al recurso. En un sistema bien hecho el resultado de

LOCKCLR puede ignorarse aqui ya que el cog es el único con el derecho lógico de limpiarlo.

- Si un recurso no se necesita mas o se convierte en no exclusivo, el seguro asociado deberá regresarse a la colección de seguros a través de LOCKRET(SemID). Usualmente esto se hace por el mismo cog que lo activo originalmente.

Las aplicaciones deberán escribirse de tal forma que los seguros no se accesen con LOCKSET o LOCKCLR a menos que ellos estén activados actualmente.

Note que los recursos definidos por el usuario no son actualmente asegurados por el hub o el seguro activado. La característica del seguro solo proporciona un significado para objetos que cooperativamente aseguran esos recursos. Depende de los objetos en si decidir y tolerar por las reglas de los seguros usar y que recursos serán gobernados por ellos. Adicionalmente el hub no asigna directamente un seguro al cog que llamo LOCKNEW, de alguna manera solo lo marca como que ha sido activado por un cog; cualquier otro cog puede regresar el seguro al grupo de seguros disponibles. también cualquier cog puede accesar cualquier seguro a través de los comandos LOCKCLR y LOCKSET aun si esos seguros nunca fueron activados. generalmente hacer tales cosas no es recomendado generalmente porque puede causar confusión con otros objetos bien hechos en la aplicación.

Ver LOCKRET, Pág. 129; LOCKCLR, Pág. 124; y LOCKSET, Pág. 130 para mas información.

LOCKRET

instrucción: Regresa el seguro al grupo de seguros, dejándolo disponible para futuras requisiciones LOCKNEW.

((PUB PRI))
LOCKRET (*ID*)

- *ID* es el ID (0 – 7) del seguro a regresar al grupo de seguros.

Explicación

LOCKRET es uno de las cuatro instrucciones (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usados para administrar recursos que se definen por usuario y que se consideran mutuamente exclusivos. LOCKRET regresa un seguro por *ID*, de vuelta al grupo de seguros del hub así puede reusarse por otros cogs en otro momento. Por ejemplo:

```
LOCKRET (2)
```

Este ejemplo regresa el seguro 2 de vuelta al hub. Esto no previene que el cog accese el seguro 2 posteriormente, solo permite al hub reasignarlo a los cogs que lo llamen en un futuro con LOCKNEW. Las aplicaciones deben escribirse de tal forma que los seguros no se accesen con LOCKSET o LOCKCLR a menos que estén activos actualmente.

Ver Acerca de los Seguros, Pág. 126, y Reglas Sugeridas para Seguros, Pág. 127 para información del uso típico de seguros y de instrucciones LOCKxxx.

Note que los recursos definidos por el usuario no son actualmente asegurados por el hub o el seguro activado. La característica del seguro solo proporciona un significado para objetos que cooperativamente aseguran esos recursos. Depende de los objetos en si decidir y tolerar por las reglas de los seguros usar y que recursos serán gobernados por ellos. Adicionalmente el hub no asigna directamente un seguro al cog que llamo LOCKNEW, de alguna manera solo lo marca como que ha sido activado por un cog; cualquier otro cog puede regresar el seguro al grupo de seguros disponibles. también cualquier cog puede accesar cualquier seguro a través de los comandos LOCKCLR y LOCKSET aun si esos seguros nunca fueron activados. generalmente hacer tales cosas no es recomendado generalmente porque puede causar confusión con otros objetos bien hechos en la aplicación.

Ver LOCKNEW, Pág. 126; LOCKCLR, Pág. 124; y LOCKSET, Pág. 130 para mas información.

LOCKSET

instrucción: Activa el seguro a verdadero y obtiene su estado previo.

((PUB PRI))
LOCKSET (*ID*)

Regresa: Estado previo del seguro (TRUE o FALSE).

- *ID* es el ID (0 – 7) del seguro para activar a TRUE.

Explicación

LOCKSET es uno de las cuatro instrucciones (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usados para administrar recursos que son definidos por usuario y considerados mutuamente exclusivos. LOCKSET activa el seguro *ID* a TRUE y regresa el estado previo de ese seguro (TRUE o FALSE).

Ver Acerca de los Seguros, Pág. 126, y Reglas Sugeridas para Seguros, Pág. 127 para información del típico uso de seguros las instrucciones LOCKxxx.

Lo siguiente asume que el cog (ya sea este u otro) ha activado un seguro utilizando LOCKNEW y ha compartido el *ID* con este cog, el cual se guarda como SemID. también asume que este cog tiene un arreglo de longs llamado LocalData.

```
PUB ReadResource | Idx
  repeat until not lockset(SemID) 'espera hasta asegurar el recurso
  repeat Idx from 0 to 9          'lee los 10 longs del recurso
    LocalData[Idx] := long[Idx]
  lockclr(SemID)                  'desbloquea el recurso

PUB WriteResource | Idx
  repeat until not lockset(SemID) 'espera hasta asegurar el recurso
  repeat Idx from 0 to 9          'escribe los 10 longs del recurso
    long[Idx] := LocalData[Idx]
  lockclr(SemID)                  'desbloquea el recurso
```

Estos dos métodos, ReadResource y WriteResource, siguen las mismas reglas antes y después de accesar el recurso. Primero esperan indefinidamente al primer ciclo REPEAT hasta que se asegura el recurso; por ejemplo si exitosamente activo el seguro asociado. Si LOCKSET regresa TRUE, la condición "until not lockset…" es falsa, lo que significa que algún otro cog esta actualmente accesando el recurso así que el primer ciclo REPEAT intenta nuevamente. Si

LOCKSET regresa FALSE, la condición “until not lockset…” es verdadera lo que significa que tenemos “locked the resource” y el primer ciclo REPEAT termina. El segundo ciclo REPEAT en cada método lee o escribe el recurso a través de las instrucciones long[Idx] y LocalData[Idx]. La ultima línea de cada método lockclr(SemID), limpia el seguro asociado del recurso a FALSE, desbloqueando lógicamente o liberando el recurso para que otros lo utilicen.

Ver LOCKNEW, Pág. 126; LOCKRET, Pág. 129; y LOCKCLR, Pág. 124 para mas información.

LONG

Denominador: Declara un símbolo de tamaño long, alineado long o lee/escribe un long en memoria principal.

```
VAR
  LONG Symbol {[Count]}
DAT
  {Symbol} LONG Data {[Count]}
((PUB  PRI))
  LONG [BaseAddress] {[Offset]}
```

- *Symbol* es el nombre de la variable (Sintaxis 1) o bloque de datos (Sintaxis 2).
- *Count* es una expresión opcional indicando el numero de elementos long para *Symbol* (Sintaxis 1) o el numero de entradas long de *Data* (Sintaxis 2) para almacenar en una tabla de datos.
- *Data* es una expresión constante o lista de coma separada de expresiones constantes.
- *BaseAddress* es una expresión que describe la dirección alineada long de memoria principal a leer o escribir. Si se omite *Offset*, *BaseAddress* es la dirección actual para operar. Si se especifica *Offset*, *BaseAddress* + *Offset* * 4 es la dirección actual para operar.
- *Offset* es una expresión opcional indicando el Offset de *BaseAddress* para operar. *Offset* esta en unidades de long.

Explicación

LONG es uno de los tres denominadores multipropósito (BYTE, WORD, y LONG) que declara u opera sobre memoria. LONG puede usarse para:

1) declara un símbolo long (32-bit) o un arreglo simbólico multi-long en un bloque VAR, o
2) declara un long alineado y/o dato tamaño long en un bloque DAT, o
3) lee o escribe un long de memoria principal a una dirección base con un Offset opcional.

Rango de Long

La memoria tamaño long (32 bits) puede contener un valor de 2^{32} combinaciones posibles de bits (ejemplo: uno de 4,294,967,296). El lenguaje spin desarrolla operaciones matemáticas usando 32-bit con signo, lo que significa que cada valor long se considera en el rango de -2,147,483,648 a +2,147,483,647. Sin embargo el valor numérico actual contenido en un long

esta sujeto a como la computadora y el usuario lo interpreten. En ensamblador Propeller un valor long puede ser tratado con o sin signo.

Declaración Variable Long (Sintaxis 1)

En bloques **VAR** la sintaxis 1de **LONG** se usa para declarar globales, variables simbólicas que son longs o arreglos longs, por ejemplo:

```
VAR
  long  Temp                  'Temp es un long (2 words, 4 bytes)
  long  List[25]              'List es un arreglo long
```

El ejemplo de arriba declara dos variables (símbolos), `Temp` y `List`. `Temp` es solo un valor sencillo variable long. La línea debajo de la declaración `Temp` usa campos opcionales *Count* para crear un arreglo de 25 elementos variables long llamado `List`. Ambos `Temp` y `List` pueden accesarse desde cualquier método **PUB** o **PRI** dentro del mismo objeto donde se declaro este bloque **VAR**; ambos son globales al objeto. Un ejemplo se muestra abajo.

```
PUB SomeMethod
  Temp := 25_000_000          'Activa Set a 25,000,000
  List[0] := 500_000          'Primer elemento de la lista a 500,000
  List[1] := 9_000            'Segundo elemento de la lista a 9,000
  List[24] := 60              'Ultimo elemento de la lista a 60
```

Para mas información del uso de **LONG** de esta forma vea la sección **VAR** Declaraciones Variables (Sintaxis 1) en Pág. 215, y tenga en cuenta que **LONG** se usa para el campo *Size* en esa descripción.

Declaración de Datos Long (Sintaxis 2)

En bloques **DAT** la sintaxis 2 de **LONG** se usa para declarar longs alineados y/o datos long que se compilan como valores constantes en memoria principal. Los bloques **DAT** permiten esta declaración para tener una símbolo opcional que lo preceda, el cual puede usarse para referencia posterior (ver **DAT**, Pág. 102). Por ejemplo:

```
DAT
  MyData  long  640_000, $BB50              'Dato tamaño Long-
alineado
  MyList  byte  long $FF995544, long 1_000  'Tamaño long / Byte
alineado
```

El ejemplo de arriba declara dos símbolos de datos `MyData` y `MyList`. `MyData` apunta al inicio de long alineado y tamaño long en memoria principal. Los valores de `MyData`, en memoria

principal son 640,000 y $0000BB50, respectivamente. MyList usa un bloque DAT especial de sintaxis LONG que cera una alineación byte pero activan tamaño long de datos en memoria principal. Los valores de MyList's en memoria principal son $FF995544 y 1,000, respectivamente. Cuando se accesan un byte a la vez MyList contiene $44, $55, $99, $FF, 232 y 3, 0 y 0 ay que los datos se almacenan en formato "little-endian"

Nota: MyList podría definirse como alineación word, tamaño de dato long si el "byte" referenciado se reemplazara por "word".

Este dato se compila en el objeto y la aplicación da un resultado como parte de la sección ejecutable del código y puede accesarse usando la forma lecto/escritura, sintaxis 3, de LONG (ver abajo). Para mayor información de como usar LONG de esta forma, vea DAT en la sección Declaración de Data(Sintaxis 1) en Pág. 103, y tenga en cuenta que LONG se usa para el campo en esa descripción.

Los datos pueden repetirse usando el campo opcional *Count*. Por ejemplo:

```
DAT
  MyData      long  640_000, $BB50[3]
```

El ejemplo anterior declara una alineación long con tabla de datos tamaño long, llamado MyData, que consiste en los siguientes cuatro valores: 640000, $BB50, $BB50, $BB50. Hay tres repeticiones de $BB50 debido al [3] en la declaración inmediata después de el.

Leyendo / Escribiendo Longs de Memoria Principal (Sintaxis 3)

En bloques PUB y PRI la sintaxis 3 de LONG se usa para leer o escribir valores tamaño long de memoria principal. Esto se hace escribiendo expresiones que se refieren a memoria principal usando la forma long [*BaseAddress*][*Offset*]. Aquí un ejemplo:

```
PUB MemTest | Temp
  Temp := LONG[@MyData][1]                 'Lee valor long
  long[@MyList][0] := Temp + $01234567     'escribe valor long

DAT
  MyData  long 640_000, $BB50              'tamaño long / alineado
long
  MyList  byte long $FF995544, long 1_000  'dato alineado long /
tamaño byte
```

En este ejemplo el bloque DAT (parte baja del código) coloca sus datos en memoria como se muestra en la Figura 2-2. El primer dato de MyData se coloca en la dirección $18. El ultimo

dato de `MyData` se coloca en la dirección $1C, con el primer elemento de `MyList` seguido inmediatamente en $20. Note que la dirección de inicio ($18) es arbitraria y seguramente cambiara según se modifique el código o si el objeto mismo se incluye en otra aplicación.

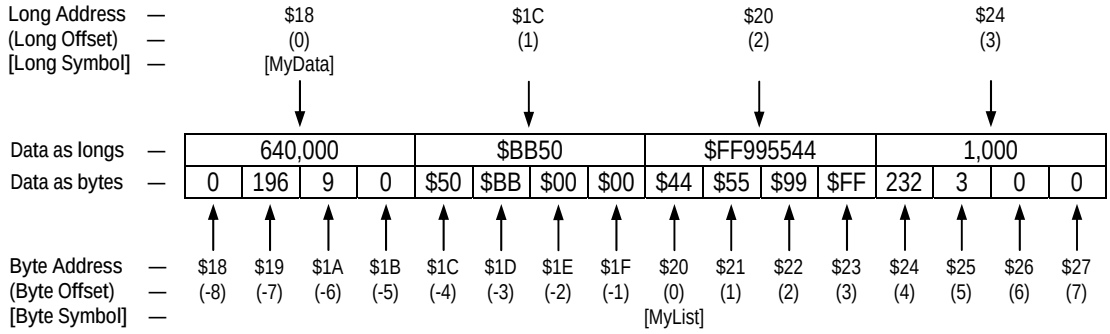


Figura 2-2: Estructura de datos y direccionamiento de datos tamaño long

Cerca de la parte superior del código la primer línea ejecutable del método `MemTest`, `Temp := long[@MyData][1]`, lee un valor tamaño long de memoria principal. Activa la variable local `Temp` a $BB50; el valor leído de la dirección de memoria principal $1C. La dirección $1C se determine por la dirección del símbolo `MyData` ($18) mas el Offset 1 (4 bytes). La siguiente simplificación progresiva demuestra esto.

```
long[@MyData][1] ➡ long[$18][1] ➡ long[$18 + (1*4)] ➡ long[$1C]
```

La siguiente línea long[@MyList][0] := Temp + $01234567, escribe un valor tamaño long a la memoria principal. Activa el valor en memoria principal en la dirección $20 a $012400BC. La dirección $20 se calculo de la dirección del símbolo MyList ($20) mas Offset 0 (0 bytes).

```
long[@MyList][0] ➡ long[$20][0] ➡ long[$20 + (0*4)] ➡ long[$20]
```

El valor $012400BC se deriva del valor actual de `Temp` mas $012400BC; $BB50 + $01234567 que es igual a $012400BC.

Direccionado Memoria Principal

Como la Figura 2-2 sugiere, la memoria principal es un grupo de bytes juntos (ver el renglón “datos como bytes”) que también pueden leerse como longs (4-byte sets) cuando se hace apropiadamente. De hecho el ejemplo muestra que las direcciones se calculan en términos de bytes. Este concepto es un tema consistente para cualquier instrucción que usa direcciones.

La memoria principal es finalmente direccionada en términos de bytes sin importar el tamaño del valor que esta accesando; byte, word, o long. Esto tiene ventajas si se piensa como los bytes, words, y longs se relacionan entre si. Pero puede ser problemático cuando se piensa en múltiples piezas de un solo tamaño, como longs.

Por esta razón el denominador **LONG** es una característica útil para facilitar el direccionamiento desde una perspectiva céntrica long. Su campo *BaseAddress* cuando se combina con el campo opcional *Offset* opera en modo conciente de la base.

Imagine accesar longs de memoria desde un punto de inicio (*BaseAddress*). Quizá piense naturalmente al siguiente long como una cierta distancia desde ese punto (*Offset*). Mientras esos longs son realmente un cierto numero de "bytes" mas allá de un punto dado, es mas sencillo pensar en ellos como un numero de "longs" mas allá de un punto (por ejemplo, el 4th long, en vez de el long que empieza después del 12th byte). El denominador **LONG** e trata apropiadamente tomando el valor de *Offset* (unidades de longs), multiplicado por 4 (numero de bytes por cada long), y suma ese resultado a *BaseAddress* para determinar el long correcto de memoria a leer. también limpia los dos bits mas bajos de *BaseAddress* para asegurar que la dirección referenciada es un alineado long.

así, cuando se leen valores de la lista `MyData`, `long[@MyData][0]` lee el valor del primer long y `long[@MyData][1]` lee el segundo.

Si el campo *Offset* no se uso las instrucciones de arriba deberán ser algo así como `long[@MyData]`, y `long[@MyData+4]`, respectivamente. El resultado es el mismo pero la forma de escribirlo puede no ser tan clara.

Para mas explicación de como se acomodan los datos en memoria vea la sección **DAT** en Declaración de Data(Sintaxis 1) en Pág. 103.

Una alternativa de referencia de Memoria

Aun hay otra forma de accesar los datos desde el código del ejemplo de arriba; se podrá referenciar los símbolos de datos directamente. por ejemplo estas instrucciones leerán los primeros dos longs de la lista `MyData`:

```
Temp := MyData[0]
Temp := MyData[1]
```

así que ¿por que no usar directamente los símbolos de referencia todo el tiempo? Considere el caso siguiente:

```
Temp := MyList[0]
Temp := MyList[1]
```

Haciendo referencia nuevamente al ejemplo del código de la Figura 2-2 puede esperar estas dos instrucciones para leer el primero y segundo long de MyList; $FF995544 y 1000, respectivamente. En vez de leer el primer y segundo "byte" de MyList, $44 y $55, respectivamente.

¿Que sucedió? a diferencia de MyData, la entrada MyList se definen el código como tamaño byte y dato alineado byte. Los datos de hecho consisten en valores long porque cada elemento es precedido de **LONG**, pero como el símbolo para la lista se declara como tamaño byte todas las referencias directas regresaran bytes individuales.

Sin embargo puede usarse el denominador **LONG**, como la lista también puede pasar long alineados porque su posición es seguida de MyData.

```
Temp := long[@MyList][0]
Temp := long[@MyList][1]
```

Lo de arriba lee el primer long, $FF995544, seguido del segundo long, 1000, de MyList. Esta característica es muy útil ya que una lista de datos necesita accesarse como bytes o longs en diferentes momentos durante la aplicación.

Otro fenómeno de Direccionamiento

Ambas técnicas, el **LONG** y la referencia de símbolo directo demostradas arriba pueden usarse para accesar cualquier localidad en memoria principal sin importar como se relaciona al dato definido. Aquí unos ejemplos:

```
Temp := long[@MyList][-1]  'lee ultimo long de MyData(antes MyList)
Temp := long[@MyData][2]   'lee primer long de MyList(después MyData)
Temp := MyList[-8]         'lee primer byte de MyData
Temp := MyData[-2]         'lee long que es dos longs antes de MyData
```

Estos ejemplos leen mas allá de las fronteras lógicas (punto de inicio o punto final) de las listas de datos a los que hacen referencia. Esto puede ser un truco útil pero mayormente es hecho por error; tenga cuidado cuando direcciona memoria, especialmente si esta escribiendo a esa memoria.

LONGFILL

instrucción: Llena longs de memoria principal con un valor.

((PUB PRI))
LONGFILL(*StartAddress*, *Value*, *Count*)

- *StartAddress* es una expresión indicando la localidad del primer long de memoria a llenar con un *Value*.
- *Value* es una expresión indicando el valor con el cual llenara los longs.
- *Count* es una expresión indicando el numero de longs a llenar empezando con *StartAddress*.

Explicación

LONGFILL es uno de tres comandos (BYTEFILL, WORDFILL, y LONGFILL) usados para llenar bloques de memoria con un valor especifico. LONGFILL llena los longs *Count* de memoria principal con *Value*, empezando en la dirección *StartAddress*.

Usando LONGFILL

LONGFILL es una excelente forma de limpiar bloques de memoria tamaño long. Po ejemplo:

```
VAR
  long   Buff[100]

PUB Main
  longfill(@Buff, 0, 100)     'Limpia Buff a 0
```

La primera línea del método Main de arriba limpia completamente el arreglo Buff de los 100-long (400-byte) todos a cero. LONGFILL es mas rápido en esta tarea que un ciclo REPEAT.

LONGMOVE

instrucción: Copia longs de una región a otra en memoria principal.

((PUB PRI))
LONGMOVE (*DestAddress*, *SrcAddress*, *Count*)

- *DestAddress* es una expresión que especifica la localidad de memoria principal para copiar el primer long a la localidad destino.
- *SrcAddress* es una expresión que especifica la localidad de memoria principal del primer long de la fuente a copiar.
- *Count* esta expresión indica el numero de longs a copiar de la fuente al destino.

Explicación

LONGMOVE es una de tres instrucciones (BYTEMOVE, WORDMOVE, y LONGMOVE) usados para copiar bloques de memoria principal de un área a otra. LONGMOVE copia los longs *Count* de memoria principal de *SrcAddress* a memoria principal en *DestAddress*.

Usando LONGMOVE

LONGMOVE es una excelente forma de copiar bloques de memoria tamaño long. Por ejemplo:

```
VAR
  long   Buff1[100]
  long   Buff2[100]

PUB Main
  longmove(@Buff2, @Buff1, 100)     'Copia Buff1 aBuff2
```

La primer línea de l método `Main` de arriba copia los 100-long del arreglo `Buff1` al arreglo `Buff2`. LONGMOVE es mas rápido en esta tarea de un ciclo dedicado REPEAT.

LOOKDOWN, LOOKDOWNZ

instrucción: Obtiene el índice de un valor en una lista.

((PUB PRI))
LOOKDOWN (*Value* : *ExpressionList*)

((PUB PRI))
LOOKDOWNZ (*Value* : *ExpressionList*)

Regresa: Posición de índice base uno (LOOKDOWN) o una posición de base cero (LOOKDOWNZ) de *Value* en *ExpressionList*, o 0 si *Value* no se encuentra.

- *Value* esta expresión indica el valor a encontrar en *ExpressionList*.
- *ExpressionList* una lista de expresiones de coma separada. Cadenas de caracteres entre comillas se permiten; estas se tratan como lista de caracteres de coma separada

Explicación

LOOKDOWN y LOOKDOWNZ son comandos que recuperan índices de valores de una lista de valores. LOOKDOWN regresa la posición de índice base uno (1..N) de *Value* de *ExpressionList*. LOOKDOWNZ es igual que LOOKDOWN solo que regresa la posición de base cero (0..N−1). Para ambas instrucciones si *Value* no se encuentra en *ExpressionList* entonces se regresa 0.

Usando LOOKDOWN o LOOKDOWNZ

LOOKDOWN y LOOKDOWNZ son útiles para mapeo de un grupo de números no contiguos (25, -103, 18, etc.) a un grupo de números contiguos (1, 2, 3, etc. –o– 0, 1, 2, etc.) donde no hay expresiones algebraicas que pueden hacerse simples. Los siguientes ejemplos asumen que Print es un método creado en otro lado:

```
PUB ShowList | Index
  Print(GetIndex(25))
  Print(GetIndex(300))
  Print(GetIndex(2510))
  Print(GetIndex(163))
  Print(GetIndex(17))
  Print(GetIndex(8000))
  Print(GetIndex(3))

PUB GetIndex(Value): Index
  Index := lookdown(Value: 25, 300, 2_510, 163, 17, 8_000, 3)
```

El método `GetIndex` es un ejemplo que usa **LOOKDOWN** para encontrar `Value` y regresa el índice donde se encontró en *ExpressionList*, o 0 si no se encontró. El método `ShowList` llama `GetIndex` repetidamente con diferentes valores e imprime el resultado del índice en un display. Asumiendo que `Print` es un método que despliega un valor, este ejemplo imprimirá 1, 2, 3, 4, 5, 6 y 7 en un display.

Si se usa **LOOKDOWNZ** en vez de **LOOKDOWN** este ejemplo imprimiría 0, 1, 2, 3, 4, 5, y 6 en un display.

Si *Value* no se encuentra **LOOKDOWN**, o **LOOKDOWNZ**, regresa 0. así si una d las líneas del método `ShowListfue`, `Print(GetIndex(50))`, el display mostrar 0 en el momento que se ejecuta.

Si se usa **LOOKDOWNZ**, tenga en cuenta que puede regresar 0 si no se encontró *Value* o si el índice es 0 en *Value*. Asegúrese que esto no ocasionara errores en el código, de lo contrario use **LOOKDOWN**.

LOOKUP, LOOKUPZ

Instrucción: Obtiene un valor de una posición índice dentro de una lista.

((PUB PRI))
LOOKUP (*Index* : *ExpressionList*)

((PUB PRI))
LOOKUPZ (*Index* : *ExpressionList*)

Regresa: El valor de la posición base uno de *Index* (LOOKUP) o posición base cero de *Index* (LOOKUPZ) de *ExpressionList*, o 0 si esta fuera de rango.

- *Index* la expresión indica la posición del valor deseado en *ExpressionList*. Para LOOKUP, *Index* es base uno (1..N). Para LOOKUPZ, *Index* es base cero (0..N-1).
- *ExpressionList* es una lista de expresiones de coma separada. Caracteres entre comillas se aceptan, estos se tratan como lista de caracteres de coma separada.

Explicación

LOOKUP y LOOKUPZ son instrucciones que recuperan accesos de una lista de valores. LOOKUP regresa el valor de *ExpressionList* que esta localizado en la posición base uno (1..N) dada por *Index*. LOOKUPZ es igual que LOOKUP excepto que osa base cero en *Index* (0..N-1). Para ambas instrucciones si *Index* esta fuera de rango se regresa un 0.

Usando LOOKUP o LOOKUPZ

LOOKUP y LOOKUPZ son útiles para hacer mapeos de grupos de números contiguos (1, 2, 3, etc. –o– 0, 1, 2, etc.) a un grupo de números no contiguos (45, -103, 18, etc.) donde no se puede encontrar expresión algebraica para hacerlo simple. Los siguientes ejemplos asumen que Print es un método creado en otro lugar.

```
PUB ShowList | Index, Temp
  repeat Index from 1 to 7
    Temp := lookup(Index: 25, 300, 2_510, 163, 17, 8_000, 3)
    Print(Temp)
```

Este ejemplo revisa todos los valores en *ExpressionList* de LOOKUP y los muestra. El ciclo REPEAT cuenta con Index de 1 a 7. Cada iteración del ciclo LOOKUP usa Index par recuperar valores de su lista. Si Index es igual a 1, el valor 25 se regresa. Si Index es igual a 2 se regresa el valor 300. Asumiendo que Print es un método que despliega el valor de Temp, este ejemplo imprimirá 25, 300, 2510, 163, 17, 8000 y 3 en el display.

Si LOOKUPZ se usa entonces la lista es base cero en vez de base uno; un Index 0 regresa 25, Index de 1 regresa 300, etc.

Si *Index* esta fuera de rango se regresa 0. así para LOOKUP, si la instrucción REPEAT fuera de 0 a 8, en vez de 1 a 7, este ejemplo imprimiría 0, 25, 300, 2510, 163, 17, 8000, 3 y 0 en el display.

NEXT

Instrucción: Ignora las instrucciones restantes de un ciclo REPEAT y continua con el siguiente ciclo.

((PUB PRI))
  NEXT

Explicación

NEXTes una de dos instrucciones (NEXT y QUIT) que afectan el ciclo REPEAT. NEXT hace que cualquier instrucción en el ciclo REPEAT se brinque a la siguiente iteracion del ciclo para ser iniciado despues.

Usando NEXT

NEXT se usa típicamente como un caso de excepción, en una instrucción condicional, en un ciclo REPEAT para moverse inmediatamente a la siguiente iteración del ciclo. Por ejemplo asume que X es una variable creada anteriormente y Print() es un método creado en otro lugar que imprime un valor en pantalla.

```
repeat X from 0 to 9        'Repite 10 veces
  if X == 4
    next                    'brinca si X = 4
  byte[$7000][X] := 0       'limpia localidades RAM
  Print(X)                  'imprime X  en pantalla
```

El código de arriba limpia las localidades RAM e imprime el valor de X en pantalla, pero con una excepción. Si X es igual a 4, la instrucción IF ejecuta el comando NEXT lo cual hace que el ciclo se brinque las líneas restantes y se vaya directamente a la siguiente iteracion. Esto tiene el efecto de limpiar localidades RAM $7000 a $7003 y localidades $7005 a $7009 e imprimir 0, 1, 2, 3, 5, 6, 7, 8, 9 en la pantalla.

La instrucción NEXT solo puede usarse en un ciclo REPEAT; de otra forma ocurrirá un error.

OBJ

Denominador: Declara un Bloque Objeto

OBJ
Symbol ⟨[*Count*]⟩: "*ObjectName*" ⟨ ↳ *Symbol* ⟨[*Count*]⟩: "*ObjectName*"⟩...

- *Symbol* es el nombre deseado para el símbolo objeto.
- *Count* es una expresión opcional encerrada en corchetes que indica si es un arreglo de objetos con *Count* numero de elementos. Cuando se referencia a esos elementos posteriormente se comienza con el elemento 0 y se termina con el elemento *Count*-1.
- *ObjectName* es el nombre del archivo sin extensión del objeto deseado. Dentro de la compilación se busca un objeto con este nombre en los tabuladores de edición, directorio de trabajo y directorio de librerías. El nombre del objeto puede contener cualquier carácter valido de nombre de archivo sin incluir \, /, :, *, ?, ", <, >, y |.

Explicación

El Bloque Objeto es una sección de código que declara cuales objetos se usan y que símbolos de objetos representan. Esta es una de las seis declaraciones especiales (**CON**, **VAR**, **OBJ**, **PUB**, **PRI**, y **DAT**) que proporcionan una estructura inherente al lenguaje Spin.

Las declaraciones de objeto comienzan con **OBJ** en una línea seguida por uno o mas declaraciones. **OBJ** comienza en la columna 1 (a la izquierda) de la línea donde esta, se recomienda que las siguientes líneas estén indentadas al menos por un espacio. Por ejemplo:

```
OBJ
  Num  : "Numbers"
  Term : "TV_Terminal"
```

Este ejemplo define `Num` como un símbolo de objeto de tipo `"Numbers"` y `Term` como un símbolo de objeto de tipo `"TV_Terminal"`. Los métodos públicos y privados se pueden referir a estos objetos usando los símbolos de objetos como en el siguiente ejemplo.

```
PUB Print | S
  S := Num.ToStr(LongVal, Num#DEC)
  Term.Str(@S)
```

Este método publico, `Print`, llama al método `ToStr` de `Numbers` y también al método `Str` de `TV_Terminal`. Usa los símbolos de objeto `Num` y `Term` seguidos del símbolo de referencia

Objeto-Metodo (un punto '.') y finalmente el nombre del método a llamar. Num.ToStr, por ejemplo, llama al método ToStr del objeto publico de Numbers. Term.Str llama al método Str del método publico de TV_Terminal. En este caso el Num.ToStr tiene dos parámetros, en paréntesis y Term.Str tiene un parámetro.

también note que el Segundo parámetro de Num.ToStr es Num#DEC. El símbolo **#** es el símbolo de referencia Objecto-Constante; le da acceso a una constante de objeto. En este caso, Num#DEC se refiere a la constante **DEC** (formato decimal) en el objeto Numbers.

Vea la referencia Objeto-método '.' Y referencia objeto-constante '**#**'> en la Tabla 2-16: en Pág. 211 para mas información.

Múltiples instancias de un objeto pueden declararse con el mismo símbolo de objeto usando una sintaxis de arreglos y pueden accesarse similar a los arreglos. Por ejemplo:

```
OBJ
  PWM[2] : "PWM"
PUB GenPWM
  PWM[0].Start
  PWM[1].Start
```

Este ejemplo declara PWM como un arreglo de dos objetos (dos instancias en el mismo objeto). El objeto mismo llama a "PWM". El método publico, GenPWM, llama al método Start de cada instancia usando indices 0 y 1 con el arreglo del símbolo de objeto PWM.

Ambas instancias del objeto PWM se compilan en la aplicación de modo que hay una copia de su código de programa (**PUB**s, **PRI**s, y **DAT**s) y dos copias de sus cloques variables (**VAR**s). Esto es porque para cada instancia el código es el mismo pero cada instancia necesita su propio espacio variable para poder operar independientemente del otro.

Un punto importante a considerar con múltiples instancias de un objeto es que solo hay una copia de su bloque **DAT** porque puede contener código ensamblador Propeller. Los bloques **DAT** también pueden contener datos inicializados y regiones reservadas para propósitos de espacio de trabajo, todos con nombres simbólicos. Como hay una sola copia para múltiples instancias de un objeto el área es compartida con todas las instancias. Esto proporciona una forma conveniente de crear memoria compartida entre múltiples instancias par un objeto.

Alcance de los Símbolos de Objetos

Los símbolos de Objetos definidos en bloques Objeto son globales al objeto en el cual son definidos pero no están disponibles fuera de ese objeto. Esto significa que estos símbolos de objetos pueden accesarse directamente de cualquier parte dentro del objeto y sus nombres no entraran en conflicto con símbolos definidos en objetos padres o hijos.

Operadores

El chip Propeller contiene un poderoso grupo de operadores lógicos y matemáticos. Un subgrupo de estos operadores es soportado por el lenguaje ensamblador Propeller, sin embargo, como el lenguaje Spin tiene un uso para cada forma de operador soportado por el Propeller, esta sección describe cada operador en detalle. Vea la sección Operadores en la Pág. 335 para una lista de operadores disponibles en ensamblador Propeller.

Apariencia del espacio de trabajo

El Propeller es un aparato de 32-bit y a menos que otra cosa se especifique las expresiones siempre se evalúan en 32-bit, enteros con signo matemático. Esto incluye resultados intermedios. Si cualquier resultado intermedio sobrepasa los 32-bit enteros de signo (arriba de 2,147,483,647 o debajo de -2,147,483,648), el resultado final de la expresión no será como se espera. Un espacio de trabajo de 32 bits proporciona mucho espacio para resultados intermedios pero es de sabios mantener las posibilidades de sobre flujo en mente.

Si matemáticas incompletas es un problema o si una expresión requiere números reales en vez de enteros, el soporte de punto flotante puede ayudar. El compilador soporta valores y expresiones constantes de 32 bits de punto flotante con muchos de los mismos operadores matemáticos. Note que esto es para expresiones constante solamente, no para expresiones variables en tiempo de ejecución. Para expresiones en tiempo real de punto flotante el chip Propeller da soporte a través del objeto FloatMath que se incluye en el software de instalación. Ver Asignación Constante '=', Pág. 152; FLOAT, Pág. 111; ROUND, Pág. 202; y TRUNC, Pág. 213, así como los objetos FloatMath y FloatString para mas información.

Atributos de Operador

Los operadores tienen los siguientes atributos importantes, cada uno de los cuales se muestra en las siguientes dos tablas y se explica mas adelante:

- Unario / Binario
- Normal / Asignación
- expresión Constante y/o Variable
- Nivel de Precedencia

Tabla 2-9: Operadores lógicos y Matemáticos					
Operador	Uso Asignado	expresión Constante[1]		Es Unario	Descripción, Numero de Pagina
		Entero	Flotante		
=	Siempre	n/a[1]	n/a[1]		Asignación de Constante (Bloques CON solo), 152
:=	Siempre	n/a[1]	n/a[1]		Asignación de Variable (Bloques PUB/PRI only), 153
+	+=	✓	✓		Suma, 153
+	Nunca	✓	✓	✓	Positivo (+X); forma unaria de suma, 154
-	-=	✓	✓		Resta, 154
-	Si solo	✓	✓	✓	Negado (-X); forma unaria de resta, 154
--	Siempre			✓	Pre-decremento (--X) o post-decremento (X--), 155
++	Siempre			✓	Pre-incremento (++X) o post-incremento (X++), 156
*	*=	✓	✓		Multiplica y regresa los 32 bits bajos (signo), 157
**	**=	✓			Multiplica y regresa los 32 bits altos (signo), 157
/	/=	✓	✓		Divide (signo), 158
//	//=	✓			Modulus (signo), 158
#>	#>=	✓	✓		Limite mínimo (signo), 159
<#	<#=	✓	✓		Limite máximo (signo), 159
^^	Si solo	✓	✓	✓	Raíz cuadrada, 160
||	Si solo	✓	✓	✓	Valor absoluto, 160
~	Siempre			✓	Signo-extendido de bit 7 (~X) o post-clear a 0 (X~); Bits bajos, 160
~~	Siempre			✓	Signo-extendido de bit 15 (~~X) o post-set a -1 (X~~); bits altos, 161
~>	~>=	✓			Corrimiento aritmético a la derecha, 162
?	Siempre			✓	Numero aleatorio adelante (?X) o reversa (X?), 163
|<	Si solo	✓		✓	Bitwise: Decodifica Valor (0 - 31) en un bit alto simple long, 164
>|	Si solo	✓		✓	Bitwise: Codifica long en un valor (0 - 32) como bit alto prioridad, 164
<<	<<=	✓			Bitwise: Mueve izquierda, 165
>>	>>=	✓			Bitwise: Mueve derecha, 165
<-	<-=	✓			Bitwise: Rota izquierda, 166
->	->=	✓			Bitwise: Rota derecha, 167
><	><=	✓			Bitwise: Reversa, 167
&	&=	✓			Bitwise: AND, 168
|	|=	✓			Bitwise: OR, 169
^	^=	✓			Bitwise: XOR, 169
!	Si solo	✓		✓	Bitwise: NOT, 170
AND	AND=	✓	✓		Boolean: AND (promueve no-0 a -1), 171
OR	OR=	✓	✓		Boolean: OR (promueve no-0 a -1), 172
NOT	Si solo	✓	✓	✓	Boolean: NOT (promueve no-0 a -1), 172
==	===	✓	✓		Boolean: Es igual, 173
<>	<>=	✓	✓		Boolean: No es igual, 174
<	<=	✓	✓		Boolean: Menor que (signo), 175
>	>=	✓	✓		Boolean: Mayor que (signo), 175
=<	=<=	✓	✓		Boolean: Igual o menor que (signo), 175
=>	=>=	✓	✓		Boolean: Igual o Mayor que (signo), 176
@	Nunca	✓		✓	símbolo de dirección, 177
@@	Nunca			✓	Dirección objeto mas símbolo, 177

[1] Formas de asignación de operadores no están permitidas en expresiones constantes.

Tabla 2-10: Niveles de Precedencia de Operadores			
Nivel	Notas	Operadores	Nombres de Operadores
Mas alto (0)	Unario	--, ++, ~, ~~, ?, @, @@	Inc/Decrementa, Limpia, active, Aleatorio, Dirección símbolo/Objeto
1	Unario	+, -, ^^, ||, |<, >|, !	Positivo, Negado, Raíz, Absoluto, Decodifica, Codifica, Bitwise NOT
2		->, <-, >>, <<, ~>, ><	Rota Der/Izq, Mueve Der/ Izq, Corrimiento aritmético Der, Reversa
3		&	Bitwise AND
4		|, ^	Bitwise OR, Bitwise XOR
5		*, **, /, //	Multiplica-bajo, Multiplica-Alto, Divide, Modulo
6		+, -	Suma, Resta
7		#>, <#	Limite Máximo/mínimo
8		<, >, <>, ==, =<, =>	Boolean: Menor/ Mayor que, No igual/Igual, Igual o Menor/Mayor
9	Unario	NOT	Boolean NOT
10		AND	Boolean AND
11		OR	Boolean OR
Mas Bajo (12)		=, :=, Otras asignaciones	Asignación Constante/ Variable, Asignación formas de operadores binarios

Unario / Binario

Cada operador es unario o binario por naturaleza.. Operadores unarios son aquellos que operan con solo un operando. Por ejemplo:

```
!Flag     ' bitwise NOT de bandera
^^Total     ' raíz cuadrada de total
```

Operadores binarios son aquellos que operan con dos operandos. Por ejemplo:

```
X + Y     ' suma X y Y
Num << 4  ' mueve Num a la izquierda 4 bits
```

Note que el termino “operador binario” significa “dos operandos” no tiene nada que ver con dígitos binarios. Para distinguir operadores cuya función se relaciona a dígitos binarios usaremos el termino “bitwise”.

Normal / Asignación

Los operadores normales como suma ‘**+**’ y corrimiento a la izquierda ‘**<<**’, operan en sus operandos y proporcionan el resultado para el uso del resto de la expresión sin afectar el operando u operandos. Aquellos que son operadores de asignación escriben sus resultados ya sea en la variable en la que operan (unario), o en la variable te su izquierda inmediata (binario), además de proporcionar el resultado para uso del resto de la expresión.

Aquí un ejemplo de operadores de asignación:

```
Count++          ' (Unario) evalua Count + 1
                 ' y escribe el resultado en Count
Data >>= 3   ' (Binario) mueve Data a la derecha 3 bits
                 ' y escribe el resultado en Data
```

Los operadores binarios tienen formas especiales terminan en igual '=' para hacerlos operadores de asignación. Los operadores unarios no tienen forma especial de asignación; algunos siempre asignan mientras otros asignan solo en situaciones especiales. Ver Tabla 2-9 y a explicación del operador para mayor información.

La mayoría de los operadores de asignación solo pueden usarse dentro de los métodos (bloques `PUB` y `PRI`). La única excepción es el operador de asignacion constante '=' el cual solo puede usarse en bloques `CON`.

Expresiones Constantes y/o Variables

Los operadores que tienen atributos de una expresión entera constante pueden usarse en tiempo de ejecución para expresiones variables y en tiempo de compilación en expresiones constantes. Los operadores con expresiones constantes float se usan en expresiones constantes durante la compilación. Los operadores sin atributos de expresiones solo se usan en tiempo de ejecución en expresiones variables. La mayoría de los operadores tienen formas de no asignación que permiten usarlos en ambas expresiones, contantes y variables.

Nivel de Precedencia

Cada operador tiene un nivel de precedencia que determina cuando tomara acción en relación a otros operadores en la misma expresión. Por ejemplo se conocen algebraicamente algunas reglas que requieren multiplicar y dividir antes de sumar y restar. Los operadores de multiplicar y dividir tienen un "nivel mas alto de precedencia" que la suma y resta. también multiplicar y dividir son conmutables; tienen el mismo nivel de precedencia, así sus operaciones tienen el mismo valor sin importar el orden en el que se realizan (multiplicar primero y luego dividir o viceversa). Los operadores conmutativos siempre son evaluados de izquierda a derecha excepto cuando un paréntesis sobrescribe la regla.

El chip Propeller aplica el orden de las reglas de operaciones como en Algebra: las expresiones se evalúan de izquierda a derecha, excepto cuando existen paréntesis y diferentes niveles de precedencia.

Siguiendo esta reglas el Propeller evaluara:

X = 20 + 8 * 4 – 6 / 2

...es igual a 49; que es, 8 * 4 = 32, 6 / 2 = 3, y 20 + 32 – 3 = 49. Si desea que la expresión se evalúe diferente use paréntesis para encerrar las porciones necesarias de la expresión.

Por ejemplo:

X = (20 + 8) * 4 – 6 / 2

Esto evaluara la expresión en paréntesis primero, el 20 + 8, hacienda que la expresión ahora sea 109, en vez de 49.

La Tabla 2-10 indica el nivel de precedencia de cada operador desde el mas alto (nivel 0) al mas majo (nivel 12). Los operadores con mas alto nivel de realizan entes que los de mas bajo nivel, multiplicar antes de sumar, absoluto antes e multiplicar, etc. La única excepción es cuando se incluye el paréntesis; este sobrescribe cada nivel de precedencia.

Asignaciones Intermedias

El motor de expresiones del chip Propeller permite y procesa operadores de asignación de etapas intermedias. Esto se llama "asignación intermedia" y puede usarse para desarrollar cálculos complejos en menos código. Por ejemplo la siguiente ecuación se soporta fuertemente en X, y X + 1.

```
X := X - 3 * (X + 1) / ||(X + 1)
```

La misma instrucción puede escribirse tomando ventaja del asignador intermedio propiedad del operador de incremento:

```
X := X++ - 3 * X / ||X
```

Asumiendo que `X` empieza en -5, ambas instrucciones evalúan a -2, y ambas almacenan ese valor en `X` cuando se completan. La segunda instrucción sin embargo lo hace soportándose en la asignación intermedia (la parte `X++`) para simplificar el resto de la instrucción. El operador incremento '**++**' se evalúa primero (mayor precedencia) e incrementa X de -5 to -4. Como este es un "post incremento" (ver Incremento, pre- o post- '++', Pág. 156) primero regresa el valor original de `X`, -5, la expresión y luego escribe un nuevo valor, -4, a `X`. así el "`X++ - 3`…" parte de la expresión se convierte en "-5 – 3…" así los operadores absoluto, multiplicación y división son evaluados, pero el valor de `X` ha cambiado, así ellos usan el nuevo valor, -4, para sus operaciones:

-5 – 3 * -4 / ||-4 → -5 – 3 * -4 / 4 → -5 – 3 * -1 → -5 – -3 = -2

Ocasionalmente el uso de asignadores intermedios pueden comprimir múltiples líneas en una expresión sencilla, resultando en código ligeramente mas pequeño y de ligeramente ejecución mas rápida.

Las paginas restantes de esta sección explican con mas detalle cada operador lógico y matemático mostrado en la Tabla 2-9 en el mismo orden que se muestra.

Asignación Constante '='

El operador de asignación constante se usa solamente en bloques **CON**, para declarar constantes en tiempo de compilación. Por ejemplo:

```
CON
  _xinfreq = 4096000
  WakeUp   = %00110000
```

Este código active el símbolo `_xinfreq` a 4,096,000 y el símbolo `WakeUp` a %00110000. A través del resto del programa el compilador usara estos números en vez de sus respectivos símbolos. Ver **CON**, Pág. 87.

Estas declaraciones son expresiones constantes, así que muchos de los operadores normales pueden usarse para calcular un valor de constante final al momento de compilar. Por ejemplo, puede ser mas claro reescribir el ejemplo anterior como sigue:

```
CON
  _xinfreq   = 4096000
  Reset      = %00100000
  Initialize = %00010000
  WakeUp     = Reset & Initialize
```

Aquí, `WakeUp` esta todavía activado a %00110000 en tiempo de compilación, pero ahora es mas claro para futuros lectores que el símbolo `WakeUp` contiene los códigos binarios para `Reset` y `Initialize` en secuencia para esa aplicación en particular.

Los ejemplos de arriba están creados en constantes enteres con signo de 32-bit; sin embargo también es posible crear constantes de punto flotante de 32-bit. Para hacer eso la expresión debe estar escrita como punto flotante en una de tres formas: 1) como valor entero seguido de un punto decimal de al menos un digito, 2) como un entero seguido de una E y un valor exponencial, o 3) ambos: 1 y 2.

Por ejemplo:

```
CON
  OneHalf = 0.5
  Ratio   = 2.0 / 5.0
  Miles   = 10e5
```

El código de arriba crea tres constantes de punto flotante. OneHalf es igual a 0.5, Ratio es igual a 0.4 y Miles es igual a 1,000,000. Note que si Ratio fuera definido como 2 / 5 en vez de 2.0 / 5.0, la expresión se trataría como entero constante y el resultado seria un entero constante igual a 0. Para las expresiones constantes de punto flotante cada valor dentro de la expresión debe ser valor de punto flotante; no se pueden mezclar enteros con punto flotante como Ratio = 2 / 5.0. Se puede sin embargo usar la declaración FLOAT para convertir un entero a valor de punto flotante tal como Ratio = FLOAT(2) / 5.0.

El compilador Propeller maneja constantes de punto flotante como números reales de precisión simple según describe el estándar IEEE-754. números reales de precisión simple se almacenan en 32 bits con un bit de signo un exponente de 8 bits y 23 bits de mantisa (la parte fraccional). Esto proporciona aproximadamente un largo de 7.2 dígitos decimales.

Para operaciones de punto flotante en tiempo de ejecución los objetos FloatMath y FloatString proporcionan funciones compatibles con números de precisión simple.

Ver FLOAT, Pág. 111; ROUND, Pág. 202; TRUNC, Pág. 213, si como los objetos FloatMath y FloatString para mayor información.

Asignación Variable ':='

El operador de asignación variable se usa solo dentro de métodos (bloques PUB y PRI), para asignar un valor a una variable. Por ejemplo,

```
Temp := 21
Triple := Temp * 3
```

En tiempo de ejecución este código activaría la variable Temp igual a 21 y activaría Triple a 21 * 3, lo cual es 63.

Con otros operadores de asignación el operador de la variable de asignación puede usarse con expresiones para asignar resultados intermedios tales como:

```
Triple := 1 + (Temp := 21) * 3
```

Este ejemplo primero activa Temp a 21, luego multiplica Temp por 3 y suma 1, finalmente asigna el resultado 64 a Triple.

Add '+', '+='

El operador Add suma dos valores juntos. Add puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y + 5
```

Add tiene una forma de asignación, **+=**, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino.

Por ejemplo:

```
X += 10  'forma corta de X := X + 10
```

Aquí el valor de X se suma a 10 y el resultado se almacena en X. La forma de asignación Add puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Positivo '+' (forma unaria de suma)

Positivo es la forma unaria de Add y puede usarse similar a negar excepto que nunca es un operador de asignación. Positivo es ignorado esencialmente por el compilador pero es útil cuando el signo de los operandos es importante ara hacer énfasis. Por ejemplo:

```
Val := +2 - A
```

Restar '-', '-='

El operador Subtract resta dos valores. Subtract puede usarse en ambas expresiones variables y constantes. Ejemplo:

```
X := Y - 5
```

Subtract tiene una forma de asignación, **-=**, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo:

```
X -= 10  'Forma corta de X := X - 10
```

Aquí se resta 10 del valor de X y el resultado se almacena en X. La forma de asignación de Subtract puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Negar '-' (forma unaria de restar)

Negar es la forma unaria de restar. Al negar el signo del pin a su derecha, un valor positivo se convierte en negativo y un negativo se convierte en positivo. Por ejemplo:

```
Val := -2 + A
```

Negar se convierte en un operador de asignación cuando es el solo operador a la izquierda de la variable en una línea por si misma. Por ejemplo:

```
-A
```

Esto negara el valor de A y almacenara el resultado de regreso en A.

Decremento, pre- o post- '--'

El operador decremento es un operador inmediato especial que decrementa una variable en uno y asigna el nuevo valor a la misma variable. Puede usarse en expresiones variables en tiempo de ejecución. Decrementos tiene dos formas, pre-decremento y post-decremento, dependiendo de en que lado de la variable aparece. El pre-decremento aparece a la izquierda de la variable y el post-decremento aparece a la derecha de la variable. Esto es extremadamente útil en programación ya que hay muchas situaciones que llaman a un decremento de una variable justo antes o despues de usar ese valor de la variable. Por ejemplo:

```
Y := --X + 2
```

El ejemplo de arriba muestra el pre-decremento; esto significa que 'reduce antes de proporcionar el valor a la siguiente operación". Reduce el valor de X en uno, escribe ese resultado en X y proporciona el resultado al resto de la expresión. Si X comienza como 5 en este ejemplo, --X almacenara 4 en X, luego la expresión , 4 + 2 se evalúa y finalmente escribe el resultado 6 en Y. Después de esta instrucción X es igual a 4 y Y igual a 6.

```
Y := X-- + 2
```

Arriba se muestra la forma de post-decremento, significa que "reduce despues de proporcionar el valor a la siguiente operación". Proporciona el valor actual de X para la siguiente operación en la expresión y luego decrementa el valor de X en uno y escribe el resultado en X. Si X comienza en 5 en este ejemplo, X proporcionara el valor actual para la expresión (5 + 2) para ser evaluado posteriormente, despues almacenara 4 en X. La expresión 5 + 2 se evalúa y el resultado, 7, se almacena en Y. Después de esta instrucción X es igual a 4 y Y es igual a 7.

Como el decremento es siempre un operador de asignación , las reglas de asignador intermediario aplican (ver Pág. 151). Asuma que X inicio en 5 para los siguientes ejemplos:

```
Y := --X + X
```

Aquí, X primero de active a 4, despues 4 + 4 se evalúa y Y se active a 8.

```
Y := X-- + X
```

Aquí el valor actual de X, 5, se guarda para la siguiente operación (la suma) y X se decrementa a 4, entonces se evalúa 5 + 4 y Y se activa a 9.

Incremento, pre- o post- '++'

El operador de incremento es un operador inmediato especial que incrementa una variable en uno y asigna el nuevo valor a la misma variable. Solo puede usarse en tiempo de ejecución en expresiones variables. Incremento tiene dos formas, pre-incremento y post-incremento, dependiendo de que lado de la variable aparezca. El pre-incremento aparece a la izquierda y el post-incremento a la derecha de la variable. Esto es extremadamente útil en programación ya que hay muchas situaciones que llaman a incrementos de una variable justo antes o despues de usar el valor de la variable. Por ejemplo:

```
Y := ++X - 4
```

El ejemplo anterior muestra la forma de pre-incremento; significa que "incremente el valor antes de proporcionar el valor para la siguiente operación" Incrementa el valor de X en uno, y proporciona el resultado al resto de la expresión. Si X empieza en 5 en este ejemplo, ++X almacenara 6 en X, luego la expresión, 6 - 4 se evalúa y finalmente escribe el resultado 2, en Y. Después de esta instrucción X es igual a 6 y Y es igual a 2.

```
Y := X++ - 4
```

El ejemplo anterior es la forma de post-incremento; significa que "incrementa antes de proporcionar el valor a la siguiente operación". Incrementa el valor de X en uno, escribe el resultado en X. Si X empezó en 5 en este ejemplo, X++ proporcionara el valor actual para la expresión (5 - 4) para ser evaluada posteriormente, entonces almacenara 6 en X. La expresión 5 - 4 se evalúa y el resultado , 1, se almacena en Y. Después de esta instrucción X es igual a 6 y Y es igual a 1.

Como el incremento es siempre un operador de asignación las reglas de asignación intermedia aplican (ver Pág. 151). Asuma que X inicio en 5 para los siguientes ejemplos:

```
Y := ++X + X
```

Aquí , X primero se active a 6, luego 6 + 6 se evalúa y Y se active a 12.

```
Y := X++ + X
```

Aquí el valor actual de X, 5, se guarda para la siguiente operación (la suma) y se incrementa X a 6, luego 5 + 6 se evalúa y Y se activa a 11.

Multiplica, Regresa Bajo '*', '*='

Este operador también se llama multiplicador bajo, o simplemente multiplicador. Puede usarse en expresiones variables y constantes. Cuando se usa en expresiones variables o constantes enteras multiplica dos valores juntos y regresa los 32 bits mas bajos del resultado de 64-bit. Cuando se usa con expresiones constantes de punto flotante multiplica dos valores y regresa el resultado de punto flotante de precisión simple en 32-bit. Ejemplo:

```
X := Y * 8
```

Multiply-Low tiene una forma de asignación, *****=, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo,

```
X *= 20      'Forma corta de X := X * 20
```

Aquí el valor de X se multiplica por 20 y los 32 bits mas bajos del resultado se almacenan de regreso en X. La forma de asignación de Multiply-Low puede usarse en expresiones para resultados intermedios, ver Asignaciones Intermedias Pág. 151.

Multiplica, Regresa Alto '**', '**='

Este operador también se llama multiplicador alto. Puede usarse en expresiones de variables y enteros constantes, pero no para expresiones constantes de punto flotante. Multiply High multiplica dos valores juntos y regresa los 32 bits altos del resultado de 64-bit. Ejemplo:

```
X := Y ** 8
```

Si Y inicio como 536,870,912 (2^{29}) entonces Y ** 8 es igual a 1; el valor en la parte alta de de los 32 bits del resultado.

Multiply-High tiene una forma de asignación, ******=, que usa la variable a su izquierda en ambos el operando y el resultado destino. Por ejemplo,

```
X **= 20 'Short form of X := X ** 20
```

Aquí el valor de X se multiplica por 20 y los 32 bits altos del resultado se almacenan de regreso en X. La forma de asignación de Multiply-High también se puede usar en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Divide '/', '/='

Divide puede usarse en expresiones constantes y variables. Cuando se usa en expresiones variables o expresiones enteras constantes divide un valor por otro y regresa el resultado en entero 32-bit. Cuando se usa como punto flotante en expresiones constantes divide un valor por otro y regresa el resultado en punto flotante de precisión simple en 32-bit. Ejemplo:

```
X := Y / 4
```

Divide tiene una forma de asignación, **/=**, que usa una variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo:

```
X /= 20       Forma corta de X := X / 20
```

Aquí el valor de `X` se divide por 20 y el resultado entero se almacena de regreso en `X`. La forma de asignación Divide puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Modulus '//', '//='

Modulus puede usarse en expresiones constantes y variables enteras, pero no en expresiones constantes de punto flotante. Modulus divide un valor por otro y regresa el restante en entero de 32-bit. Ejemplo:

```
X := Y // 4
```

Si `Y` inicio en 5 entonces `Y // 4` es igual a 1, lo que significa que la división 5 por 4 resulta en un numero real del cual el numero fraccional es ¼, o .25.

Modulus tiene una forma de asignación, **//=**, que usa la variable a su izquierda tanto en su operando como en el resultado destino. Por ejemplo,

```
X //= 20 'Forma corta de X := X // 20
```

Aquí el valor de `X` se divide por 20 y el restante entero de 32-bit se almacena de regreso en `X`. La forma de asignación de Modulus puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Limite mínimo '#>', '#>='

El operador Limit Minimum compara dos valores y regresa el valor mas alto. Limit Minimum puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y - 5 #> 100
```

El ejemplo de arriba resta 5 de Y y limita el resultado a un valor de 100. Si Y es 120 entonces 120 – 5 = 115; es un entero mayor que 100 así que X se activa a 115. Si Y es 102 entonces 102 – 5 = 97; es menor que 100 así que X se activa a 100.

Limit Minimum tiene una forma de asignación, **#>=**, que usa la variable a su izquierda en ambos el primer operando y el resultado destino, por ejemplo:

```
X #>= 50 'Forma corta de X := X #> 50
```

Aquí el valor de X se limita a un valor mínimo de 50 y el resultado se almacena de regreso en X. La forma de asignación de Limit Minimum se puede usar en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Limite Máximo '<#', '<#='

El operador Limit Maximum compara dos valores y regresa el valor mas bajo. Limit Maximum puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y + 21 <# 250
```

El ejemplo de arriba suma 21 a Y y limita el resultado a un máximo de 250. Si Y es 200 entonces 200 + 21 = 221; es menor que 250 así que X se activa a 221. Si Y es 240 entonces 240 + 21 = 261; es mayor que 250 así que X se activa a 250.

Limit Maximum tiene una forma de asignación **<#=**, que usa la variable a su izquierda en ambos el primer operando y el resultado destino. Por ejemplo:

```
X <#= 50 'Forma corta de X := X <# 50
```

Aquí el valor de X esta limitado a un valor máximo de 50 y el resultado se almacena de regreso en X. La forma de asignación de Limit Maximum puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Raíz Cuadrada '^^'

El operador Square Root regresa el valor de la raíz cuadrada de un valor. Square Root puede usarse en expresiones constantes variables. Cuando se usa en expresiones variables o expresiones constante enteras Square Root regresa los 32-bit cortadas del resultado entero. Cuando se usa con expresiones constantes de punto flotante Square Root regresa los 32-bit en punto flotante de precisión simple. Ejemplo:

```
X := ^^Y
```

Square Root se convierte en operador de asignación cando es el operador solo a la izquierda de una variable en una línea. Por ejemplo:

```
^^Y
```

Esto almacenara la raíz cuadrada del valor de `Y` de regreso en `Y`.

Valor Absoluto '||'

El operador Absolute Value, también llamado absolute regresa el valor absoluto (la forma positiva) de un numero. Absolute Value puede usarse en expresiones constantes y variables. Cuando se usa en expresiones variables o constantes enteras Absolute Value regresa el resultado en 32-bit entero. Cuando se usa como expresión constante de punto flotante Absolute Value regresa el resultado en precisión simple de punto flotante. Ejemplo:

```
X := ||Y
```

Si `Y` es -15, el valor absolute , 15, se almacenara en `X`.

Absolute Value se convierte en operador de asignación cuando se usa solo a la izquierda de una variable en una línea. Por ejemplo:

```
||Y
```

Esto almacenara el valor absolute de `Y` de regreso en `Y`.

Signo-Extendido 7 o Post-Clear '~'

Este es un operador inmediato especial de doble propósito dependiendo en que lado de la variable aparece. Solo puede usarse en expresiones variables en tiempo de ejecución. La forma del Signo-Extendido 7 del operador aparece a la izquierda de la variable y la forma Post-Clear aparece a la derecha de la variable.

El siguiente es un ejemplo de la forma del operador Sign-Extend 7:

```
Y := ~X + 25
```

El operador Sign-Extend 7 en este ejemplo extiende el signo del valor, X en este caso, del bit 7 hasta el bit 31. Un entero con signo de 32-bit se almacena en forma de complementos de dos y el bit mas significativo (31) indica el signo del valor (positivo o negativo). Quizá hay veces donde los cálculos resultaran valores en tamaño byte que debe tratarse como enteros con signo en el rango de -128 to +127. Cuando necesita desarrollar cálculos con esos valores de tamaño byte use el operador Sign-Extend 7 para convertir el numero en la forma apropiada 32-bit entero con signo. En el ejemplo de arriba se asume que X representa el valor -20, el cual en complementos de dos de 8-bit es actualmente el valor 236 (%11101100). La porción ~X de la expresión extiende el bit de signo del bit 7 hasta el bit 31, convirtiendo el numero en la forma de complemento de dos de 32 bit de -20 (%11111111 11111111 11111111 11101100). Sumando ese valor extendido signado a 25 resulta en 5, el resultado pretendido, mientras podría haber resultado en 261 sin la apropiada extensión de signo.

El siguiente es un ejemplo de la forma del operador Post-Clear.

```
Y := X~ + 2
```

El operador Post-Clear en este ejemplo limpia la variable a 0 (todos los bits bajos) después de proporcionar su valor actual para la siguiente operación. En este ejemplo si X comenzó en 5, X~ proporcionara el valor actual para la expresión (5 + 2) para poder evaluar despues, luego almacenara 0 en X. La expresión 5 + 2 se evalúa y el resultado, 7, se almacena en Y. Después de esta instrucción X se iguala a 0 y Y se iguala a 7.

Como Sign-Extend 7 ay Post-Clear son siempre operadores de asignación, las reglas de Asignación Intermedia aplican a ambos (ver Pág. 151).

Signo-Extendido 15 o Post-Set '~~'

Este operador es inmediato especial y tiene doble propósito dependiendo de en que lado de la variable aparece. Solo puede usarse en expresiones variables en tiempo de ejecución. La forma Signo-Extendido 15 del operador aparece a la izquierda de la variable y la forma Post-Set aparece a la derecha de la variable.

El siguiente ejemplo es la forma del operador Sign-Extend 15:

```
Y := ~~X + 50
```

El operador Sign-Extend 15 en este ejemplo extiende el signo del valor, X en este caso, del bit 15 hasta el bit 31. Un entero de 32-bit con signo se almacena en complementos de dos y el bit mas significativo (31) indica el signo del valor (positivo o negativo). Algunas veces donde los

cálculos son simples el resultado en un valor tamaño Word deberá tratarse como entero signado en el rango de -32768 to +32767. Cuando necesita desarrollar mas cálculos con esos valores de tamaño word use el operador Sign-Extend 15 para convertir el numero en la forma apropiada entera de 32-bit con signo. En el ejemplo de arriba se asume que `X` representa el valor -300, el cual en la forma de complemento de dos de 16-bit es actualmente el valor 65,236 (%11111110 11010100). La porción `~~X` de la expresión extiende el bit de signo desde el bit 15 hasta el bit 31 convirtiendo el numero en la forma apropiada del complemento a dos de 32-bit de -300 (%11111111 11111111 11111110 11010100). Sumando ese valor de signo extendido a 50 el resultado es -250, que es la intención, mientras por otro lado podría haber resultado en 65,286 sin la extensión de signo apropiada.

El siguiente es un ejemplo de la forma del operador Post-Set.

```
Y := X~~ + 2
```

El operador Post-Set en este ejemplo active la variable a -1 (todos los bits altos) despues de proporcionar su valor actual para la siguiente operación. En este ejemplo si `X` comienza en 6, `X~~` proporcionara el valor actual para la expresión (6 + 2) para evaluar después, y almacenara -1 en `X`. La expresión 6 + 2 se evalúa y el resultado, 8, se almacena en `Y`. Después de esta instrucción `X` se iguala a -1 y `Y` se iguala a 8.

Como Sign-Extend 15 y Post-Set son siempre operadores de asignación aplican las reglas de Asignación Intermedia para ambos (ver Pág. 151).

Corrimiento aritmético a la Derecha '~>', '~>='

El operador de corrimiento aritmético a la derecha es igual que el operador de corrimiento a la derecha excepto que mantiene el signo como en una división por 2, 4, 8, etc. en un valor signado. El corrimiento aritmético a la derecha puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y ~> 4
```

El ejemplo de arriba mueve `Y` a la derecha en 4 bits, manteniendo el signo. Si `Y` es -3200 (%11111111 11111111 11110011 10000000) entonces `-3200 ~> 4` = -200 (%11111111 11111111 11111111 00111000). Si la misma operación se hubiera hecho con el corrimiento a la derecha el resultado seria en cambio 268,435,256 (%00001111 11111111 11111111 00111000).

El corrimiento aritmético a la derecha tiene una forma de asignación, **~>=**, que usa la variable a su izquierda del primer operando y en el resultado destino. Por ejemplo,

```
X ~>= 2      'Forma corta de X := X ~> 2
```

Aquí el valor de X se mueve a la derecha 2 bits, manteniendo el signo, y el resultado se almacena de regreso en X. La forma de asignación del corrimiento aritmético a la derecha puede usarse en expresiones para resultados de Asignaciones Intermedias, Pág. 151.

Random '?'

El operador Random es un operador inmediato especial que usa un valor de variable como semilla para crear un numero pseudo aleatorio y asigna ese numero a la misma variable. Solo puede usarse en expresiones variables en tiempo de ejecución. Random tiene dos formas al frente y reversa, dependiendo de que lado de la variable aparezca. El numero hacia adelante aparece a la izquierda de la variable y en reversa aparece a la derecha de la variable.

Random genera numero pseudo aleatorios de -2,147,483,648 to +2,147,483,647. Se llama "pseudo-random" porque los números aparecen aleatorios pero realmente son generados por una operación lógica que usa un valor semilla como golpeteo en una secuencia de arriba de 4 números esencialmente aleatorios. Si la misma semilla se usa nuevamente se genera la misma secuencia de números. La salida Random del chip Propeller es reversible, de hecho específicamente es de una longitud máxima de 32-bit, cuatro-golpes LFSR (Linear Feedback Shift Register) con golpeteo en ambos LSB (Least Significant Bit, rightmost bit) y el MSB (Most Significant Bit, leftmost bit) permitiendo una operación bidireccional.

Pensar en la secuencia pseudo aleatoria genera una lista estática simple de arriba de 4 billones de números. Empezando con un valor semilla particular moviéndose hacia adelante da como resultado una lista de números. Sin embargo si toma el ultimo numero generado y se usa como valor semilla, moviéndolo hacia atras se podría terminar con la lista de los mismos números anteriores, pero en sentido contrario. Esto es útil para muchas aplicaciones.

Aquí un ejemplo:

```
?X
```

el ejemplo muestra un Random al frente; usa el valor actual de X para recuperar el siguiente numero pseudo aleatorio hacia adelante y almacena el valor de regreso en X. Ejecutando ?X nuevamente da como resultado todavía diferente almacenado nuevamente en X.

```
X?
```

El ejemplo de arriba muestra la forma de reversa de Random; usa el valor actual de X para tomar un valor pseudo aleatorio en reversa y almacenarlo en la misma X. Si ejecuta X? nuevamente el resultado es otro numero diferente, que se almacena nuevamente en X.

Como Random es siempre un operador de asignación las reglas de Asignación Intermedia aplican a la instrucción (ver Pág. 151).

Bitwise Decode '|<'

El operador Bitwise Decode decodifica un valor (0 – 31) en un valor long de 32-bit con un bit simple activado en alto correspondiente a la posición del bit del valor original. Bitwise Decode puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Ejemplo:

```
Pin := |<PinNum
```

El ejemplo de arriba activa Pin igual al valor de 32-bit en el cual el bit sencillo en alto corresponde a la posición indicada por PinNum.

Si PinNum es 3, Pin se activa igual a %00000000 00000000 00000000 00001000.

Si PinNum es 31, Pin se activa igual a %10000000 00000000 00000000 00000000.

Hay muchas formas de usar Bitwise Decode, pero la mas útil es para convertir de un pin E/S a un patrón de 32-bit que describe ese numero de pin en relación a los registros E/S. Por ejemplo Bitwise Decode es muy útil para los parámetros de mascara de las instrucciones WAITPEQ y WAITPNE.

Bitwise Decode se convierte en operador de asignación cuando esta el operador solo a la izquierda de la variable en una línea. Por ejemplo:

```
|<PinNum
```

Esto almacenara el valor decodificado de PinNum de regreso en PinNum.

Bitwise Encode '>|'

El operador Bitwise Encode codifica un valor de 32-bit long en un valor (0 – 32) que representa el grupo de bit mas alto mas 1. Bitwise puede usarse en variables y expresiones constantes enteras pero no puede usarse en expresiones constantes de punto flotante. Ejemplo:

```
PinNum := >|Pin
```

El ejemplo anterior activa PinNum igual al numero del bit mas alto activado en Pin, mas 1.

Si Pin es %00000000 00000000 00000000 00000000, PinNum es 0; no hay bits activos.

Si Pin es %00000000 00000000 00000000 10000000, PinNum es 8; el bit 7 es activo.

Si Pin es %10000000 00000000 00000000 00000000, PinNum es 32; el bit 31 es activo.

si Pin es %00000000 00010011 00010010 00100000, PinNum es 21; el bit 20 es el bit mas alto activo.

Bitwise Shift Left '<<', '<<='

El operador Bitwise Shift Left mueve los bits del primer operando a la izquierda por la cantidad de bits indicada en el segundo operando. Los originales MSBs (bits de la izquierda extrema) se descargan y los nuevos LSBs (bits de la extrema derecha) se ponen a ceros. El Bitwise Shift Left puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y << 2
```

Si Y inicio como:

%10000000 01110000 11111111 00110101

...el operador Bitwise Shift Left moverá ese valor a la izquierda dos bits dejando X a:

%00000001 11000011 11111100 11010100

Como la naturaleza del binario en la base es mover un valor a la izquierda es como multiplicar ese valor por potencias de dos, 2^b, donde b es el numero de los bits que se mueve.

Bitwise Shift Left tiene una forma de asignación, **<<=**, que usa la variable a la izquierda tanto en el primer operando como en el resultado destino. Por ejemplo,

```
X <<= 4  'Forma corta de X := X << 4
```

Aquí el valor de X se mueve a la izquierda cuatro bits y se almacena de regreso en X. La forma de asignación de Bitwise Shift Left puede ser usada en expresiones de resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Bitwise Shift Right '>>', '>>='

El operador Bitwise Shift Right mueve los bits del primer operando a la derecha por el numero de bits indicado en el segundo operando. Los originales LSBs (bits de la extrema derecha) se descargan y los nuevos MSBs (bits de la extrema izquierda) se activan a cero. El Bitwise Shift Right puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y >> 3
```

Si Y inicio como:

%10000000 01110000 11111111 00110101

...el operador Bitwise Shift Right moverá ese valor a la derecha tres bits dejando X a:

%00010000 00001110 00011111 11100110

Como la naturaleza binaria es base-2, al mover un valor a la derecha es como desarrollar una división de enteros del valor en potencias de dos, 2^b, donde b es el numero de bits a mover.

Bitwise Shift Right tiene una forma de asignación, **>>=**, que usa la variable a su izquierda tanto en el primer operando y en el resultado destino. Por ejemplo,

```
X >>= 2  'Forma corta de X := X >> 2
```

Aquí el valor de X se mueve a la derecha dos bits y se almacena de regreso en X. La forma de asignación de Bitwise Shift Right puede ser usada en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Bitwise Rotate Left '<-', '<-='

El operador Bitwise Rotate Left es similar al operador Bitwise Shift Left, excepto que los MSBs (bits de la extrema izquierda) se rotan con los LSBs (bits de la extrema derecha). Bitwise Rotate Left pede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y <- 4
```

Si Y inicio como:

%10000000 01110000 11111111 00110101

El operador Bitwise Rotate Left rotara el valor a la izquierda por 4 bits, moviendo el valor original de los cuatro MSBs a los nuevos cuatro LSBs, y dejando X a:

%00000111 00001111 11110011 01011000

Bitwise Rotate Left tiene una forma de asignación, **<-=**, que utiliza la variable a la izquierda tanto del primer operando como del resultado destino. Por ejemplo,

```
X <-= 1  'Forma corta de X := X <- 1
```

Aquí el valor de X se rota un bit y se almacena de regreso en X. La forma de asignación de Bitwise Rotate Left puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Bitwise Rotate Right '->', '->='

El operador Bitwise Rotate Right es similar al operador Bitwise Shift Right, excepto que los LSBs (bits de la extrema derecha) se rotan con los MSBs (bits de la extrema izquierda). Bitwise Rotate Right puede usarse en ambas expresiones variables y constantes enteras, pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y -> 5
```

Si Y inicio como:

%10000000 01110000 11111111 00110101

...el operador Bitwise Rotate Right rotara ese valor por 5 bits, moviendo el valor original por cinco bits, Moviendo los originales cinco LSBs a los nuevos cinco MSBs, dejando X a:

%10101100 00000011 10000111 11111001

Bitwise Rotate Right tiene una forma d asignación, `->=`, que usa la variable a su izquierda para el operando y el resultado destino. Por ejemplo,

```
X ->= 3  'Forma corta de X := X -> 3
```

Aquí el valor de `X` se rota a la derecha por tres bits y se almacenan de regreso en `X`. La forma de asignacion de Bitwise Rotate Right puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Bitwise Reverse '><', '><='

El operador Bitwise Reverse regresa los bits menos significativos del primer operando en su orden inverso. El total de números de bits menos significativos a invertirse se indica por el segundo operando. Todos los otros bits a la izquierda de los bits invertidos son ceros en el resultado. Bitwise Reverse puede usarse en expresiones variables y constantes enteros, pero no en expresiones constantes de punto flotante. Ejemplo:

```
X := Y >< 6
```

Si Y inicio como:

%10000000 01110000 11111111 00110101

...el operador Bitwise Reverse regresara los seis LSBs invertidos y los demás bits en ceros, dejando `X` a:

%00000000 00000000 00000000 00101011

Bitwise Reverse tiene una forma de asignación, **><=**, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo,

```
X ><= 8  'Forma corta de X := X >< 8
```

Aquí los ocho LSBs del valor de X se colocan en sentido inverso, los demás bits se ponen a ceros y el resultado se almacena en X. La forma de asignacion de Bitwise Reverse puede usarse en expresiones de resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Observe que especificar 0 como el numero de bits a invertir es lo mismo que especificar 32.

Bitwise AND '&', '&='

El operador Bitwise AND desarrolla un bitwise AND de los bits del primer operando con los bits del segundo operando. Bitwise AND puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante.

Cada bit de dos operandos esta sujeto a la siguiente lógica:

Tabla 2-11: Bitwise AND Tabla de Verdad

Estado de Bit		Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Ejemplo:

```
X := %00101100 & %00001111
```

En el ejemplo de arriba ANDs %00101100 con %00001111 escribe el resultado %00001100, en X.

Bitwise AND tiene una forma de asignación , **&=**, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo,

```
X &= $F  'Forma corta de X := X & $F
```

Aquí el valor de X es AND con $F y el resultado de almacena en X. La forma de asignación del Bitwise AND puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Tenga cuidado de no confundir el Bitwise AND '&' con la forma booleanas AND 'AND'. Bitwise AND es para manipulación de bits mientras que la forma booleanas es para propósitos de comparación (ver Pág. 171).

Bitwise OR '|', '|='

El operador Bitwise OR desarrolla la operación OR de los bis del primer operando con los bits del segundo operando. Bitwise OR puede usarse en ambas expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante. Cada bit de los dos operandos esta sujeto a la siguiente lógica:

Tabla 2-12: Bitwise OR Tabla de Verdad

Estado de Bit		Resultado
0	0	0
0	1	1
1	0	1
1	1	1

Ejemplo:

```
X := %00101100 | %00001111
```

En el ejemplo anterior se realiza OR de %00101100 con %00001111 y se escribe el resultado %00101111 a X.

El Bitwise OR tiene una forma de asignación, |=, que usa la variable a su izquierda tanto en el primer operando como en el resultado destino. Por ejemplo,

```
X |= $F  'Forma corta de X := X | $F
```

Aquí el valor de X se hace OR con $F y el resultado se almacena de regreso en X. La forma de asignación del Bitwise OR también puede usarse dentro de expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Tenga cuidado de no confundir el Bitwise OR '|'con la forma booleanas OR 'OR'. El Bitwise OR es para manipular bits mientras que el booleano OR es para comparar (ver Pág. 172).

Bitwise XOR '^', '^='

El operador Bitwise XOR desarrolla una operación XOR de los bits del primer operando con los bits del segundo operando. El Bitwise XOR puede usarse en expresiones variables y constantes enteras pero no en expresiones constantes de punto flotante.

Cada bit de los dos operandos esta sujeto a la siguiente lógica:

Tabla 2-13: Bitwise XOR Tabla de Verdad		
Estado de Bit		Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Ejemplo:

```
X := %00101100 ^ %00001111
```

El ejemplo anterior calcula el XOR de %00101100 con %00001111 y escribe el resultado %00100011 a X.

El Bitwise XOR tiene una forma de asignación, **^=**, que usa la variable a su izquierda en el primer operando y el resultado destino. Por ejemplo,

```
X ^= $F  'Forma corta de X := X ^ $F
```

Aquí el valor de X se hace XOR con $F y el resultado se almacena de regreso en X. La forma de asignación de Bitwise XOR puede utilizarse para expresiones de resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Bitwise NOT '!'

El operador Bitwise NOT '**!**' desarrolla un bitwise NOT (inverso o complemento de uno) de los bits del operando que lo siguen. El Bitwise NOT puede usarse para expresiones variables y constantes enteras pero no para expresiones constantes de punto flotante.

Cada bit de dos operandos esta sujeto a la siguiente lógica:

Tabla 2-14: Bitwise NOT Tabla de Verdad	
Estado de Bit	Resultado
0	1
1	0

Ejemplo:

```
X := !%00101100
```

El ejemplo anterior hace NOT `%00101100` y escribe el resultado `%11010011` a X.

El Bitwise NOT se convierte en operador de asignación cuando esta solo a la izquierda de la variable en una línea. Por ejemplo:

```
!Flag
```

Esto almacenara el valor invertido del valor `Flag` de regreso en `Flag`.

Tenga cuidado de no confundir el Bitwise NOT '**!**'con el booleano NOT '**NOT**'. El Bitwise NOT es para manipulación de bits mientras NOT es para comparación (ver Pág. 172).

Booleano AND 'AND', 'AND='

El operador Booleano AND '**AND**' compara dos operandos y regresa **TRUE** (-1) si ambos valores son **TRUE** (no-cero), o regresa **FALSE** (0) si uno o mas operandos son **FALSE** (0). El Booleano AND puede usarse en expresiones variables y constantes.

Ejemplo:

```
X := Y AND Z
```

El ejemplo de arriba compara el valor de `Y` con el valor de `Z` y activa `X` a: **TRUE** (-1) si ambos `Y` y `Z` son no-cero, o **FALSE** (0) si alguno es cero. Durante la comparación forza cada uno de los valores a -1 si con no-cero, haciendo cualquier valor diferente de 0 un -1, así la comparación de realiza: "Si `Y` es verdad y `Z` es verdad…"

Este operador se usa continuamente en combinación con otros operadores de comparación tales como los del siguiente ejemplo.

```
IF (Y == 20) AND (Z == 100)
```

Este ejemplo evalúa el resultado de `Y == 20` contra el de `Z == 100`, y si ambos son verdaderos, el operador booleano AND regresa **TRUE** (-1).

El Booleano AND tiene una forma de asignación, **AND=**, que usa la variable a su izquierda en el primer operando y el resultado destino. Por ejemplo,

```
X AND= True     'Forma corta de X := X AND True
```

Aquí el valor de `X` se forza a **TRUE** si es no-cero y se compara con **TRUE** y el resultado booleano (**TRUE** / **FALSE**, -1 / 0) se almacena de regreso en `X`. la forma de asignación del booleano AND puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

tenga cuidado de no confundir el Boleano AND 'AND' con el Bitwise AND '&'. El Booleano AND es para propósitos de comparación mientras el Bitwise AND es para manipulación de bits (ver Pág. 168).

Booleano OR 'OR', 'OR='

El operador booleano OR 'OR' compara dos operandos y regresa TRUE (-1) si cualquiera de los valores es TRUE (no-cero), o regresa FALSE (0) si ambos operandos son FALSE (0). El Booleano OR puede usarse en ambas expresiones variables y constantes. Ejemplo:

```
X := Y OR Z
```

El ejemplo de arriba compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si alguno, Y o Z es no-cero, o FALSE (0) si ambos Y y Z son cero. Durante la comparación se forza cada uno de los dos valores a -1 si son no-cero, haciendo cualquier valor diferente de 0 un -1, de esta forma la comparación se convierte en: "Si Y es verdad o Z es verdad…"

Frecuentemente este operador se usa en combinación con otros operadores de comparación tal como en el siguiente ejemplo.

```
IF (Y == 1) OR (Z > 50)
```

Este ejemplo evalúa el resultado de Y == 1 contra el de Z > 50, y si alguno en verda del operador Bolle ano OR regresa TRUE (-1).

El Boolean OR tiene una forma de asignación, OR=, que usa la variable a la izquierda de el primer operando o el resultado destino. por ejemplo,

```
X OR= Y      'Forma corta de X := X OR Y
```

Aquí el valor de X se forza a TRUE si es no-cero y luego se compara con Y (también forzado a TRUE si es no-cero) y el resultado booleano (TRUE / FALSE, -1 / 0) s almacena de regreso en X. La forma de asignación del booleano OR puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Tenga cuidado de no confundir el Booleano OR 'OR' con el Bitwise OR'|'. El Boolean OR es para comparación mientras el Bitwise OR es para manipulación (ver Pág. 169).

Booleano NOT 'NOT'

El operador Booleano NOT 'NOT' regresa TRUE (-1) si el operando es FALSE (0), o regresa FALSE (0) si el operando es TRUE (no-cero). El Booleano NOT puede usarse en expresiones variables y constantes. Ejemplo:

```
X := NOT Y
```

En el ejemplo anterior se regresa el booleano opuesto de Y; TRUE (-1) si Y es cero, o FALSE (0) si Y es no-cero. Durante la comparación se forza el valor de Y a -1 si es no-cero, haciendo cualquier valor diferente de 0 un -1, de esta forma se compara: “Si NOT es verdad” o “Si NOT es falso”

Regularmente este operador se utiliza en combinación con otros operadores de comparación como se muestra en el siguiente ejemplo.

```
IF NOT ( (Y > 9) AND (Y < 21) )
```

Este ejemplo evalúa el resultado de (Y > 9 AND Y < 21), y regresa el booleano opuesto del resultado; TRUE (-1) si Y esta en el rango de 10 a 20, en este caso.

El Booleano NOT se convierte en operador de asignación cuando esta solo a la izquierda de la variable en un alinea. Por ejemplo:

```
NOT Flag
```

Esto almacenara el Booleano opuesto de Flag de regreso en Flag.

Tenga cuidado de no confundir el Booleano NOT ‘NOT’ con el Bitwise NOT‘!’. El Booleano NOT es para comparar mientras el Bitwise NOT es para manipulación de bits (ver Pág. 170).

Booleano Is Equal ‘==’, ‘===’

El operador Booleano Is Equal compara dos operandos y regresa TRUE (-1) si ambos valores son los mismos, o de otra forma regresa FALSE (0). Is Equal puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y == Z
```

El ejemplo de arriba compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si Y es el mismo valor de Z, o FALSE (0) si Y no es el mismo valor de Z.

Este operador se usa frecuentemente en expresiones condicionales tales como.

```
IF (Y == 1)
```

Aquí el operador Is Equal regresa TRUE si Y es igual a 1.

Is Equal tiene una forma de asignación, ===, que usa la variable a su izquierda en el primer operado y en el resultado destino. Por ejemplo,

```
X === Y     'Forma corta de X := X == Y
```

Aquí X se compara con Y, y si son iguales, X se activa a TRUE (-1), de otra forma se activa X a FALSE (0). La forma de asignación de Is Equal puede usarse para expresiones de resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Booleano Is Not Equal '<>', '<>='

El operador Booleano Is Not Equal compara dos operandos y regresa True (-1) si los valores no son los mismos, de otra forma regresa FALSE (0). Is Not Equal puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y <> Z
```

El ejemplo de arriba compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si Y no es el mismo valor que Z, o FALSE (0) si Y es el mismo valor que Z.

Este operador se usa frecuentemente para expresiones condicionales como en el siguiente ejemplo.

```
IF (Y <> 25)
```

Aquí, el operador Is Not Equal regresa TRUE si Y no es 25.

Is Not Equal tiene una forma de asignación, <>=, que usa la variable a su izquierda tanto en el primer operando como en su resultado destino. Por ejemplo,

```
X <>= Y      'Forma corta de X := X <> Y
```

Aquí, X se compara con Y, y si no son iguales, X se activa a TRUE (-1), de otra forma X se activa a FALSE (0). La forma de asignación de Is Not Equal puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Booleano Is Less Than '<', '<='

El operador Booleano Is Less Than compara dos operandos y regresa TRUE (-1) si el primer valor es menor que el segundo valor, de otra forma regresa FALSE (0). Is Less Than puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y < Z
```

El ejemplo anterior compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si Y es menor que el valor de Z, o FALSE (0) si Y es igual o mayor que el valor de Z.

Este operador se usa comúnmente en expresiones condicionales como en el siguiente ejemplo.

```
IF (Y < 32)
```

Aquí el operador Is Less Than regresa **TRUE** si Y es menor que 32.

Is Less Than tiene su forma de asignación, **<=**, que usa la variable a la izquierda del primer operando y el resultado destino. Por ejemplo,

```
X <= Y      'Forma corta de X := X < Y
```

Aquí se compara X con Y, y si X es menor que Y, X se activa a **TRUE** (-1), de otra forma X se activa a **FALSE** (0). La forma de asignación de Is Less Than puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Booleano Is Greater Than '>', '>='

El operador Booleano Is Greater Than compara dos operandos y regresa **TRUE** (-1) si el primer valor en mayor que el segundo valor, de otra forma regresa **FALSE** (0). Is Greater Than puede usarse en expresiones variables y constantes. Ejemplo:

```
X := Y > Z
```

El ejemplo anterior compara el valor de Y con el valor de Z y activa X a: **TRUE** (-1) si Y es mayor que el valor de Z, o **FALSE** (0) si Y es igual o menor que el valor de Z.

Este operador es frecuentemente usado en expresiones condicionales tales como:

```
IF (Y > 50)
```

Aquí el operador Is Greater Than regresa **TRUE** si Y es mayor que 50.

Is Greater Than tiene una forma de asignación, **>=**, que usa la variable a su izquierda en el primer operando o el resultado destino. Por ejemplo,

```
X >= Y      'Forma corta de X := X > Y
```

Aquí X se compara con Y, y si X es mayor que Y, X se activa a **TRUE** (-1), de lo contrario X se activa a **FALSE** (0). la forma de asignación de Is Greater Than puede usarse para expresiones de resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Booleano Is Equal or Less '=<', '=<='

El operador Booleano Is Equal or Less compara dos operandos y regresa **TRUE** (-1) si el primer valor es igual a o menor que el segundo valor, de lo contrario regresa **FALSE** (0). Is Equal or Less puede usarse en expresiones variables o constantes. Ejemplo:

```
X := Y =< Z
```

El ejemplo anterior compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si Y es igual a o menor que el valor de Z, o FALSE (0) si Y es mayor que el valor de Z.

Este operador se usa normalmente en expresiones condicionales tales como la que se ve en el siguiente ejemplo:

```
IF (Y =< 75)
```

Aquí el operador Is Equal or Less regresa TRUE si Y es igual a o menor que 75.

Is Equal or Less tiene una forma de asignación, **=<=**, que usa la variable a su izquierda en el primer operando y en el resultado destino. Por ejemplo,

```
X =<= Y      'Forma corta de X := X =< Y
```

Aquí se compara X con Y, y si X es igual a o menor que Y, X se activa a TRUE (-1), de lo contrario X se pone a FALSE (0). La forma de asignación de Is Equal or Less puede usarse en expresiones para resultados intermedios; ver Asignaciones Intermedias, pag 151.

Booleano Is Equal or Greater '=>', '=>='

El operador Booleano Is Equal or Greater compara dos operandos y regresa TRUE (-1) si el primer valor es igual o mayor que el segundo valor, de otra forma regresa FALSE (0). Is Equal or Greater puede usarse en ambas expresiones, variables y constantes. Ejemplo:

```
X := Y => Z
```

El ejemplo anterior compara el valor de Y con el valor de Z y activa X a: TRUE (-1) si Y es igual o mayor a Z, o FALSE (0) si Y es menor que el valor de Z.

Este operado se usa en expresiones condicionales como las del siguiente ejemplo.

```
IF (Y => 100)
```

En este ejemplo el operador Is Equal or Greater regresa TRUE si Y es igual o mayor que 100.

Is Equal or Greater tiene una forma de asignación, **=>=**, que usa la variable a la izquierda de el primer operando y en el resultado destino. Por ejemplo,

```
X =>= Y      'Forma corta de X := X => Y
```

Aquí, X se compara con Y, y si X es igual o mayor que Y, X se activa como TRUE (-1), de otra forma X se activa FALSE (0). La forma de asignación de Is Equal or Greater puede ser usada en expresiones para resultados intermedios; ver Asignaciones Intermedias, Pág. 151.

Symbol Address '@'

El operador Symbol Address regresa la dirección del símbolo que lo sigue. Symbol Address puede usarse en expresiones variables y constantes enteras pero no en expresiones contantes de punto flotante. Ejemplo:

```
BYTE[@Str] := "A"
```

En este ejemplo el operador Symbol Address regresa la dirección del símbolo Str, el cual es usado por el arreglo de memoria BYTE para almacenar el caracter "A" en esa dirección.

Symbol Address se utiliza frecuentemente para pasar la dirección de cadenas y estructuras de datos, definidas en bloques DAT, a métodos que operan en ellos.

Es importante notar que este es un operador especial que se comporta diferente en expresiones variables. En tiempo de ejecución, como en nuestro ejemplo anterior, regresa la dirección absoluta del símbolo que le sigue. Esta dirección absoluta en tiempo de ejecución consiste en la dirección base del programa del objeto mas la dirección Offset del símbolo.

En expresiones constantes solo regresa el Offset del símbolo dentro del objeto. No puede regresar la dirección absoluta, efectiva en tiempo de ejecución, porque esa dirección cambia dependiendo de la dirección actual del objeto. Para usar apropiadamente Symbol Address en una constante, vea el Operador Object Address Plus Symbol operador a continuación.

Object Address Plus Symbol '@@'

El operador Object Address Plus Symbol regresa el valor del símbolo que lo sigue mas la dirección base del programa del objeto actual. Object Address Plus Symbol solo puede usarse en expresiones variables.

Este operador es útil cuando se crea una tabla de direcciones Offset, luego en tiempo de ejecución, se usan esos Offset para referenciar las direcciones absolutas en tiempo de ejecución que representan. Por ejemplo, un bloque DAT puede contener un numero de cadenas de las cuales se quiere acceso directo e indirecto. Aquí se presenta un ejemplo del bloque DAT que contiene cadenas.

```
DAT
  Str1  byte  "Hola.", 0
  Str2  byte  "Esto es un ejemplo ", 0
  Str3  byte  "de cadenas en Bloques DAT.",0
```

En tiempo de ejecución podemos accesar esas cadenas directamente, usando `@Str1`, `@Str2`, y `@Str3`, pero accesarlas indirectamente es problemático porque cada cadena es de una longitud diferente; haciendo difícil de usar cualquiera como cálculos de base indirecta.

La solución podría parecer simplemente haciendo otra tabla de direcciones como en::

```
DAT
  StrAddr  word  @Str1, @Str2, @Str3
```

Esto crea una tabla de words, empezando en `StrAddr`, donde cada word contiene la dirección de una cadena única. Desafortunadamente para constantes en tiempo de compilación (como en la tabla `StrAddr`), las direcciones que se regresan por **@** es solo el Offset de la dirección del tiempo de compilación, en vez de la dirección absoluto de tiempo de ejecución del símbolo. Para obtener la verdadera dirección de tiempo de ejecución, necesitamos agregar la dirección Offset del símbolo de la dirección del programa base del objeto. Esto es lo que el operador Object Address Plus Symbol hace. Ejemplo:

```
REPEAT Idx FROM 0 TO 2
  PrintStr(@@StrAddr[Idx])
```

El ejemplo anterior incrementa `Idx` de 0 a 2. la instrucción `StrAddr[Idx]` recupera el Offset de la cadena almacenada en tiempo de compilación almacenada en el elemento `Idx` de la tabla `StrAddr`. El operador **@@** en frente de la instrucción `StrAddr[Idx]` agrega la dirección base del objeto al valor Offset del tiempo de compilación que se recupero, resultando en una dirección valida en tiempo de ejecución para la cadena. El método `PrintStr`, cuyo código no se muestra en este ejemplo, puede usar esa dirección para procesar cada caracter de la cadena.

OUTA, OUTB

Registro: Registros de salida para puertos A y B de 32-bit.

((PUB PRI))
OUTA ⟨[*Pin(s)*]⟩

((PUB PRI))
OUTB ⟨[*Pin(s)*]⟩ (Reservado para uso futuro)

Regresa: Valor actual del pin de salida para el puerto A o B, si se usa como fuente variable.

- *Pin(s)* es una expresión opcional o una expresión de rango que especiita el pin E/S o pins para accesar en Puerto A (0-31) o Puerto B (32-63). Si se da como expresión simple solo se acceso el pin especificado. Si se da como expresión de rango (dos expresiones en formato de rango; x..y) se accesan los pins contiguos en la expresión del principio al fin.

Explicación

OUTA y OUTB son dos de seis registros (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pin de E/S. El registro OUTA mantiene el estado de salida de cada uno de los 32 pins E/S en el puerto A, los bits 0 a 31 corresponden de P0 a P31. El registro OUTB mantiene el estado de salida de cada uno de los pins en el Puerto B; los bits 0 a 31 corresponden de P32 a P63.

NOTA: OUTB esta reservado para uso futuro; el Propeller P8X32A no incluye los pins del Puerto B así que solo OUTA se discutirá en esta sección.

OUTA se usa para activar y obtener el estado actual de la salida de uno o mas pins de E/S en el Puerto A. Un bit bajo (0) activa el correspondiente pin E/S a Tierra. Un bit alto (1) activa el pin correspondiente a VDD (3.3 Volts). Todos los bits del registro OUTA están puestos a cero (0) automáticamente cuando se inicia un cog.

Cada cog tiene acceso a los pins E/S en cualquier momento. Básicamente todos los pins están conectados directamente a cada cog para que no exista acceso mutuamente exclusivo de por medio. CAda cog mantiene su propio registro OUTA que le da la habilidad de activar el estado de cualquier pin de E/S. Cada estado de salida del cog esta en OR con el de los estados de salida de los otros cogs y el valor resultante se convierte en el estado de la salida del Puerto A, pins P0 a P31. El resultado es que el estado de cada pin E/S es el “OR-cableado” del colectivo del cog. Vea Pins E/S en Pág. 26 para mas información.

Observe que cada estado de salida del cog esta formado por el OR de los estados del hardware interno (Registro de salida, Generador de Video, etc.) y estos está en AND con su dirección del estado del registro

Un pin E/S tiene como salida alto o bajo, como se especifica en el estado de salida del cog si y solo si ese bit del pin esta en alto en el registro (DIRA). De otra forma especifica el pin como entrada y su estado de salida se ignora.

Esta configuración puede fácilmente describirse con las siguientes reglas simples:

A. Un pin es bajo si todos los cogs donde esta activado a salida están activados a bajo.
B. Un pin es alto si cualquiera de los cogs donde esta activado a salida lo active alto.

Si un cog esta deshabilitado, su registro de dirección se trata como si fuera 0, ocasionando que no se haga uso de influencia sobre la dirección y estado de los pins E/S.

Observe que debido a la naturaleza del "OR-cableado" de los pins E/S no es posible una contención eléctrica entre cogs, por lo que todos pueden accesar los pins E/S simultáneamente. Depende del desarrollador de la aplicación asegurarse que dos cogs no ocasionen problemas lógicos en el mismo pin E/S durante la ejecución.

Usando OUTA

Activa o limpia un bit en OUTA para afectar el estado de salida del pin E/S según se desea. Asegúrese de activar los bits correspondientes DIRA para hacer el pin salida. Por ejemplo:

```
DIRA := %00000100_00110000_00000001_11110000
OUTA := %01000100_00110000_00000001_10010000
```

La línea DIRA de arriba actívalos pins E/S 26, 21, 20, 8, 7, 6, 5 y 4 a salidas y el resto a entradas. La línea OUTA active los pins E/S 30, 26, 21, 20, 8, 7, y 4 a alto, el resto a bajos. El resultado es que los pins E/S 26, 21, 20, 8, 7, y 4 son salidas altas y los pins 6 y 5 bajas. El pin E/S 30 esta como entrada (de acuerdo a DIRA) así que el alto en el bit 30 de OUTA se ignora y el pin se mantiene como entrada de acuerdo al cog.

Usando el campo opcional *Pin(s)*, y los operadores unarios post-clear (**~**) y post-set (**~~**), el cog puede afectar un pin E/S (un bit) a la vez. El campo *Pin(s)* trata a los registros del pin como un arreglo de 32 bits. Por ejemplo:

```
DIRA[10]~~                'Activa P10 a salida
OUTA[10]~                 'Hace P10 bajo
OUTA[10]~~                'Hace P10 alto
```

La primera línea en el código de arriba active el pin 10 a salida. La segunda línea limpia la salida de P10, haciendo P10 salida baja (aterrizada). La tercera línea activa P10 como salida alta(VDD).

En Spin, el registro **OUTA** soporta una expresión de forma especial llamada expresión de rango, el cual permite afectar un grupo de pins E/S a la vez sin afectar otros fuera del rango especificado. Para afectar múltiples y seguidos pins E/S a la vez use la expresión de rango (como x..y) en el campo *Pin(s)*.

```
  DIRA[12..8]~~               'Activa DIRA12:8 (P12-P8 a salida)
  OUTA[12..8] := %11001       'Activa P12:8 a 1, 1, 0, 0, y 1
```

La primer línea, "DIRA…," activa P12, P11, P10, P9 y P8 a salidas; los demás pins se mantienen en el estado previo. La segunda línea, "OUTA…," activa P12, P11, y P8 a salida alta y P10 y P9 a salida baja.

IMPORTANTE: El orden de los valores en una expresión de rango afecta su uso. Por ejemplo el siguiente ejemplo cambia el orden del rango del ejemplo anterior.

```
  DIRA[8..12]~~               'Activa DIRA8:12 (P8-P12 a salida)
  OUTA[8..12] := %11001       'Activa OUTA8:12 a 1, 1, 0, 0, y 1
```

Aquí los bits **DIRA** del 8 al 12 se activan en alto (igual que antes) pero los bits de **OUTA** 8, 9, 10, 11 y 12 se igualan a 1, 1, 0, 0, y 1, respectivamente haciendo P8, P9 y P12 salidas altas y P10 y P11 salidas bajas.

Esta es una herramienta ponderosa de expresiones de rango, pero si no se tiene cuidado puede ocasionar comportamientos y resultados inesperados.

Normalmente **OUTA** es de solo escritura pero también se puede leer para recuperar el estado actual del pin E/S. Esto es SOLO el estado de la salida del cog, no necesariamente la salida actual del pin en el chip Propeller, ya que estas pueden afectarse por otros cogs o incluso por hardware de E/S (Generador de Video, Contador A, etc.). El siguiente ejemplo asume que Temp es una variable creada en otro lugar:

```
  Temp := OUTA[15..13]        'Obtiene el estado de la llave de salida
de P15 a P13
```

El ejemplo anterior activa Temp igual a los bits de **OUTA** 15, 14, y 13; ejemplo, los bits mas bajos de Temp ahora son iguales a OUTA15:13 y los otros bits de Temp se limpian a cero.

PAR

Registro: Registro de parámetro de Inicio de Cog

((PUB PRI))
  PAR

Regresa: La dirección que pasa en el inicio de código ensamblador con COGINIT o COGNEW.

Explicación

El registro PAR contiene el valor de la dirección que se paso al campo *Parameter* de una instrucción COGINIT o COGNEW; ver COGINIT, Pág. 79 y COGNEW, Pág. 81. El contenido del registro PAR se usa en el código ensamblador Propeller para localizar y operar en memoria compartida entre código Spin y código ensamblador.

Como la intención del registro PAR es contener una dirección de inicio de cog, el valor que se almacena en el a través de COGINIT y COGNEW esta limitado a 14 bits: un Word de 16-bit con los dos bits mas bajos puestos a cero.

Usando PAR

PAR se afecta por código Spin y se usa por el código ensamblador como mecanismo apuntador de memoria para apuntar a la memoria principal compartida entre los dos. Ya sea la instrucción COGINIT o COGNEW cuando se inicia ensamblador Propeller en un cg se afecta el registro PAR. Por ejemplo:

```
VAR
  long   Shared                         'Variable compartida (Spin y
Ensamblador)

PUB Main | Temp
  cognew(@Process, @Shared)             'Inicia ensamblador, pasa la
dirección compartida
  repeat
    <Hace algo con las vars compartidas >

DAT
                  org 0
Process           mov Mem, PAR          'Recupera la dirección de
memoria compartida  :loop <hace algo >
```

```
                    wrlong ValReg, Mem  'Mueve valor ValReg para
compartir
                    jmp #:loop
Mem       res  1
ValReg    res  1
```

En el ejemplo anterior el método `Main` inicia la rutina ensamblador `Process` en un cog Nuevo con **COGNEW**. El segundo parámetro de **COGNEW** se usa por `Main` para pasar la dirección de la variable, `Shared`. La rutina ensamblador, `Process`, recupera ese valor de dirección del registro **PAR** y lo almacena localmente en `Mem`. Luego hace algunas tareas, actualizando su registro local `ValReg` (creado al final del bloque **DAT**) y finalmente actualice la variable `Shared` a través de `wrlong ValReg, Mem`.

En ensamblador Propeller el registro **PAR** es de solo lectura así que solo se puede accesar como valor fuente (campo-s) (ejemplo: **mov** *dest, source*).

PHSA, PHSB

Registro: Registros de Fase Contador A y Contador B

((PUB PRI))
PHSA

((PUB PRI))
PHSB

Regresa: Valor actual del registro de fase del contador A y B, si se usa como una fuente variable.

Explicación

PHSA y PHSB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores del cog. Cada cog tiene dos módulos contadores idénticos (A y B) que pueden desarrollar muchas tareas repetitivas. Los registros PHSA y PHSB contienen valores que pueden leerse directamente o ser escritos por el cog, pero también pueden acumularse con el valor de FRQA y FRQB, respectivamente, potencialmente en cada ciclo del Reloj del Sistema. Ver CTRA en Pág. 98 para mas información.

Usando PHSA y PHSB

PHSA y PHSB pueden leerse o escribirse como otros registros o variables pre-definidas. Por ejemplo:

```
PHSA := $1FFFFFFF
```

El código de arriba activa PHSA a $1FFFFFFF. Dependiendo del campo CTRMODE del registro CTRA, este valor puede quedar igual, o incrementar automáticamente con el valor de FRQA a una frecuencia determinada por el reloj de sistema y el pin de E/S primario y/o secundario. Ver CTRA, CTRB en Pág. 98 para mayor información.

Observe que en Ensamblador Propeller los registros PHSA y PHSB no pueden usarse en operaciones leer-modificar-escribir en el campo destino de una instrucción. En cambio deben realizarse operaciones separadas de leer, modificar y escribir (instrucciones).

Tenga en cuenta que escribir a PHSA o PHSB directamente sobrescribe ambos, el actual valor acumulado y cualquier potencial acumulación programada para el mismo momento en que se realice la escritura.

PRI

Denominador: Declara un Bloque de método Privado

((PUB PRI))
PRI *Name* ⟨(*Param* ⟨, *Param*⟩...)⟩ ⟨:*RValue*⟩ ⟨| *LocalVar* ⟨[*Count*]⟩⟩ ⟨,*LocalVar* ⟨[*Count*]⟩⟩...
SourceCodeStatements

- *Name* es el nombre deseado para el método Privado.
- *Param* nombre de parámetro (opcional). Los métodos pueden contener ceros o mas parámetros de coma-delimitado encerrados entre paréntesis. *Param* es globalmente única pero otros métodos pueden usar el mismo nombre de símbolo. Cada parámetro es esencialmente una variable long y se puede tratar como tal.
- *RValue* nombre para el valor de regreso del método (opcional). Esto se convierte en un alias a la variable RESULT incorporada del método. *RValue* es globalmente única pero otros métodos pueden usar el mismo nombre de símbolo. El *RValue* (y/o la variable RESULT) se inicia a cero (0) antes de cada llamada a método.
- *LocalVar* nombre de una variable local (opcional). *LocalVar* debe ser globalmente única, otros métodos pueden usar el mismo nombre de símbolo. Todas las variables locales son long (cuatro bytes) y se queden sin inicializar hasta la llamada al método. Los métodos pueden tener cero o variables locales de coma-delimitada.
- *Count* expresión opcional entre corchetes, indica que es un arreglo local variable, con *Count* números de elementos, comienza con un long. Cuando se hace referencia a estos elementos, comienzan con el elemento 0 y terminan con el elemento *Count*-1.
- *SourceCodeStatements* es una o mas líneas de código fuente ejecutable, indentado por al menos un espacio, que desarrolla la función del método.

Explicación

PRI es la declaración del Bloque del método Privado. Un método Privado es una sección de código fuente que desarrolla una función especifica y regresa un resultado. Este es uno de seis declaraciones especiales (CON, VAR, OBJ, PUB, PRI, y DAT) que proporcionan una estructura inherente al lenguaje Spin.

Cada objeto puede contener un numero de métodos privados (PRI) y públicos (PUB). Los Métodos Privados solo pueden accesarse desde el objeto y servir para desarrollar funciones protegidas y vitales para el objeto. Los métodos privados son como métodos públicos en todo sentido excepto que se declaran con PRI, en lugar de PUB, y no se accesan de afuera del objeto. Vea PUB, Pág. 186, para mas información.

PUB

Denominador: Declara un Bloque de método Publico

((PUB PRI))
PUB *Name* 〈 (*Param* 〈 , *Param*〉...) 〉 〈 : *RValue*〉 〈 | *LocalVar* 〈 [*Count*] 〉〉 〈 , *LocalVar* 〈 [*Count*] 〉〉...
SourceCodeStatements

- *Name* es el nombre deseado para el método publico.
- *Param* parámetro (opcional). Los métodos pueden contener ceros o parámetros de coma-delimitados entre paréntesis. *Param* es globalmente único pero lo pueden usar otros métodos. Cada parámetro es esencialmente una variable long y se puede tratar como tal.
- *RValue* nombre para el valor de regreso del método (opcional). Se convierte en alias de la variable RESULT generada del método. *RValue* es globalmente única pero otros métodos pueden usar el nombre del símbolo. El *RValue* (y/o variable RESULT) se inicia a cero (0) en cada llamada a método.
- *LocalVar* nombre de la variable local (opcional). *LocalVar* debe ser globalmente única pero otros métodos pueden usar el nombre del símbolo. Todas las variables locales son long (cuatro bytes) y se dejan sin inicializar hasta la llamada a método. Los métodos pueden contener cero o mas variables locales de coma delimitada.
- *Count* expresión opcional, encerrada en corchetes, que indica que es un arreglo variable local con *Count* numero de elementos; cada uno long. Cuando se hace referencia a estos elementos se comienza con el elemento 0 y se termina con el elemento *Count*-1.
- *SourceCodeStatements* es uno o mas líneas de código fuente ejecutable, indentado por al menos un espacio, que desarrolla la función del método.

Explicación

PUB la declaración del Bloque del método Publico. Un método publico es una sección de código fuente que desarrolla una función especifica que regresa un resultado. Este es uno de seis declaraciones especiales (CON, VAR, OBJ, PUB, PRI, y DAT) que proporciona una estructura inherente al lenguaje Spin

Cada objeto puede contener un numero de métodos públicos (PUB) y privados (PRI). Los públicos pueden accesarse desde afuera del objeto y sirve como interfase del objeto.

Las declaraciones PUB y PRI no regresan un valor por si mismas, pero representan siempre un valor de regreso cuando se llaman desde otro lado en el código.

Declaración de método Publico

Las declaraciones de método Publico comienzan con **PUB**, en la columna 1 de una línea, seguidas de un nombre único y un grupo de parámetros opcionales, un resultado variable y variables locales.

Ejemplo:

```
PUB Init
  <código de inicialización>

PUB MotorPos : Position
  Position := <código para recuperar posicion del motor>

PUB MoveMotor(Position, Speed) : Success | PosIndex
  <código que mueve el motor a posicion a una velocidad y regresa
True/False>
```

Este ejemplo contiene tres métodos públicos, `Init`, `MotorPos` y `MoveMotor`. El método `Init` no tiene parámetros y no declara un valor de regreso o variables locales. El método `MotorPos` no tiene parámetros pero declara un valor de regreso llamado `Position`. El método `MoveMotor` tiene dos parámetros, `Position` y `Speed`, un valor de regreso, `Success`, y una variable local, `PosIndex`.

Todos las instrucciones ejecutables que pertenecen al método **PUB** aparecen debajo de su declaración, indentadas por al menos un espacio.

El Valor de Regreso

Ya sea que una declaración **PUB** especifique un *RValue* o no, siempre hay un valor implicado de regreso que es por defecto cero (0). Hay un nombre pre-definido para este valor de regreso en cada método **PUB** llamado **RESULT**. En cualquier momento en un método, **RESULT** puede actualizarse como cualquier otra variable y al salir del método, el valor actual de **RESULT** se pasara de regreso a quien llamo. Además si se declara un **RESULT** para este método el nombre puede usarse con la variable integrada **RESULT**. Por ejemplo, el método `MotorPos` de arriba activa "`Position := …`" y podría usar también "`Result := …`" para el mismo efecto. Aun con esto, se considera buena practica dar un nombre descriptivo al valor de regreso (en la declaración `PUB`) para cualquier método para el cual el valor de regreso es importante. también es una buena practica dejar el valor de regreso sin nombre (en declaraciones **PUB**) para cualquier método en el que el valor de regreso no es importante.

Parámetros y Variables Locales

Los parámetros y variables locales son longs (cuatro bits). De hecho los parámetros son realmente solo variables que se inicializan a los valores correspondientes especificados por el llamador del método. Las variables locales, sin embargo, contienen datos aleatorios cada que se llama al método.

Todos los parámetros se pasan en un método por valor, no por referencia, así cualquier cambio a los parámetros en si no se reflejan fuera del método. Por ejemplo, si llamamos `MoveMotor` usando una variable `Pos` para el primer parámetro podría verse algo como esto:

```
Pos := 250
MoveMotor(Pos, 100)
```

Cuando el método `MoveMotor` se ejecuta, recibe el valor de `Pos` en su `Position` parámetro, y el valor 100 en su parámetro `Speed`. Dentro del método `MoveMotor`, puede cambiar `Position` y `Speed` en cualquier momento pero el valor de `Pos` (el llamador de la variable) se mantiene en 250.

Si una variable debe alterarse por una rutina, el llamador debe pasar la variable por referencia; lo que significa que debe pasar la dirección de la variable en vez del valor de la variable, y la rutina debe tratar ese parámetro como la dirección de la localidad de memoria en la cual operar. La dirección de la variable, o otros símbolos de base registro pueden recuperarse usando el operador Symbol Address, '**@**'. Por ejemplo,

```
Pos := 250
MoveMotor(@Pos, 100)
```

EL llamador pasa la dirección de `Pos` para el primer parámetro a `MoveMotor`. Lo que `MoveMotor` recibe en su parámetro `Position` es la dirección de la variable de; llamador `Pos`. La dirección es solo un numero, como cualquier otro, así que el método `MoveMotor` debe ser designada para tratarlo como dirección, en vez de un valor. El método `MoveMotor` entonces debe usar algo como esto:

```
PosIndex := LONG[Position]
```

...para recuperar el valor de la variable `Pos` del llamador y algo como:

```
LONG[Position] := <some expresión>
```

...para modificar la variable `Pos` del llamador en caso necesario

Al pasar un valor por referencia el operador Symbol Address se usa comúnmente cuando se proporciona una cadena variable a un método. Como las cadenas variables son solo arreglos de bytes no hay forma de pasarlos al método por valor; hacerlo resultaría en el método recibiendo solo el primer carácter. Aun si un método no necesita modificar una cadena u otro arreglo lógico, el arreglo en cuestión necesita pasarse como referencia porque hay múltiples elementos para ser accesados.

Optimizando Direccionamientos

En la aplicación Propeller compilada, los primeros ocho (8) longs que forman los parámetros, la variable RESULT y las variables locales son direccionadas usando un esquema optimizado de codificación. Esto significa accesar los primeros ocho longs (parámetros, RESULT, y variables locales) toma ligeramente menos tiempo que accesar el noveno o posterior long. Para optimizar la velocidad de ejecución, se asegura que todas las locales long usadas por las rutinas mas repetitivas del método pertenecen a las primeras ocho. Un mecanismo similar aplica para variables globales; ver la sección VAR, Pág. 217, para mayor información.

Saliendo de un método

Se sale de un método cuando la ejecución alcanza la ultima instrucción del método o cuando llega una instrucción RETURN o ABORT. Un método puede tener solo un punto de salida (la ultima instrucción ejecutable), o puede tener muchos puntos de salida (cualquier numero de instrucciones RETURN o ABORT además de la ultima instrucción ejecutable). La instrucción RETURN y ABORT pueden usarse también para activar la variable RESULT para salir; para mayor información ver RETURN, Pág. 200, y ABORT, Pág. 49.

QUIT

instrucción: Sale de un ciclo REPEAT inmediatamente.

```
((PUB   PRI))
  QUIT
```

Explicación

QUIT es uno de dos comandos (NEXT y QUIT) que afectan el ciclo REPEAT. QUIT ocasiona que un ciclo REPEAT termine inmediatamente.

Usando QUIT

QUIT se usa típicamente como un caso excepcional, en una instrucción condicional, en los ciclos REPEAT para terminar el ciclo prematuramente. Por ejemplo, asuma que DoMore y SystemOkay son métodos creados en otro lugar y que cada uno regresa valores Booleanos:

```
repeat while DoMore          'Repite mientras more hace
  !outa[0]                   'cambia estatus de luz
  <do something>             'hace alguna tarea
  if !SystemOkay
    quit                     'Si el sistema falla, sale
  <more code here>           'Desarrolla otras tareas
```

El código de arriba cambia una luz de estado en P0 y hace otras tareas mientras el método DoMore regresa TRUE. Sin embargo si el método SystemOkay regresa FALSE en medio del camino del ciclo, la instrucción IF ejecuta la instrucción QUIT lo cual ocasiona que el ciclo termine inmediatamente.

La instrucción QUIT solo puede usarse en un ciclo REPEAT; de otra forma marcara error.

REBOOT

instrucción: Reinicia el chip propeller.

((PUB PRI))
  REBOOT

Explicación

Este es un reinicio controlado por software, pero actúa como reinicio de hardware a través del pin E/S.

Use REBOOT si quiere reiniciar el chip Propeller a su estado inicial. Toda la base de hardware, encendido, arranque, retrasos así como el proceso de inicio se aplican si el Propeller se reinicio a través del pin RESn o un corte de corriente.

REPEAT

instrucción: Ejecuta un bloque de código repetitivamente.

((PUB PRI))
REPEAT ⟨*Count*⟩
→ *Statement(s)*

((PUB PRI))
REPEAT *Variable* FROM *Start* TO *Finish* ⟨STEP *Delta*⟩
→ *Statement(s)*

((PUB PRI))
REPEAT ((UNTIL WHILE)) *Condition(s)*
→ *Statement(s)*

((PUB PRI))
REPEAT
→ *Statement(s)*
((UNTIL WHILE)) *Condition(s)*

- *Count* es una expresión opcional indicando el numero finito de veces a ejecutar *Statement(s)*. Si se omite *Count* la sintaxis 1 crea un ciclo infinito hecho de *Statement(s)*.
- *Statement(s)* es un bloque opcional de una o mas líneas de código a ejecutar repetidamente. Omitir *Statement(s)* es raro, pero puede ser útil en la sintaxis 3 y 4 si *Condition(s)* alcanza los efectos necesarios.
- *Variable* es una variable, usualmente definida por usuario, que se repetirá de *Start* a *Finish*, opcionalmente por *Delta* unidades por repetición. *Variable* puede usarse en *Statement(s)* para determinar o utilizar la cuenta de repetición.
- *Start* es una expresión que determina el inicio del valor de *Variable* en la sintaxis 2. Si *Start* es menor que *Finish*, *Variable* se incrementara en cada repetición; de lo contrario se decrementara.
- *Finish* es una expresión que determina el valor final de *Variable* en la sintaxis 2. Si *Finish* es mayor que *Start*, *Variable* se incrementará en cada repetición; de lo contrario se decrementara.
- *Delta* es una expresión opcional que determina las unidades en las cuales se incrementa/decrementa *Variable* en cada repetición (sintaxis 2). Si se omite, *Variable* se incrementa/decrementa por 1 en cada repetición.

- *Condition(s)* una o mas expresiones Booleanas usadas en sintaxis 3 y 4 para continuar o terminar ciclos. Cuando precede **UNTIL**, *Condition(s)* termina el ciclo cuando es verdadero. Cuando precede **WHILE**, *Conditions(s)* termina el ciclo cuando es **FALSE**.

Explicación

REPEAT es una estructura de ciclo muy flexible para el código Spin. Puede usarse para crear cualquier tipo de ciclo, incluyendo infinito, finito, con o sin contador y condicional cero a muchos/uno a muchos ciclos.

Indentacion es Critica

IMPORTANTE: La indentacion es critica. El lenguaje spin se apoya en la indentacion (de uno o mas) en líneas seguidas de instrucciones condicionales para determinar si pertenecen a la instrucción o no. Para que la herramienta Propeller indique estos grupos lógicos de bloques de código en pantalla, puede presionar Ctrl + I para moverse a los indicadores de grupo. Si presiona Ctrl + I nuevamente deshabilitará esta característica. Vea la ayuda de la Herramienta Propeller para una lista completa de accesos rápidos.

Ciclos Infinitos (Sintaxis 1)

Honestamente cualquiera de las cuatro formas de **REPEAT** puede realizar ciclos infinitos, pero la forma usada con mayor frecuencia para este propósito es la sintaxis 1 sin el campo *Count*. Por ejemplo:

```
repeat                      'Repite sin final
  !outa[25]                 'Cambia P25
  waitcnt(2_000 + cnt)      'Pausa por 2,000 ciclos
```

Este código repite el `!outa[25]` y `waitcnt(2_000 + cnt)` sin terminar. Ambas líneas están indentadas de **REPEAT** así que pertenecen al ciclo **REPEAT**.

Como *Statement(s)* es realmente una parte opcional de **REPEAT**, la instrucción **REPEAT** por si misma puede usarse como un ciclo sin fin que no hace nada pero mantiene al cog activo. Esto puede ser intencional, pero algunas veces no es intencional o se debe a una indentacion inapropiada. Por ejemplo:

```
repeat                      'Repite sin final
!outa[25]                   'Cambia P25       <-- Eso nunca corre
```

El ejemplo anterior es erróneo, la ultima línea nuca se ejecutara porque el **REPEAT** sobre ella es un ciclo sin fin sin *Statement(s)*; no hay nada indentado inmediatamente debajo de el, así que el cog simplemente se queda en el ciclo sin fin en la línea **REPEAT** que no hace nada pero mantiene al cog activo y consumiendo potencia.

Ciclos Simples Finitos (Sintaxis 1)

La mayoría de los ciclos son finitos por naturaleza; ejecutan un numero limitado de iteraciones solamente. La forma mas sencilla en la sintaxis 1 con el campo *Count* incluido.

Por ejemplo:

```
repeat 10                   'Repite 10 veces
  !outa[25]                 'Cambia P25
byte[$7000]++               'Incrementa localidad RAM $7000
```

El código cambia P25 diez veces, luego incrementa el valor en la localidad RAM $7000

Observe que el campo *Count* puede ser una expresión numérica pero la expresión se evalúa solo una vez, la primera vez que se ingresa al ciclo. Esto significa que cualquier cambio a las variables de expresión en el ciclo no afectaran el numero de iteraciones del ciclo. El siguiente ejemplo asume que la variable `Index` se creo previamente.

```
Index := 10             'Activa ciclo repite 10 veces
repeat Index            'Repite Index veces
  !outa[25]             'Cambia P25
  Index := 20           'Cambia Index a 20
```

En el ejemplo anterior `Index` es 10 al ingresar al ciclo **REPEAT** la primera vez. Sin embargo, cada vez en el ciclo, `Index` s pone a 20, pero el ciclo continua ejecutándolo solo 10 veces.

Ciclos Finitos Contados (Sintaxis 2)

Frecuentemente es necesario contar las iteraciones en el ciclo para que el código del ciclo pueda desarrollar diferentes actividades basadas en ese conteo. La instrucción **REPEAT** hace esta parte simple con la sintaxis 2. El siguiente ejemplo asume que la variable `Index` se creo previamente.

```
repeat Index from 0 to 9    'Repite 10 veces
  byte[$7000][Index]++      'Incrementa localidades RAM $7000 a
$7009
```

Como en el primer ejemplo el código cicla 10 veces, pero en cada vez se ajusta la variable `Index`. La primera vez en el ciclo, `Index` será 0 (como se indica en “from 0”) y cada iteracion despues `Index` será 1 mayor que el valor anterior (como se indica en “to 9”): ..1, 2, 3…9. Después de la décima iteracion `Index` se incrementara a 10 y terminara el ciclo, ocasionando que se ejecute el siguiente ciclo **REPEAT** en la estructura, si es que existe. El código en el ciclo

usa Index como un Offset para afectar memoria, byte[$7000][Index]++; en este caso esta incrementando cada uno de los valores bytes en RAM $7000 a $7009 por 1, uno a la vez.

La instrucción **REPEAT** automáticamente determina si el rango sugerido por *Start* y *Finish* esta incrementando o decrementando. Como en el ejemplo se uso 0 a 9, el rango es de incremento; ajustando Index en +1 cada vez. Para obtener el conteo de reversa simplemente cambie los valores de *Start* y *Finish* como en::

```
  repeat Index from 9 to 0    'Repite 10 veces
    byte[$7000][Index]++      'Incrementa RAM $7009 bajando a $7000
```

Este ejemplo también cicla 10 veces pero contando Index de 9 bajando a 0; ajustando Index en -1 cada vez. El contenido del ciclo aun incrementa el valor en RAM, pero de las localidades $7009 bajando a $7000. despues de la décima iteracion Index es igual a -1.

Como los campos *Start* y *Finish* pueden ser expresiones, estos pueden contener variables. El siguiente ejemplo asume que S y F son variables creadas previamente.

```
  S := 0
  F := 9
  repeat 2                        'Repite dos veces
    repeat Index from S to F      'Repite 10 veces
      byte[$7000][Index]++        'Incrementa localidades RAM
$7000..$7009
    S := 9
    F := 0
```

El ejemplo anterior usa ciclos anidados. El ciclo externo (el primero) repite 2 veces. El ciclo interno repite con Index de S a F, los cuales previamente se activaron a 0 y 9, respectivamente. El ciclo interno incrementa el valor de localidades RAM $7000 a $7009, en ese orden, porque el ciclo interno esta contando iteraciones de 0 a 9. Entonces el ciclo interno termina (con Index activado a 10) y las dos ultimas líneas activan S a 9 y F a 0, efectivamente cambiando el valor de *Start* y *Finish*. Como esto esta aun dentro del ciclo externo, el ciclo externo ejecuta su contenido nuevamente (por segunda vez) ocasionando que el ciclo interno repita con Index de 9 bajando a 0. El ciclo interno incrementa valores en localidades RAM $7009 a $7000, en ese orden (inverso al anterior) y termina con Index igual a -1. Las dos ultimas líneas activan S y F nuevamente, pero el ciclo externo ya no repite una tercera vez.

Los ciclos **REPEAT** no están limitados a incrementar o decrementar en uno. Si la instrucción **REPEAT** utiliza la sintaxis opcional **STEP** *Delta*, incrementara o decrementara la *Variable* por la cantidad *Delta*. En la forma de sintaxis 2, **REPEAT** esta actualmente usando un valor *Delta* pero cuando el componente "**STEP** *Delta*" se omite usa +1 o -1 por defecto, dependiendo del rango

de *Start* y *Finish*. El siguiente ejemplo incluye el valor opcional *Delta* para incrementar de dos en dos.

```
  repeat Index from 0 to 8 step 2 'Repite 5 veces
    byte[$7000][Index]++          'Incrementa en par RAM de $7000 a
$7008
```

Aquí el ciclo **REPEAT** se ejecuta cinco veces con Index activado a 0, 2, 4, 6, y 8, respectivamente. Este código efectivamente incrementa cada dos localidades RAM (las localidades pares) de $7000 a $7008 y termina con Index igual a 10.

El campo Delta puede ser positivo o negativo, sin importar la naturaleza ascendente o descendente de los valores *Start* y *Finish*, y puede incluso ajustarse con el ciclo para alcanzar efectos interesantes. Por ejemplo, asumiendo Index y D son variables predefinidas, el siguiente código activa Index a la siguiente secuencia: 5, 6, 6, 5, 3.

```
  D := 2
  repeat Index from 5 to 10 step D
    --D
```

This loop started out with Index at 5 and a *Delta* (D) of +2. But each iteration of the loop decrements D by one, so at the end of iteration 1, Index = 5 and D = +1. Iteration 2 has Index = 6 and D = 0. Iteration 3 has Index = 6 and D = -1. Iteration 4 has Index = 5 and D = -2. Iteration 5 has Index = 3 and D = -3. The loop then terminates because Index plus *Delta* (3 + -3) is outside the range of *Start* to *Finish* (5 to 10).

Ciclos Condicionales (Sintaxis 3 y 4)

La forma final de **REPEAT**, la sintaxis 3 y 4 son ciclos finitos con condicionales de salida y tienen opciones flexibles que permiten el uso de lógica positiva o negativa y la creación de ciclos cero-a –muchos o uno-a-muchos iteraciones. Estas dos formas de **REPEAT** son usualmente referidas como ciclos “repite mientras” o “repite hasta”.

Vamos a ver la forma **REPEAT** descrita por la sintaxis 3. Consiste en la instrucción **REPEAT** seguida inmediatamente por **WHILE** o **UNTIL** luego *Condition(s)* y finalmente, en las líneas debajo el opcional *Statement(s)*. Como esta forma prueba *Condition(s)* al inicio de cada iteracion, crea un ciclo cero-a-muchos; el bloque *Statement(s)* se ejecutara cero o mas veces dependiendo de la *Condition(s)*. Por ejemplo, asuma que X es una variable que se creo anteriormente:

```
  X := 0
  repeat while X < 10         'Repite mientras X es menor que 10
    byte[$7000][X] := 0       'Incrementa valor RAM
```

```
  X++                         'Incrementa X
```

Este ejemplo primero activa X a 0, luego repite el ciclo mientras X es menor que 10. El código dentro del ciclo limpia RAM basado en X (empezando en la localidad $7000) e incrementa X. Después de la décima iteracion del ciclo X se iguala a 10, haciendo que la condición mientras X < 10 falsa y el ciclo termina.

Se dice que el ciclo usa lógica "positiva" porque continua mientras "**WHILE**" una condición es verdadera. podría ser escrito en lógica "negativa" usando **UNTIL**, en vez de while, como:

```
X := 0
repeat until X > 9            'Repite hasta que X es mayor que 9
  byte[$7000][X] := 0         'Incrementa valor RAM
  X++                         'Incrementa X
```

El ejemplo anterior se desarrolla de la misma forma que el anterior; pero el ciclo **REPEAT** usa lógica negativa porque continua "**UNTIL**" una condición es verdadera; Ejemplo: continua mientras una condición es falsa.

En cualquier ejemplo si X fuera igual o mayor a 10 antes de la primera iteracion del ciclo **REPEAT**, la condición hubiera ocasionado que el ciclo nunca se ejecutara, por esto se llama cero a muchos ciclos.

La forma **REPEAT** descrita por la sintaxis 4 es muy similar a la sintaxis 3, pero la condición se prueba al final de cada iteracion haciéndolo uno-a-muchos ciclos. Por ejemplo:

```
X := 0
repeat
  byte[$7000][X] := 0         'Incrementa valor RAM
  X++                         'Incrementa X
while X < 10                  'repite mientras X es menor que 10
```

Esto trabaja de la misma forma que el ejemplo anterior, ciclando 10 veces excepto que la condición no se prueba si no hasta el final de la iteracion. Sin embargo a diferencia del ejemplo previo aun si X es igual a 10 o mayor antes de la primera iteracion el ciclo correrá, por lo tanto se le llama uno a muchos ciclos.

Otras Opciones REPEAT

Hay otras dos instrucciones que afectan el comportamiento del ciclo **REPEAT**: **NEXT** y **QUIT**. Vea las instrucciones **NEXT** (Pág. 144) y **QUIT** (Pág. 190) para mayor información.

RESULT

Variable: El valor de regreso variable para métodos.

((PUB PRI))
  RESULT

Explicación

La variable RESULT es una variable local pre-definida para cada método PUB y PRI. RESULT contiene el valor de regreso del método; el valor que se pasa de regreso al llamador del método, cuando el método se termina.

Cuando un método publico o privado se llama, se genera una variable RESULT que se inicializa a cero (0). Si ese método no altera RESULT, o no llama a RETURN o ABORT con un valor especifico, entonces se regresara cero cuando se termine el método.

Usando RESULT

En el ejemplo anterior, el método DoSomething activa RESULT igual a 100 al final . El método Main llama DoSomething y active su variable local, Temp, iguala el resultado; así que cuando DoSomething sale, Temp será activado a 100

```
PUB Main | Temp
  Temp := DoSomething     'Llama DoSomething, Activa Temp para
regresar un valor

PUB DoSomething
  <hace algo aquí>
  result := 100           'Activa result a 100
```

También puede proporcionar un nombre alias para la variable del método RESULT con la finalidad de hacer mas claro o que regresa el método. Es muy recomendable ya que hace que la intención del método sea ms fácil d discernir. Por ejemplo:

```
PUB GetChar : Char
  <hace algo>
  Char := <recupera caracter>   'Activa Car (result) al caracter
```

El método de arriba GetChar, declara Char como alias para su variable RESULT construida; Ver PUB, Pág. 186 o PRI, Pág. 185, para mas información. El método GetChar desarrolla entonces algunas tareas para obtener un carácter y luego activa Char al valor del carácter recuperado.

podría usar también "`result := …`" para activar el valor de regreso ya que cualquier instrucción afecta el valor de regreso del método.

Ya sea la variable `RESULT` o el alias proporcionado para el, puede modificarse múltiples veces con el método antes de salir ya que ambos afectan `RESULT` y solo el ultimo valor de `RESULT` se usara durante la salida.

RETURN

instrucción: Sale de un método PUB/PRI con la opción de un regreso *Value*.

((PUB PRI))
RETURN ⟨*Value*⟩

Regresa: Ya sea el valor actual RESULT, o *Value* si se proporciona.

- *Value* es una expresión opcional cuyo valor se regresara desde el método PUB o PRI.

Explicación

RETURN es uno de dos instrucciones (ABORT y RETURN) que terminan una ejecución de método PUB o PRI. RETURN ocasiona un regreso desde un método PUB o PRI con estado normal; lo que significa que muestra la llamada de pila una vez y regresa al llamador de este método, entregando un valor en el proceso.

Cada método PUB o PRI implica un RETURN al final, pero RETURN puede también ingresarse manualmente en uno o mas lugares dentro del método para crear múltiples puntos de salida.

Cuando RETURN aparece sin el opcional *Value*, regresa el valor actual de la variable RESULT creada de PUB/PRI. Sin embargo si el campo *Value* se ingreso, el PUB o PRI regresa con el *Value*.

Acerca de la llamada de Pila

Cuando se llaman los métodos, simplemente por referencia desde otros métodos, debe existir un mecanismo en el cual almacenar un regreso una vez que el método se completo. Este mecanismo se llama “pila” pero aquí la usaremos como “llamada a pila”. Es simplemente memoria RAM utilizada para almacenar la dirección de regreso, valores de regreso, parámetros y resultados intermedios. Mientras mas y mas métodos se llamen, la llamada de pila crece. Mientras mas y mas métodos regresan (A través de RETURN o al alcanzar el final del método) la llamada de pila se hace mas corta. Esto se llama “pushing” en la pila o “popping” de la pila respectivamente.

La instrucción RETURN muestra el dato mas reciente de la llamada de pila para facilitar el regreso al inmediato llamador; el que directamente llamo al método del que regreso.

Usando RETURN

El siguiente ejemplo demuestra dos usos de **RETURN**. Asuma que DisplayDivByZeroError es un método definido en otra parte.

```
PUB Add(Num1, Num2)
  Result := Num1 + Num2                 'Suma Num1 + Num2
  return

PUB Divide(Dividend, Divisor)
  if Divisor == 0                       'Verifica si Divisor = 0
    DisplayDivByZeroError               'Si es así, despliega error
    return 0                            'y regresa con 0
  return Dividend / Divisor             'si no regresa cociente
```

El método Add active su variable incorporada **RESULT** igual a Num1 mas Num2, luego ejecuta **RETURN**. El **RETURN** hace que Add regrese el valor de **RESULT** al llamador. Observe que este **RETURN** no fue realmente requerido porque el compilador Propeller automáticamente lo pondrá al final de cualquier método que no tenga uno.

El método Divide verifica el valor Divisor. Si Divisor es igual a cero, llama al método DisplayDivByZeroError y luego ejecuta return 0, el cual inmediatamente hace que el método regrese con el valor 0. Sin embargo si Divisor no es igual a cero, se ejecuta return Dividend / Divisor, lo cual hace que regrese al método con el resultado de la división. Esto es un ejemplo donde el ultimo **RETURN** se uso para realizar el calculo y regresar el resultado, todo en un paso, en vez de afectar separadamente la variable integrada **RESULT** previamente.

ROUND

instrucción: Redondea una constante de punto flotante al entero mas cercano.

((CON VAR OBJ PUB PRI DAT))
ROUND (*FloatConstant*)

Regresa: El entero mas cercano al valor de la constante de punto flotante.

- *FloatConstant* es la expresión constante de punto flotante a ser redondeada al entero mas cercano.

Explicación

ROUND es una de tres instrucciones (FLOAT, ROUND y TRUNC) utilizadas para expresiones de punto flotante. La instrucción ROUND regresa un entero constante que es el entero mas cercano al valor de una expresión dada en punto flotante. Valores fraccionales de ½ (.5) o mayor se redondean al siguiente numero entero mientras fracciones menores se redondean hacia abajo.

Usando ROUND

ROUND puede usarse para redondear constantes de punto flotante arriba o abajo al siguiente valor entero. Observe que esto es para expresiones constantes en tiempo de compilación únicamente, no para expresiones variables en tiempo de ejecución, Ejemplo:

```
CON
  OneHalf = 0.5
  Smaller = 0.4999
  Rnd1    = round(OneHalf)
  Rnd2    = round(Smaller)
  Rnd3    = round(Smaller * 10.0) + 4
```

El código anterior crea dos constantes de punto flotante OneHalf y Smaller, igual a 0.5 y 0.4999, respectivamente. Las siguientes tres constantes, Rnd1, Rnd2 y Rnd3, son enteros constantes que se basan en OneHalf y Smaller usando la instrucción ROUND. Rnd1 = 1, Rnd2 = 0, y Rnd3 = 9.

Acerca de Constates de Punto Flotante

El compilador Propeller maneja constantes de punto flotante como números reales de precisión simple como se describe en el estándar IEEE-754. Los números reales de precisión

simple se almacenan en 32 bits, con 1-bit de signo, 8-bit exponentes, y 23-bit mantisa (la parte fraccional). Esto proporciona aproximadamente 7.2 dígitos decimales significativos.

Expresiones constantes de punto flotante pueden definirse y usarse para múltiples propósitos en tiempo de compilación, pero para tiempo de ejecución en operaciones de punto flotante los objetos FloatMath y FloatString proporcionan funciones matemáticas compatibles con números de precisión simple.

Ver Asignación Constante '=' en la sección de Operadores en Pág. 152, FLOAT en Pág. 111, y TRUNC en Pág. 213, así como los objetos FloatMath y FloatString para mayor información.

SPR

Registro: Registro de Propósito Especial Arreglo, proporciona acceso indirecto a los registros especiales del cog.

((PUB PRI))
SPR [*Index*]

Regresa: Valor en el registro de propósito especial en *Index*.

- *Index* es una expresión que especifica el Index (0-15) del registro de propósito especial a accesar (PAR a VSCL).

Explicación

SPR es un arreglo de 16 registros de propósito especial en el cog. El elemento 0 es el registro PAR y el elemento 15 es el registro VSCL. Ver Tabla 2-15 abajo. SPR proporciona un método indirecto de acceso a los registros de propósito especial del cog.

Tabla 2-15: Cog RAM Registros de Propósito Especial

Nombre	índice	Tipo	Descripción
PAR	0	Solo Lectura	parámetro Boot
CNT	1	Solo Lectura	Contador del sistema
INA	2	Solo Lectura	Estado de entradas P31 - P0
INB	3	Solo Lectura	Estado de entradas P63- P32[1]
OUTA	4	Lecto/Escritura	Estado de salidas P31 - P0
OUTB	5	Lecto/Escritura	Estado de salidas P63 – P32[1]
DIRA	6	Lecto/Escritura	Estado de direcciones P31 - P0
DIRB	7	Lecto/Escritura	Estado de direcciones P63 - P32[1]
CTRA	8	Lecto/Escritura	Control Contador A
CTRB	9	Lecto/Escritura	Control Contador B
FRQA	10	Lecto/Escritura	Contador de Frecuencia A
FRQB	11	Lecto/Escritura	Contador de Frecuencia B
PHSA	12	Lecto/Escritura	Contador Fase A
PHSB	13	Lecto/Escritura	Contador Fase B
VCFG	14	Lecto/Escritura	configuración de Video
VSCL	15	Lecto/Escritura	Escala de Video

Nota 1: Reservado para uso Futuro

Usando SPR

SPR puede usarse como cualquier arreglo de tamaño long. El siguiente ejemplo asume que Temp es una variable definida en otro lado:

```
spr[4] := %11001010         'Activa registro outa
Temp := spr[2]              'Obtiene valor ina
```

Este ejemplo activa el registro OUTA (índice 4 de SPR) a %11001010 y luego activa Temp igual al registro INA (índice 2 de SPR).

_STACK

Constante: Pre-definida, constante activable una sola vez para especificar el tamaño de un espacio de pila de una aplicación.

```
CON
  _STACK = expresión
```

- *expresión* es una expresión entera que indica el numero de longs a reservar por el espacio de pila.

Explicación

_STACK es una constante opcional pre-definida activable una vez que especifica el espacio de pila requerida por una aplicación. Este valor se agrega a _FREE si se especifica, para determinar el total de memoria libre/pila a reservar por una aplicación Propeller. Use _STACK una aplicación requiere un monto mínimo de espacio de pila par correr apropiadamente. Si el resultado de la aplicación compilada en demasiado largo para permitir el espacio de pila, un mensaje de error se desplegara. Por ejemplo:

```
CON
  _STACK   = 3000
```

La declaración _STACK en el bloque CON de arriba indica que la aplicación necesita al menos 3,000 longs de espacio de pila después de la compilación. Si el resultado de la aplicación compilada no tiene ese espacio disponible se genera un mensaje de error indicando por cuanto se excedió. Esta es una buena forma de prevenir compilaciones completas de la aplicación pero que fallaran en su ejecución por falta de memoria.

Observe que solo el objeto superior puede activar el valor de _STACK. Cualquier declaración en los objetos hijo será ignorada. El espacio de pila reservado por esta constante se usa por el la aplicación del cog principal para almacenar datos temporales como llamadas a pila, parámetros y resultados de expresiones intermedias.

STRCOMP

instrucción: Compara dos cadenas por igualdad.

((PUB PRI))
STRCOMP (*StringAddress1*, *StringAddress2*)

Regresa: TRUE si ambas cadenas son iguales, de lo contrario regresa FALSE.

- *StringAddress1* es una expresión especificando la dirección de inicio de la primer cadena a comparar.
- *StringAddress2* es una expresión especificando la dirección de inicio de la segunda cadena a comparar.

Explicación

STRCOMP es una de dos instrucciones (STRCOMP y STRSIZE) que recupera información acerca de una cadena. STRCOMP compara el contenido de la cadena en *StringAddress1* con el contenido de la cadena en *StringAddress2*, hasta el terminador cero de cada cadena, y regresa TRUE si ambas cadenas son equivalentes o FALSE de otra forma. Esta comparación es sensitiva a mayúsculas.

Usando STRCOMP

El siguiente ejemplo asume que PrintStr es un método creado en otro lado:

```
PUB Main
  if strcomp(@Str1, @Str2)
    PrintStr(string("Str1 and Str2 are equal"))
  else
    PrintStr(string("Str1 and Str2 are different"))

DAT
  Str1  byte  "Hello World", 0
  Str2  byte  "Testing.", 0
```

El ejemplo anterior tiene dos cadenas cero-terminadas en el bloque DAT, Str1 y Str2. El método Main llama STRCOMP para comparar los contenidos de cada cadena. Asumiendo que PrintStr es un método que despliega una cadena este ejemplo imprime “Str1 y Str2 are diferente” en el display

Cadenas Cero-Terminadas

La instrucción STRCOMP requiere que las cadenas a comparar sean cero-terminadas; un byte igual a 0 debe seguir inmediatamente a cada cadena. Esta practica es común y recomendada ya que la mayoría de los métodos de manejo de cadenas se soportan en terminaciones cero.

STRING

Instrucción: Declara una cadena constante en línea y obtiene su dirección.

((PUB PRI))
STRING (*StringExpression*)

Regresa: Dirección de la cadena constante en línea

- *StringExpression* es la expresión de cadena deseada a ser usada para propósitos temporales in línea.

Explicación

El bloque DAT se usa frecuentemente para crear cadenas o campos de cadenas que son reusables para varios propósitos, pero hay situaciones cuando una cadena se necesita para propósitos temporales como limpiar o uso único en un objeto. La instrucción STRING se crea para aquellas veces en las que se necesita una sola vez, cadenas cero-terminadas en memoria y regreso de direcciones de esa cadena.

Usando STRING

La instrucción STRING es muy buena para generar cadenas de un solo uso y pasar la dirección de esa cadena a otros métodos. Por ejemplo asumiendo que PrintStr es un método creado en otro lugar:

```
PrintStr(string(''Esta es una cadena de prueba.''))
```

El ejemplo de arriba usa la instrucción STRING para compilar una cadena. “Esta es una cadena de prueba.” En memoria y regresar la dirección de esa cadena como parámetro para el método ficticio PrintStr.

Si una cadena necesita usarse en mas de un lugar en el código es mejor definirla en el bloque DAT para que la dirección se pueda usar en múltiples ocasiones.

STRSIZE

instrucción: Obtiene el tamaño de una cadena

((PUB PRI))
STRSIZE (*StringAddress*)

Regresa: Tamaño (en bytes) de una cadena cero-terminada.

- *StringAddress* es una expresión especificando la dirección de inicio de la cadena a medir.

Explicación

STRSIZE es una de dos instrucciones (STRCOMP y STRSIZE) que recuperan información de la cadena.. STRSIZE mide la longitu de de la cadena de *StringAddress*, en bytes hasta , pero no incluyendo, el byte de cero-terminación.

Usando STRSIZE

El siguiente ejemplo asume que Print es un método creado en otro lugar:

```
PUB Main
  Print(strsize(@Str1))
  Print(strsize(@Str2))

DAT
  Str1  byte  "Hello World", 0
  Str2  byte  "Testing.", 0
```

El ejemplo tiene dos cadenas cero-terminadas en el bloque DAT, Str1 y Str2. El método Main llama STRSIZE para obtener la longitud de cada cadena. Asumiendo que Print es un método que despliega un valor, este ejemplo imprime 11 y 8 en el display.

Cadenas Cero-Terminadas

La instrucción STRSIZE requiere que la cadena a medir sea cero terminada; un byte igual a cero debe seguir inmediatamente a la cadena. Esta practica es frecuente y recomendada ya que la mayoría de los métodos que manejan cadenas se soportan en cero terminaciones.

Símbolos

Los símbolos en la Tabla 2-16 sirven uno o mas propósitos especiales en código Spin. Para símbolos ensamblador Propeller vea Símbolos, Pág. 375. Cada propósito se describe brevemente con referencias a otras secciones que lo describen directo o usa ejemplos.

Tabla 2-16: Símbolos

símbolo	Propósito(s)
%	Indicador Binario. Usado para indicar que un valor es expresado en binario (base-2). Ver Representaciones de Valores en Pág. 47.
%%	Indicador Cuaternario. Usado para indicar que un valor es expresado en cuaternario (base-4). Ver Representaciones de Valores en Pág. 47.
$	Indicador Hexadecimal. Usado para indicar que un valor es expresado en hexadecimal (base-16). Ver Representaciones de Valores en Pág. 47.
''	Indicador de cadena: se usa para comenzar y terminar una cadena de texto. Normalmente usada en bloques de objetos (Pág. 145), Bloques de datos (Pág. 102), o en bloques públicos/privados con la directiva STRING (Pág. 209).
@	Indicador de símbolo de dirección: se usa inmediatamente antes de un símbolo para indicar que la dirección de ese símbolo se usara, no es el valor de la localidad del símbolo. Ver Symbol Address '@' en Pág. 177.
@@	Indicador de símbolo Suma Dirección Objeto. Su usa inmediatamente despues de un símbolo para indicar el valor de que el símbolo debe sumarse a ka dirección base del objeto. Ver Object Address Plus Symbol '@@' en Pág. 177.
_	1) Delimitador. Se usa como delimitador de grupo en valores contantes (donde una coma ',' o punto '.' pede usarse como delimitador de un grupo de números) Ver Representaciones de Valores 2) en Pág. 47. 3) guión bajo: se usa como parte de un símbolo. Ver Reglas de en Pág. 47.
#	1) Referencia Constante Objeto: Se usa para referencias una constante de sub-objeto. Ver la sección de CON Alcance de Constantes, Pág. 92, y OBJ, Pág. 145. 2) Activación de listas: Se usa en bloques CON para activar el inicio de un grupo de símbolos. Ver la sección CON Enumeración (Sintaxis 2 y 3) en Pág. 90.
.	1) Referencia Objeto-método: Se usa para referenciar un método de sub-objetos. Ver OBJ, Pág. 145. 2) Punto decimal: Se usa para expresiones constantes de punto flotante. Ver CON, Pág. 87.

..	Indicador de rango: Indica un rango de una expresión a otra para instrucciones CASE o un índice de registro E/S. Ver OUTA, OUTB en Pág. 179, INA, INB en Pág. 122, y DIRA, DIRB en Pág. 107.

(La tabla continua en la siguiente pagina.)

Tabla 2-16: (continuación)

símbolo	Propósito(s)
:	1) Separador de valor de regreso: aparece antes de un valor de regreso simbólico en una declaración PUB o PRI. Ver PUB en pag 186, PRI en pag 185, y RESULT en pag 198. 2) Asignación de objeto: Aparece en una instrucción de referencia en un objeto, en el bloque OBJ. Ver OBJ, Pág. 145. 3) Separador de instrucción case: Aparece despues de las expresiones equivalentes en una estructura CASE. Ver CASE, Pág. 62.
|	1) Separador de variable local. Aparece antes de una lista de variables locales en una declaración PUB o PRI. Ver PUB, Pág. 186 y PRI, Pág. 185. 2) Bitwise OR: usado en expresiones, Ver Bitwise OR '|', '|=' en Pág. 169.
\	Trampa de Aborto: Aparece antes de una llamada a metodo que podría abortar potencialmente. Ver ABORT en Pág. 49.
,	Delimitador de Lista. Se usa para separar piezas en las listas. Ver LOOKUP, LOOKUPZ en Pág. 142, LOOKDOWN, LOOKDOWNZ en Pág. 140, y la sección DAT Declaración de Data(Sintaxis 1) en Pág. 103.
( )	Asignador de Lista de Parámetros: Se usan contenidos en parámetros de método. Ver PUB, Pág. 186 y PRI, Pág. 185.
[ ]	Asignador de Arreglos índice: Se usa para delimitar índices en arreglos variables o referencias de memoria principal. Ver VAR, Pág. 215; BYTE, Pág. 54; WORD, Pág. 232; y LONG, Pág. 132.
'	Asignador de comentario de código. Se usa para escribir comentarios de una línea (texto no compilado) para propósitos de vista de código. Ver "Usando la herramienta Propeller".
''	Asignador de comentario de documento: Se usa para ingresar comentarios de documento de línea sencilla con el propósito de documentar. Ver "Usando la herramienta Propeller".
{ }	Asignador de comentarios en línea/multi línea: Usado para ingresar comentarios de código para propósitos de documentación de código.
{{ }}	Asignador de comentarios en línea/multi línea: Se usa para comentarios del documento, para propósitos de documentación. Ver "Usando la herramienta Propeller".

TRUNC

Instrucción: Remueve o "acorta" la parte fraccional de una constante de punto flotante.

((CON VAR OBJ PUB PRI DAT))
TRUNC (*FloatConstant*)

Regresa: Un entero dado por una constante de punto flotante acortando al punto decimal.

- *FloatConstant* es la expresión constante de punto flotante a ser acortada a un entero.

Explicación

TRUNC es una de tres instrucciones (FLOAT, ROUND y TRUNC) usadas en expresiones constantes de punto flotante. TRUNC regresa una constante entera que es la expresión de punto flotante pero sin la parte fraccional.

Usando TRUNC

TRUNC se usa para recuperar la parte entera de una constante de punto flotante. Por ejemplo:

```
CON
  OneHalf = 0.5
  Bigger  = 1.4999
  Int1    = trunc(OneHalf)
  Int2    = trunc(Bigger)
  Int3    = trunc(Bigger * 10.0) + 4
```

El código de arriba crea dos constantes de punto flotante OneHalf y Bigger, igual a 0.5 y 1.4999, respectivamente. Las siguientes tres constantes, Int1, Int2 y Int3, son constantes enteras basadas en OneHalf y Bigger usando la instrucción TRUNC. Int1 = 0, Int2 = 1, y Int3 = 18.

Acerca de Constantes de Punto Flotante

El compilador Propeller maneja constantes de punto flotante como números reales de precisión simple por el estándar IEEE-754. Los números reales se almacenan en 32-bits, con 1 bit de signo, 8-bit de exponente, y 23-bit de mantisa (la parte fraccional). Esto proporciona aproximadamente 7.2 dígitos significativos decimales.

Las expresiones constantes de punto flotante pueden definirse y usarse para muchos propósitos, pero para operaciones en tiempo de ejecución los objetos FloatMath y FloatString proporciona funciones matemáticas compatibles con números de precisión simple. Ver

Asignación Constante '=' en la sección Operadores en Pág. 152, FLOAT en Pág. 111, y ROUND en Pág. 202, así como los objetos FloatMath y FloatString para mayor información.

VAR

Asignador: Declara un Bloque Variable

VAR
Size Symbol ⟨[*Count*]⟩ ⟨ ↳ *Size Symbol* ⟨ [*Count*]⟩⟩...

VAR
Size Symbol ⟨[*Count*]⟩ ⟨ , *Symbol* ⟨ [*Count*]⟩⟩...

- *Size* es el tamaño deseado de la variable, BYTE, WORD o LONG.
- *Symbol* es el nombre deseado de la variable.
- *Count* es una expresión opcional entre corchetes, que indica que es un arreglo variable con un numero *Count* de elementos; cada uno de tamaño byte, word o long. Cuando se hace referencia posterior a estos elementos comienzan con el elemento 0 y terminan con el elemento *Count*-1.

Explicación

VAR es la declaración del bloque variable. El bloque variable es una sección de código que declara símbolos globales variables. Este es una de seis declaraciones especiales (CON, VAR, OBJ, PUB, PRI, y DAT) que proporcionan una estructura inherente al código spin.

Declaraciones Variables (Sintaxis 1)

La forma mas común de las declaraciones de variables comienza con VAR en una línea seguida por una o mas declaraciones. VAR debe comenzar en una columna(la de la extrema izquierda) de la línea y las líneas que siguen deben estar indentadas por al menos un espacio.

```
VAR
  byte  Str[10]
  word  Code
  long  LargeNumber
```

Este ejemplo define Str como arreglo byte de 10 elementos, Code como word (dos bytes) y LargeNumber como long (cuatro bytes). Los métodos públicos y privados en el mismo objeto pueden hacer referencia a estas variables de diversas formas entre las cuales están:

```
PUB SomeMethod
  Code := 60000
  LargeNumber := Code * 250
  GetString(@Str)
  if Str[0] == "A"
    <more code here>
```

Observe que Code y LargeNumber son usadas directamente por expresiones. La Referencia Str en la lista del parámetro del método GetString se ve diferente; tiene un @, el símbolo operador de dirección precediéndolo. Esto es porque nuestro método ficticio GetString necesita escribir de regreso a la variable Str. Si dijimos GetString(Str), entonces el primer byte de Str, elemento 0, debería pasarse a GetString. Usando el símbolo operador de dirección, @, hacemos que la dirección de Str se pase en cambio a GetString; GetString puede usar esa dirección para escribir a los elementos de Str. Al ultimo, usamos Str[0] en la condición de una instrucción IF para ver si el primer byte es igual al caracter "A". Recuerde que el primer elemento de un arreglo es siempre cero (0).

Declaraciones Variables (Sintaxis 2)

Una variación de la sintaxis 1 permite variables de coma delimitada del mismo tamaño. EL siguiente es similar al ejemplo de arriba, pero declara dos words, Code y Index.

```
VAR
  byte  Str[10]
  word  Code, Index
  long  LargeNumber
```

Rango de Variables

El rango y naturaleza de cada tipo de variables es como sigue:

- BYTE – 8 bits (no signado); 0 a 255.
- WORD – 16 bits (no signado); 0 a 65,535.
- LONG – 32 bits (signado); -2,147,483,648 a +2,147,483,647.

Para mayor detalles respecto a los rangos y usos ver BYTE en Pág. 54, WORD en Pág. 232, y LONG en Pág. 132.

Organización de Variables

Durante la compilación de un objeto todas las declaraciones en sus bloques colectivos VAR se agrupan juntas por tipo. Las variables en RAM se acomodan con todos los long primero, seguidos por los Word y finalmente los bytes. Esto se hace para que el espacio de RAM se acomode eficientemente sin necesidad de espacios. Tenga en cuenta accesar las variables indirectamente basadas en su posición relativa a las otras cuando escribe el código.

Optimizando Direccionamientos

En la aplicación del Propeller compilado, las primeras ocho (8) variables globales se direccionan utilizando una codificación optimizada. Esto significa que a las primeras ocho variables long globales les toma un poco menos de tiempo para accesar que la novena variable o posterior. Las variables Word y byte no tienen este esquema optimizado. Para optimizar la velocidad de ejecución asegúrese que todas las variables globales usadas con mayor frecuencia por la aplicación pertenezcan a las primeras ocho long. Un mecanismo similar aplica a las variables locales, ver la sección PUB, Pág. 189, para mas información.

Alcance de las Variables

Variables simbólicas definidas en bloques VAR son globales al objeto en el cual se definieron pero no fuera de ese objeto. Esto significa que estas variables se pueden accesar directamente por cualquier método publico o privado en el objeto pero esos nombres no entraran en conflicto con símbolos definidos en objetos padres o hijos.

Note que los método públicos y privados tienen la facilidad de declarar sus propias variables locales. Ver PUB, Pág. 186, y PRI, Pág. 185.

Las variables globales no son accesibles fuera de un objeto a menos que la dirección se pase a, o de regreso a otro objeto a través de una llamada a método.

Alcance Extendido mas allá de un Cog Sencillo

El alcance de las variables globales no esta limitado a un cog sencillo. Un método publico y privado de un objeto naturalmente tiene acceso a sus variables globales sin importar en que cog están corriendo. Si el objeto inicia alguno de sus métodos en múltiples cogs, cada uno de esos métodos y por lo tanto los cogs en los que esta corriendo, tienen acceso a esas variables.

Esta característica permite a un objeto simple administrar aspectos de múltiples procesos en una forma centralizada. Por ejemplo un objeto sofisticado puede tomar ventaja de esto inmediatamente afectando los parámetros globales usados por cada código multiprocesador.

Por supuesto se debe tener cuidado de asegurar que los múltiples procesos en el mismo bloque no genere situaciones no deseadas. Vea LOCKNEW en Pág. 126 para ejemplos.

VCFG

Registro: Registro de configuración de Video

((PUB PRI))
VCFG

Regresa: Valor actual del registro de Configuración de Video, si usa una fuente variable.

Explicación

VCFG es uno de dos registros (VCFG y VSCL) que afecta el comportamiento del generador de Video del Cog. Cada cog tiene un modulo generador de video que facilita transmitir imágenes de video a un ritmo constante. EL registro VCFG contiene la configuración de ese generador de video, como se muestra en Tabla 2-17.

Tabla 2-17: Registro VCFG

Bits VCFG									
31	**30..29**	**28**	**27**	**26**	**25..23**	**22..12**	**11..9**	**8**	**7..0**
-	VMode	CMode	Chroma1	Chroma0	AuralSub	-	VGroup	-	VPins

En ensamblador Propeller los campos VMode a AuralSub pueden escribirse convenientemente usando la instrucción MOVI, el campo VGroupt puede escribirse con la instrucción MOVD, y el campo VPins puede escribirse con la instrucción MOVS.

VMode

The 2-bit VMode (video mode) field selects the type and orientation of video output, if any, according to Tabla 2-18.

Tabla 2-18: El Campo Video Mode

VMode	Video Mode
00	Desactivado, no genera video.
01	VGA modo; 8-bit salida paralela en VPins7:0
10	Modo Compuesto 1; transmite en VPins 7:4, banda base en VPins 3:0
11	Modo Compuesto 2; transmite en VPins 7:4, banda base en VPins 3:0

CMode

El campo CMode (modo color) selecciona dos o cuatro modos de color. 0 = modo color dos; datos de píxel 32 bits por 1 bit y solo usa colores 0 o 1. 1 = modo de color cuatro; datos de píxel 16 bits por 2 bits y usa colores de 0 a 3.

Chroma1

El bit Chroma1 (transmisión chroma) habilita o deshabilita chroma (color) en la señal de transmisión. 0 = desactivado, 1 = activado.

Chroma0

EL bit Chroma0 (banda base chroma) activa o desactiva chroma (color) en la señal de banda base. 0 = desactiva, 1 = activa.

AuralSub

El campo AuralSub (aural sub-carrier) selecciona la fuente del aura FM (audio) sub-acarreador de frecuencia modulada. La fuente es el PLLA de uno de los cogs, identificado como valor de AuralSub.

Tabla 2-19: El Campo AuralSub

AuralSub	Sub-Carrier Frecuencia Fuente
000	Cog 0's PLLA
001	Cog 1's PLLA
010	Cog 2's PLLA
011	Cog 3's PLLA
100	Cog 4's PLLA
101	Cog 5's PLLA
110	Cog 6's PLLA
111	Cog 7's PLLA

VGroup

El campo VGroup (grupo de pin de salida de video) selecciona en cual grupo de ocho pins E/S poner la salida de video.

Tabla 2-20: El Campo VGroup

VGroup	Grupo de Pins
000	Grupo 0: P7..P0
001	Grupo 1: P15..P8
010	Grupo 2: P23..P16
011	Grupo 3: P31..P24
100-111	<reservado para uso futuro >

VPins

El campo VPins (pins de salida de video) es una mascara aplicada a los pins de VGroup que indica en que pins esta la señal de salida de video.

Tabla 2-21: El Campo VPins

VPins	Efecto
00001111	Drive Video on lower 4 pins only; composite
11110000	Drive Video on upper 4 pins only; composite
11111111	Drive video on all 8 pins; VGA

Usando VCFG

VCFG puede leerse/escribirse como otros registros o variables pre-definidas. Por ejemplo:

```
VCFG := %0_10_1_0_1_000_00000000000_001_0_00001111
```

Esto activa el registro de configuración de video a video activo en modo compuesto 1 con 4 colores, banda base chroma (color) activada, en grupo de pins 1, los 4 pins bajos (los cuales son pins P11:8).

VSCL

Registro: Registro de Escala de Video

((PUB PRI))
VSCL

Regresa: Valor actual del registro de la escala de video del cog, si usa una fuente variable.

Explicación

VSCL es uno de dos registros (VCFG y VSCL) que afecta el comportamiento del generador de video del cog. Cada Cog tiene un generador de video que facilita la transmisión de imágenes de video a un rango constante. El registro VSCL activa el rango de generación de video.

Tabla 2-22: Registro VSCL

Bits VSCL		
31..20	**19..12**	**11..0**
–	PixelClocks	FrameClocks

PixelClocks

El campo PixelClocks 8-bits indica el numero de ciclos por píxel; el numero de ciclos que deberá pasar antes de que cada píxel se mueva por el modulo generador de video. Estos ciclos son el reloj PLLA y no los ciclos del reloj del sistema.

FrameClocks

El campo FrameClocks 12-bits indica el numero de ciclos por cuadro; el numero de ciclos que deberán pasar antes de que el modulo generador de video cambie el cuadro. Estos ciclos son del PLLA no los del reloj de sistema. Un cuadro es un long de datos píxel (entregado a través de la instrucción WAITVID). Como el dato de píxel es ya sea de16 por 2 bits, o 32 bits por 1 bit (significa 16 pixeles de ancho con 4 colores, o 32 pixeles de ancho con 2 colores, respectivamente), el FrameClocks es normalmente 16 o 32 veces el valor de PixelClocks.

Usando VSCL

VSCL se puede leer/escribir como en otros registros o variables pre-definidas. Por ejemplo:

```
VSCL := %000000000000_10100000_101000000000
```

Esto activa el registro de la escala de video para 160 PixelClocks y 2,560 FrameClocks (en una configuración 16-pixeles por 2-bit cuadro color). Por supuesto el rango actual al cual los ciclos de pixeles dependen de la frecuencia del PLLA en combinación con este factor.

WAITCNT

instrucción: Detiene la ejecución de un cog momentáneamente.

((PUB PRI))
  WAITCNT (*Value*)

- *Value* es el valor del contador del sistema 32-bits que se desea esperar.

Explicación

WAITCNT, "Wait for System Counter," es uno de cuatro instrucciones de espera (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener la ejecución de un cog hasta que una condición se cumple. WAITCNT detiene el cog hasta que el contador del Sistema Global iguala *Value*.

Cuando se ejecuta, WAITCNT activa un hardware especial "wait" en el cog que previene que el reloj del sistema ejecute mas código en el cog hasta el momento en el que el reloj del sistema es igual a *Value*. El hardware wait verifica el sistema contador cada ciclo del reloj del sistema y el consumo de potencia del cog se reduce aproximadamente en 7/8 durante este tiempo. En aplicaciones normales WAITCNT puede usarse estratégicamente para reducir consumo de potencia en cualquier parte del programa donde se espera tiempo para eventos de bajo ancho de banda.

Hay dos tipos de pausas WAITCNT puede usarse para: pausas fijas y pausas sincronizadas, ambas se explican a continuación:

Pausas Fijas

Las pausas fijas son aquellas que no están relacionadas a un punto especifico en el tiempo y solo sirven para propósitos de detener la ejecución por un monto fijo de tiempo. Una pausa fija se puede usar por ejemplo para esperar 10 milisegundos despues de ocurrido un evento, antes d proceder con otra acción; Por ejemplo:

```
CON
  _clkfreq = xtal1              'Activa el cristal lento
  _xinfreq = 5_000_000          'Usa cristal de 5 MHz de precisión

   repeat
    !outa[0]                    'Cambia pin 0
    waitcnt(50_000 + cnt)       'Espera 10 ms
```

Este código cambia el estado del pin E/S pin 0 y espera 50,000 ciclos de reloj de sistema antes de repetir el ciclo nuevamente. Recuerde que el parámetro *Value* debe ser el valor deseado de 32-bit a igualar contra el valor del valor del Reloj del Sistema. Como el reloj del sistema es un recurso global que cambia cada ciclo de reloj, para retrasar por un cierto numero de ciclos desde "ahora" se necesita un valor que se agregue al valor actual del contador del sistema. El `cnt` en "`50_000 + cnt`" es el registro variable del contador del sistema; regresa el valor del contador del sistema en ese momento a tiempo. así nuestro código dice esperar 50,000 ciclos mas el valor actual del contador del sistema; Ejemplo: espera 50,000 ciclos "desde este momento". Asumiendo que un cristal externo de 5MHz esta siendo usado, 50,000 ciclos es alrededor de 10ms ($1/100^{h}$ segundo) de tiempo.

IMPORTANTE: Como **WAITCNT** detiene el cog hasta que el contador del sistema iguala al valor dado, se debe tener cuidado de asegurar que el valor dado no se ha pasado por el contador del sistema. Si el contador del sistema ya paso el valor dado antes de que se active el hardware de espera entonces el cog aparecerá como detenido permanentemente cuando, de hecho, esta esperando que el contador exceda los 32 bits y regrese nuevamente al valor dado. Aun a 80 MHz, toma arriba de 53 segundos para un contador de sistema darle la vuelta!

Relacionado a esto, cuando use **WAITCNT** en código Spin como se muestra arriba, asegúrese de escribirla expresión *Value* de la misma forma que lo hicimos: en la forma "`Offset + cnt`" Y no "`cnt + Offset`." Esto es porque el interprete spin evaluara la expresión de izquierda a derecha, y cada evaluación intermedia en una expresión tomo tiempo desarrollarse. Si `cnt` se coloca al inicio de la expresión, el contador del sistema lo leerá primero y despues el resto de la expresión será evaluada, tomando un desconocido monto de ciclos y haciendo nuestro valor `cnt` muy viejo para el momento en que el resultado final se calcula. Sin embargo teniendo `cnt` como el ultimo valor en la expresión **WAITCNT** se asegura un monto fijo de tiempo extra (ciclos) entre la lectura del contador del sistema y la activación del hardware de espera. De hecho el interprete toma 381 ciclos del tiempo extra final cuando la instrucción se escribe de la forma `waitcnt(Offset + cnt)`. **Esto significa que el valor de `Offset` debe ser siempre al menos 381 para evitar inesperados retrasos largos.**

Pausas Sincronizadas

Sincronizar los retrasos son aquellos que están relacionados directamente a un punto especifico en el tiempo, un tiempo "base", y sirve al propósito de "alineación de tiempo" de eventos futuros relativos a ese punto. Una pausa sincronizada, por ejemplo, puede usarse para sacar o ingresar una señal en un intervalo especifico, sin importar el monto desconocido de tiempo asociado con el código mismo. Para entender como esta situación es diferente a una pausa fija veamos el siguiente ejemplo en el diagrama:

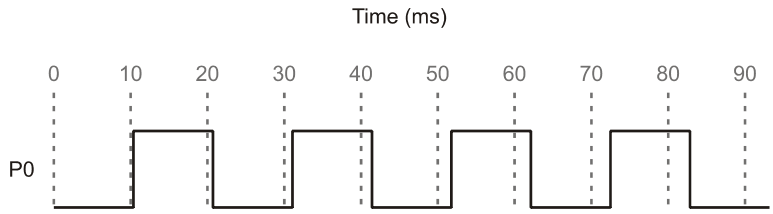


Figura 2-3: Tiempo en Pausa Fija

La Figura 2-3 muestra la salida de nuestro ejemplo anterior, el ejemplo de pausa fija. Observe como el P0 de los pins E/S cambia cada 10 milisegundos, pero ¿no exactamente? De hecho hay un error acumulativo que se hace sucesivo al estado de los cambios y se va quedando fuera d sincronía en relación a nuestro inicio, 0 ms. La pausa es 10 ms en longitud, pero el error ocurre porque la pausa no compensa la longitud del resto del ciclo. Las instrucciones `repeat, !outa[0]` y **WAITCNT** toma un poco de tiempo en ejecutarse, y el y ese tiempo extra se suma a la pausa de los 10 ms que se especifico en **WAITCNT**.

Usando **WAITCNT** de una forma diferente, para una pausa sincronizada, eliminaremos este error. El siguiente ejemplo asume que estamos usando un cristal externo de 5 MHz.

```
CON
  _clkfreq = xtal1                'Activa a cristal lento
  _xinfreq = 5_000_000            'Usa un cristal de 5MHz de precision

PUB Toggle | Time
  Time := cnt                     'Obtiene el valor actual del contador
  repeat
    waitcnt(Time += 50_000)       'Espera 10 ms
    !outa[0]                      'Cambia Pin 0
```

Este código primero recupera el valor del contador del sistema, `Time := cnt`, luego inicia el ciclo `repeat` donde espera por el contador del sistema para alcanzar `Time` + 50,000, cambia el estado del pin E/S Pin P0 y repite el ciclo. La instrucción `Time += 50_000` es actualmente una instrucción de asignación; Suma el valor de `Time` a 50,000, almacena ese resultado de regreso a `Time` y luego ejecuta la instrucción **WAITCNT** usando ese resultado. Observe que hemos recuperado el valor del contador del sistema solo una vez, al inicio del ejemplo; que es nuestro tiempo base. Luego esperamos por el contador del sistema para igualar esa base original de tiempo mas 50,000 y desarrollar las acciones en el ciclo. Cada iteracion sucesiva a través del ciclo esperara por el contador del sistema para igualar a otro múltiplo de 50,000 de

la base de tiempo. Este método compensa automáticamente el tiempo extra que consumen las instrucciones del ciclo: `repeat`, !outa[0] y `waitcnt`. El resultado se ve como en la Figura 2-4.

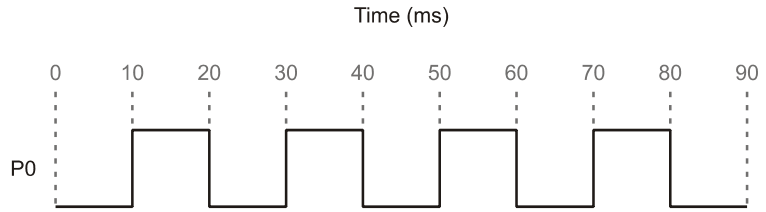


Figura 2-4: Pausa Sincronizada de Tiempo

Usando el método sincronizado de pausa, nuestra señal de salida esta siempre alineada perfectamente al tiempo base mas un múltiplo de nuestro intervalo. Esto trabajara de acuerdo a la precisión del tiempo base (un cristal externo) y que el tiempo extra en el ciclo no exceda el tiempo del intervalo mismo. Observe que esperamos, con **WAITCNT**, antes del primer cambio así que el tiempo entre el primer cambio y el segundo iguala al resto.

Calculando Tiempo

Un objeto puede retrasar un monto especifico de tiempo aun si la aplicación cambia la frecuencia del Reloj del Sistema ocasionalmente. Para hacer esto, use **WAITCNT** combinado con una expresión que incluye la frecuencia actual del reloj del sistema (**CLKFREQ**). por ejemplo, sin saber cual es la frecuencia actual del reloj será para la aplicación usando el objeto, la siguiente línea puede usarse para retrasar el cog por 1 milisegundo; siempre y cuando la frecuencia del reloj sea suficientemente rápida.

```
waitcnt(clkfreq / 1000 + cnt)          'retrasa el cog 1 milisegundo
```

Para mas información ver **CLKFREQ** en Pág. 66.

WAITPEQ

instrucción: Detiene la ejecución de un cog hasta que el pin E/S es igual a un estado dado.

((PUB PRI))
WAITPEQ (*State*, *Mask*, *Port*)

- *State* es el estado lógico de los pins para comparar. Es un valor de 32-bits que indica el estado alto o bajo de hasta 32 pins E/S. *State* se compara contra (INA y *Mask*), o (INB y *Mask*), dependiendo de *Port*.
- *Mask* es el pin a monitorear. *Mask* es un valor de 32-bits que contiene bits altos (1) para cada pin E/S que se desea monitorear; un bit bajo (0) indica los pins que deben ignorarse. Mask es un bitwised-AND con el estado de entrada del puerto de 32 bits y el valor resultante se compara contra el valor de *State*.
- *Port* es un valor de 1 bit que indica el puerto E/S a monitorear; 0 = Puerto A, 1 = Puerto B. En el chip Propeller solo existe Puerto A actualmente.

Explicación

WAITPEQ, "Wait for Pin(s) to Equal," es una de cuatro instrucciones (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener la ejecución de un cog hasta que una condición se cumple. WAITPEQ detiene el cog hasta que el valor del estado de los pins E/S de *Port*, el bitwised-AND con *Mask*, coincide con *State*.

Cuando se ejecuta WAITPEQ se activa un hardware "wait" especial en el cog que previene al reloj del sistema de ejecutar mas código en el cog hasta el momento que l pin designado o grupo de pins iguala al estado indicado. El hardware wait verifica lo pins E/S cada ciclo del reloj del sistema y el consumo de potencia se reduce aproximadamente 7/8 durante este tiempo.

Usando WAITPEQ

WAITPEQ es una forma de sincronizar el código a eventos externos. Por ejemplo:

```
waitpeq(%0100, %1100, 0)       'Espera que P3 y P2 sean alto y bajo
outa[0] := 1                   'Activa P0 alto
```

El código de arriba detiene el cog hasta que el pin E/S 3 esta en bajo y el pin 2 en alto, despues activa el pin 0 en alto.

Usando Diferentes números de Pin

En los Objetos Propeller es necesario frecuentemente monitorear un pin sencillo el cual esta especificado fuera del objeto mismo. Una forma sencilla d trasladar ese numero de pin al valor apropiado de 32 bits *State* y *Mask* es usando el operador Bitwise Decode "|<" (Ver 164 para mas información). Por ejemplo, si el numero de pin se especifico por la variable Pin, y necesitamos esperar hasta que el pin este en alto, podríamos usar el siguiente código:

```
waitpeq(|< Pin, |< Pin, 0)  'Espera para Pin a que vaya en alto
```

Los parámetros *Mask*, |< Pin, evalúan a un valor long donde solo un bit esta en alto; el bit que corresponde al numero de pin dado por Pin.

Esperando por Transiciones

Si necesitamos esperar para una transición de un estado a otro (alto a bajo por ejemplo) podríamos usar el siguiente código:

```
waitpeq(%100000, |< 5, 0)   'Espera a que el pin 5 este en alto
waitpeq(%000000, |< 5, 0)   'Luego espera que el pin 5 este en bajo
```

Este ejemplo primero espera a que pin P5 vaya a alto, luego espera a que vaya a bajo; un cambio alto a bajo. Si hemos usado la segunda línea de código sin la primera el cog no habría pausado en ningún momento P5 porque hubiera estado en bajo desde el comienzo.

WAITPNE

instrucción: Detiene la ejecución del cog hasta que un pin E/S no es igual al estado designado.

((PUB PRI))
WAITPNE (*State*, *Mask*, *Port*)

- *State* es el estado lógico a usar para comparar los pins. Es un valor de 32 bits que indica el estado alo o bajo de hasta 32 pins E/S. *State* se compara contra (INA y *Mask*), o (INB y *Mask*), dependiendo de *Port*.
- *Mask* es el pin que se desea monitorear. *Mask* es un valor de 32 bits que contiene un bit alto (1) por cada pin E/S a monitorear; un pin bajo (0) indica que los pins se deben ignorar. Mask es un bitwised-AND con el estado de los 32 bits de entrada y el valor resultante se compara contra el valor de *State*.
- *Port* es un valor de 1 bit que indica el puerto a monitorear; 0 = Puerto A, 1 = Puerto B. El chip Propeller solo tiene actualmente el Puerto A.

Explicación

WAITPNE, "Wait for Pin(s) to Not Equal," es uno de cuatro comandos de espera (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) utilizados para detener la ejecución de un cog hasta que una condición se cumpla. WAITPNE es la forma complementario de WAITPEQ; detiene el cog hasta que el valor del estado de los pins E/S y el bitwised-AND con *Mask*, no coincidan con *State*.

Cuando se ejecuta WAITPNE activa un hardware especial "wait" en el cog que previene que el reloj del sistema continúe realizando tareas de ejecución de código en el cog hasta el momento que el pin o grupo de pins no sean igual al estado asignado. El hardware wait verifica los pins E/S en cada ciclo del reloj del sistema y el consumo de potencia del cog se reduce aproximadamente 7/8 durante este tiempo.

Usando WAITPNE

WAITPNE es una excelente forma de sincronizar el código con eventos externos. Por ejemplo:

```
waitpeq(%0100, %1100, 0) 'Espera que P3 y P2 estén a bajo y alto
waitpne(%0100, %1100, 0) 'Espera que P3 y P2 no coincidan
outa[0] := 1             'Activa P0 a alto
```

El código anterior detiene el cog hasta que P3 esta en bajo y P2 en alto, despues detiene el cog otra vez hasta que uno o ambos pins cambian de estado y entonces activa P0 en alto.

WAITVID

instrucción: Detiene la ejecución de un cog hasta que el generador de video esta disponible para tomar datos de pixeles.

((PUB PRI))
WAITVID (*Colors*, *Pixels*)

- *colores* es un long que contiene valores de color de cuatro bytes, cada uno describe los cuatro posibles colores del patrón de pixeles en *Pixels*.
- *Pixels* es el patrón siguiente de 16-pixeles por 2-bit (o 32-pixeles por 1-bit).

Explicación

WAITVID, "Wait for Video Generator," es uno de cuatro comandos de espera (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) usados para detener la ejecución de un cog hasta que se cumpla una condición. WAITVID detiene el cog hasta que el hardware de su generador de video esta listo para los siguientes datos de pixels, entonces el generador de video acepta los datos y el cog continua la ejecución con la siguiente línea de código.

Cuando se ejecuta WAITVID se activa un hardware especial "wait" en el cog que previene que el reloj del sistema continúe la ejecución de código hasta el momento en el que el generador de video esta listo. El hardware verifica el estado del generador cada ciclo del reloj del sistema y el consumo de potencia en el cog se reduce significantemente durante este tiempo.

Usando WAITVID

WAITVID es solo un mecanismo de entrega para datos al hardware del generador de video del cog. Como el generador de video trabaja independientemente del mismo cog, los dos deben estar sincronizados en cada momento que se necesita desplegar datos en pantalla. la frecuencia en la cual ocurre depende del tipo de pantalla y los parámetros correspondientes del generador de video, pero en cada caso, el cog debe tener nuevos datos disponibles al momento que el generador de video esta listo para ello. El cog usa la instrucción WAITVID para esperar por el tiempo adecuado y luego "suelta" estos datos al generador de video.

El parámetro *colores* es un valor de 32 bits que contiene cuatro valores de color de 8 bits (para el modo color-4) o dos valores de color en los 16 bits mas bajos (para modo de 2 colores). Para VGA, cada valor de color arriba de 6 bits sus componentes son 2-bit rojo, 2-bit verde, y 2-bit azul y describen el color deseado; los 2 bits mas bajos "no importan". Cada uno de los valores corresponde a uno de los cuatro posibles colores por pixeles de 2 bits (cuando

Pixels se usa como patrón de pixeles 16x2 bits) o como uno de dos posibles colores por pixeles de 1 bit (cuando *Pixels* se usa a 32x1).

Pixels describe el patrón de pixeles a desplegar, ya sea 16 pixeles o 32 pixeles dependiendo de la profundidad de color en la configuración del generador de video.

revise los objetos TV y VGA para ejemplos de como usar WAITVID.

Asegúrese de iniciar el modulo generador de video del cog y el contador A antes de ejecutar la instrucción WAITVID o esperara por siempre. ver VCFG en Pág. 218, VSCL en Pág. 221, y CTRA, CTRB en Pág. 98.

WORD

Asignador: Declara un símbolo de tamaño word, datos alineados word, o lee/escribe word de memoria principal.

```
VAR
   WORD Symbol {[Count]}
```

```
DAT
   {Symbol} WORD Data {[Count]}
```

```
((PUB   PRI))
   WORD [BaseAddress] {[Offset]}
```

```
((PUB   PRI))
   Symbol.WORD {[Offset]}
```

- *Symbol* es el nombre deseado para la variable (sintaxis 1)o bloque de datos (sintaxis 2) o es el nombre existente de la variable (sintaxis 4).
- *Count* es una expresión opcional indicando el numero de elementos tamaño word para *Symbol* (Sintaxis 1),o el numero de entradas tamaño word de *Data* (Sintaxis 2) a almacenar en una tabla de datos .
- *Data* es una expresión constante o lista de coma separada de expresiones constantes.
- *BaseAddress* es una expresión que describe la dirección de la expresión alineada word de memoria principal a leer o escribir. Si *Offset* se omite, *BaseAddress* es la dirección actual en donde operar. Si *Offset* se especifica, *BaseAddress* + *Offset* * 2 es la dirección actual en donde operar.
- *Offset* es una expresión opcional indicando el Offset de *BaseAddress* en donde operar, o el Offset del word 0 de *Symbol*. *Offset* es una unidad de words.

Explicación

WORD es una de tres declaraciones multipropósito (BYTE, WORD, y LONG) que declara u opera en memoria. WORD puede usarse para:

1) declarar símbolo word (16-bit), o un arreglo simbólico múltiple en un bloque VAR, o
2) declarar un word alineado, y/o tamaño word, datos en un bloque DAT, o
3) leer/escribir word de memoria principal en una dirección base con un Offset opcional
4) accesar un word en una variable tamaño long.

Rango de Word

La memoria de tamaño word (16 bits) puede contener un valor que es una de 2^{16} combinaciones posibles de bits (ejemplo: una de 65,536). Esto le da a los valores de tamaño word un rango de 65,535. Como el lenguaje spin desarrolla todas las operaciones matemáticas usando matemáticas de 32 bits con signo, cualquier valor de tamaño word se tratara como tamaño long positivo. Sin embargo el valor numérico actual contenido en un word esta sujeto a como una computadora y usuario lo interprete. Por ejemplo, puede escoger usar el operador a Signo-Exten (**~~**), en la Pág. 161, en una expresión Spin para convertir un valor word que interpreta como "signado" (-32,768 a +32,767) para un valor long con signo.

Declaración Variable Word (Sintaxis 1)

En bloques **VAR** , la sintaxis 1 de **WORD** se usa para declarar variables simbólicas, globales que son de tamaño word o un arreglo de words, Por ejemplo:

```
VAR
  word  Temp                  'Temp es un word (2 bytes)
  word  List[25]              'List es un arreglo word
```

El ejemplo de arriba declara dos variables (símbolos), `Temp` y `List`. `Temp` es solo una variable simple de tamaño word. la línea debajo de la declaración `Temp` usa el campo opcional *Count* para crear un arreglo de 25 elementos variables de tamaño word llamado `List`. Ambos `Temp` y `List` se pueden accesar desde cualquier método **PUB** o **PRI** en el mismo objeto donde el bloque **VAR** se declaro; estos son globales al objeto. Un ejemplo de esto se muestra abajo.

```
PUB SomeMethod
  Temp := 25_000              'Activa Temp a 25,000
  List[0] := 500              'Activa el primer elemento de List a 500
  List[1] := 9_000            'Activa segundo elemento de List a 9,000
  List[24] := 60_000          'Activa ultimo elemento de List a 60,000
```

Para mayor información acerca del uso de **WORD** de esta forma, vea la sección **VAR** de las Declaraciones Variables (Sintaxis 1) en Pág. 215, y tenga en cuenta que **WORD** se usa para el campo *Size* en esa descripción.

Declaración de Datos Word (Sintaxis 2)

En los bloques DAT la sintaxis 2 de WORD se usa para declarar datos alineados word o tamaño word que se compilan como valores constantes en memoria principal. Los bloques DAT permiten esta declaración para tener un símbolo opcional que lo preceda, el cual puede usarse para referencia posterior. Ver DAT, Pág. 102. Por ejemplo:

```
DAT
  MyData  word  640, $AAAA, 5_500       'Dato tamaño/alineado Word
  MyList  byte  word $FF99, word 1_000  'Alineado byte/ tamaño word
```

El ejemplo anterior declara dos símbolos de datos, MyData y MyList. MyData apunta al inicio de los datos de alineación word y tamaño word en memoria principal. El valor de MyData en memoria principal son 640, $AAAA y 5,500, respectivamente. MyList usa un bloque especial DAT con una sintaxis de WORD que genera un dato alineado byte pero tamaño word en memoria principal. Los valores de MyList, en memoria principal, son $FF99 y 1,000, respectivamente. Cuando se acceso un byte a la vez, MyList contiene $99, $FF, 232 y 3 ya que los datos están almacenados en formato little-endian.

Este dato se compila en el objeto resultando en la aplicación como parte de la sección de código ejecutable y puede accesarse usando la forma leer/escribir, sintaxis 3 de WORD (ver abajo). Para mas información acerca de usar WORD de esta forma, véala sección DAT Declaración de Data(Sintaxis 1) en Pág. 103 y tenga en cuenta que WORD se usa para el campo *Size* en esa descripción.

Los datos pueden repetirse usando el campo opcional *Count*. Por ejemplo:

```
DAT
  MyData      word  640, $AAAA[4], 5_500
```

El ejemplo de arriba declara una tabla de datos alineado word, tamaño word llamada MyData, que consiste de los siguientes seis valores: 640, $AAAA, $AAAA, $AAAA, $AAAA, 5500. Hay cuatro repeticiones de $AAAA debido al [4] en la declaración inmediata.

Lectura/Escritura de Words en Memoria Principal (Sintaxis 3)

En bloques PUB y PRI la sintaxis 3 de WORD se usa para leer o escribir valores tamaño word de memoria principal. Esto se hace escribiendo expresiones que se refieren a memoria principal usando la forma: word[*BaseAddress*][*Offset*]. Aquí un ejemplo.

```
PUB MemTest | Temp
  Temp := word[@MyData][1]              'Lee valor Word
  word[@MyList][0] := Temp + $0123      'Escribe valor Word
```

```
DAT
  MyData  word  640, $AAAA, 5_500       'Dato alineado/tamaño Word
  MyList  byte  word $FF99, word 1_000  'Tamaño Byte/Alineado Word
```

En este ejemplo el bloque **DAT** (parte baja del código) coloca sus datos de memoria como muestra la Figura 2-5. El primer elemento de `MyData` se coloca en $18. El ultimo elemento de `MyData` se coloca en $1C, con el primer elemento de `MyList` seguido inmediatamente a $1E. Observe que la dirección de inicio ($18) es arbitraria y probablemente cambiara si el código se modifica o si el objeto mismo se incluye en otra aplicación.

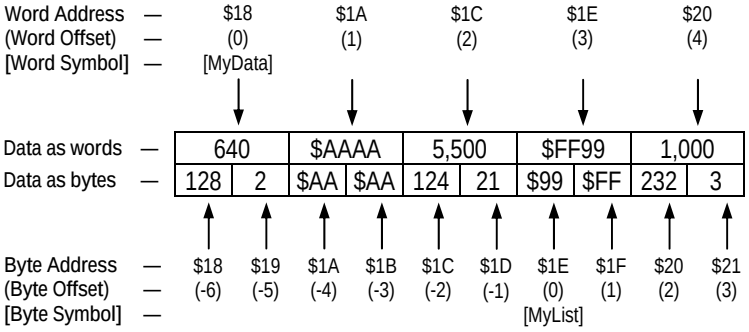


Figura 2-5: Estructura y Dirección Memoria Principal Tamaño Word

Cerca de la parte alta del código, la primer línea ejecutable del método `MemTest`, `Temp := word[@MyData][1]`, lee un valor de tamaño word de la memoria principal. Activa la variable local `Temp` a $AAAA; el valor lee de la dirección de memoria principal $1A. La dirección $1A se determino por la dirección del símbolo `MyData` ($18) mas el Offset 1 (2 bytes). La siguiente simplificación progresiva demuestra esto.

`word[@MyData][1]` ➡ `word[$18][1]` ➡ `word[$18 + (1*2)]` ➡ `word[$1A]`

La siguiente línea, `word[@MyList][0] := Temp + $0123`, escribe un valor tamaño word a memoria principal. Activa el valor en la memoria principal en las direcciones $1E a $ABCD. La dirección $1E se calcula de la dirección del símbolo `MyList` ($1E) mas el Offset 0.

`word[@MyList][0]` ➡ `word[$1E][0]` ➡ `word[$1E + (0*2)]` ➡ `word[$1E]`

El valor $ABCD se derivo del valor actual de Temp más $0123; $AAAA + $0123 que es igual a $ABCD.

Direccionado Memoria Principal

Como sugiere la Figura 2-5, la memoria principal es realmente un set de bytes seguidos (ver renglón “dato como bytes”) que pueden leerse como words (pares de 2 bytes) cuando se hace apropiadamente. El ejemplo anterior muestra que las direcciones son calculadas en términos de bytes. Este concepto es un tema consistente para cualquier instrucción que usa direcciones.

La memoria principal es ultimadamente direccionada en términos de bytes sin importar el tamaño del valor que se esta accesando; byte, word, o long. Esto tiene ventajas cuando se piensa en cuantos bytes, words y longs se relacionan, pero puede causar problemas cuando se piensa en términos de casos diferentes de un solo tamaño, como words.

Por esta razón el asignador WORD tiene una útil característica para facilitar direccionamiento de una perspectiva céntrica de word. Su campo *BaseAddress* cuando se combina con el ampo opcional *Offset* opera de una forma de base consciente.

Imagine accesar words de memoria de un punto de inicio conocido (*BaseAddress*). Quizá piense en el siguiente word o words desde una cierta distancia de ese punto (*Offset*). Mientras esos words están realmente a un cierto numero de "bytes" mas allá de un punto dado, es mas fácil pensar en words mas allá de un punto (Ej.: el 4to word, en vez de el word que empieza despues del 6to byte). El designador WORD lo trata apropiadamente tomando el valor de un valor *Offset* (unidades de words), multiplicado por 2 (numero de bytes por word), y suma ese resultado a *BaseAddress* para determinar la memoria word correcta a leer. también limpia el bit mas bajo de *BaseAddress* para asegurar la dirección referenciada como word alineada.

así, cuando los valores de la lista MyData, word[@MyData][0] lee el primer valor word, word[@MyData][1] lee el segundo valor word y word[@MyData][2] lee el tercero.

Si el campo *Offset* no fuera usado, las instrucciones de arriba tendría algo como esto word[@MyData], word[@MyData+2], y word[@MyData+4], respectivamente. El resultado es el mismo, pero el camino como se escribe puede no ser tan claro.

Para mayor explicación de como los datos se arreglan en memoria, ver la sección DAT Declaración de Data(Sintaxis 1) en Pág. 103.

Una Referencia Alternativa de Memoria

Hay todavía otra forma de accesar los datos desde el código en el ejemplo de arriba; podría referenciar los símbolos de datos directamente. Por ejemplo esta instrucción lee el primer word de la lista MyData:

```
Temp := MyData[0]
```

...y estas instrucciones leen la segunda y tercer words de MyData:

```
Temp := MyData[1]
Temp := MyData[2]
```

así que ¿por que no utilizar referencia de símbolos directos todo el tiempo? Considere el siguiente caso:

```
Temp := MyList[0]
Temp := MyList[1]
```

Refiriendo de regreso al código ejemplo de arriba la Figura 2-5, quizá espere que estas dos instrucciones lean la primer y segunda word de MyList; $FF99 y 1000, respectivamente. En cambio lee el primer y segundo byte de MyList, $99 y $FF, respectivamente.

¿Que sucede? a diferencia de MyData, la entrada MyList se define en el código como tamaño byte alineación byte. Los datos realmente consisten en valores tamaño word, porque cada elemento es precedido por WORD, pero debido a que el símbolo para las listas son declaradas como tamaño byte, todas las referencias directas regresaran bytes individuales.

Sin embargo, el designador WORD puede utilizarse, ya que la lista también toma lugar para ser alineado word por su posicion siguiendo MyData.

```
Temp := word[@MyList][0]
Temp := word[@MyList][1]
```

El código de arriba lee la primer word, $FF99, seguida por la segunda word, 1000, de MyList. Esta característica es muy útil si una lista de datos necesita accesarse como bytes y words en diferentes tiempos en una aplicación.

Otro Fenómeno de Direccionamiento

Ambas técnicas de referencias, WORD y símbolos directos demostrados arriba pueden utilizarse para accesar cualquier localidad en memoria principal, sin importar como se relaciona para definir datos. Aquí algunos ejemplos:

```
Temp := word[@MyList][-1]  'Lee ultimo word MyData (antes MyList)
Temp := word[@MyData][3]   'Lee primer word MyList (despues MyData)
Temp := MyList[-6]         'Lee primer byte MyData
Temp := MyData[-2]         'Lee word que es dos words antes MyData
```

Este ejemplo lee mas allá de las fronteras lógicas (punto de inicio y punto final) de la lista de datos que referencian. Esto puede ser un truco útil, pero con mayor frecuencia se hace por error; tenga cuidado cuando direcciona memoria, especialmente si esta escribiéndola.

Accesando Words de Símbolos de Tamaños mas Largos (Sintaxis 4)

En bloques **PUB** y **PRI**, se usa la sintaxis 4 de **WORD** para leer o escribir componentes tamaño word de variables tamaño long. Por ejemplo:

```
VAR
  long  LongVar

PUB Main
  LongVar.word := 65000      'Activa primer word de LongVar a 65000
  LongVar.word[0] := 65000   'igual que arriba
  LongVar.word[1] := 1       'Activa segundo word de LongVar a 1
```

Este ejemplo acceso los componentes de tamaño word de `LongVar`, individualmente. Los comentarios indican lo que hace cada línea. Al final del método `Main` el metodo `LongVar` será igual a 130,536.

Las mismas técnicas pueden usarse para referenciar componentes tamaño word o símbolos de datos tamaño long.

```
PUB Main | Temp
  Temp := MyList.word[0]           'Lee word bajo de MyList long 0
  Temp := MyList.word[1]           'Lee word alto de MyList long 0
  MyList.word[1] := $1234          'Escribe word alto de MyList long
0
  MyList.word[2] := $FFEE          'Escribe word bajo de MyList long
1

DAT
  MyList  long  $FF998877, $DDDDEEEE  'Dato alineado/tamaño long
```

La primer y segunda línea ejecutable de `Main` lee los valores $8877 y $FF99, respectivamente, de `MyList`. El tercero escribe $1234 al word alto de el long en el elemento 0 de `MyList`, resultando en un valor de $12348877. La cuarta línea escribe $FFEE al word en el índice 2 en `MyList` (el word bajo del long en el elemento 1), resultando en un valor de $DDDDFFEE.

WORDFILL

instrucción: Llena words de memoria principal con un valor.

((PUB PRI))
  WORDFILL (*StartAddress*, *Value*, *Count*)

- *StartAddress* es una expresión indicando la localidad del primer word de memoria a llenar con *Value*.
- *Value* es una expresión indicando el valor a llenar con words.
- *Count* es una expresión indicando el numero de words a llenar, iniciando con *StartAddress*.

Explicación

WORDFILL es una de tres instrucciones (BYTEFILL, WORDFILL, y LONGFILL) usadas para llenar bloques de memoria con un valor especifico. WORDFILL llena *Count* words de la memoria principal con *Value*, comenzando por la localidad *StartAddress*.

Usando WORDFILL

WORDFILL es una forma de limpiar bloques grandes de memoria tamaño Word. Por ejemplo:

```
VAR
  word   Buff[100]

PUB Main
  wordfill(@Buff, 0, 100)     'Limpia Buff a 0
```

La primer línea del método Main de arriba, limpia las 100 word (200 byte) del arreglo Buff, todos a cero. WORDFILL es mas rápido en esta tarea que un ciclo REPEAT dedicado.

WORDMOVE

instrucción: Copia words de una región a otra en memoria principal.

((PUB PRI))
WORDMOVE (*DestAddress*, *SrcAddress*, *Count*)

- *DestAddress* es una expresión que indica la localidad de memoria principal del primer Word destino.
- *SrcAddress* es una expresión que especifica la localidad de memoria del primer Word de la fuente a copiar.
- *Count* esta es una expresión que indica el numero de words a copiar de la fuente al destino.

Explicación

WORDMOVE es una de tres instrucciones (BYTEMOVE, WORDMOVE, y LONGMOVE) que se usan para copiar bloques de memoria principal de un área a otra. WORDMOVE copia *Count* words de memoria principal desde *SrcAddress* a la memoria principal en *DestAddress*.

Usando WORDMOVE

WORDMOVE es una forma de copiar largos bloques de memoria tamaño Word. Por ejemplo:

```
VAR
  word   Buff1[100]
  word   Buff2[100]

PUB Main
  wordmove(@Buff2, @Buff1, 100)     'Copia Buff1 a Buff2
```

La primer línea del método `in`, copia completamente las 100 words (200 bytes) del arreglo `Buff1` al arreglo `Buff2`. WORDMOVE es mas rápido en esta tarea que un ciclo REPEAT.

_XINFREQ

Constante: Constante pre-definida, una vez activable que especifica la frecuencia del cristal externo.

```
CON
  _XINFREQ = expresión
```

- *expresión* es una expresión entera que indica la frecuencia del cristal externo, la frecuencia en el pin XI. Este valor se usa para la aplicación de arranque.

Explicación

_XINFREQ especifica la frecuencia del cristal externo, el cual se usa junto con el modo clock para determinar la frecuencia del reloj del sistema en el arranque. Es un símbolo constante predefinido cuyo valor se determina por el objeto superior de la aplicación. _XINFREQ se active directamente por la aplicación o indirectamente como resultado de los parámetros de _CLKMODE y _CLKFREQ.

El objeto superior en una aplicación(en donde comienza la compilación) puede especificar un parámetro para _XINFREQ en su bloque CON. Esto, junto con el modo clock, define la frecuencia a la cual el reloj del sistema cambiara tan pronto como la aplicación inicie y empiece la ejecución.

La aplicación puede especificar _XINFREQ o _CLKFREQ en el bloque CON; estos son mutuamente exclusivos y los no especificados se calculan automáticamente y se activan como un resultado de la especificación del otro.

Los siguientes ejemplos asumen que están contenidos en el objeto superior. Cualquier parámetro _XINFREQ en los objetos hijos será ignorado por el compilador.

Por ejemplo:

```
CON
  _CLKMODE = XTAL1 + PLL8X
  _XINFREQ = 4_000_000
```

La primera declaración en el bloque CON activa el modo clock para un cristal externo de baja velocidad y un clock PLL multiplicador de 8. La segunda declaración indica que el cristal externo es de 4 MHz, lo cual significa que la frecuencia del reloj del sistema será 32 MHz porque 4 MHz * 8 = 32 MHz. El valor de _CLKFREQ se active automáticamente a 32 MHz debido a estas declaraciones.

```
CON
  _CLKMODE = XTAL2
  _XINFREQ = 10_000_000
```

Estas dos declaraciones activan el modo clock para un cristal externo de velocidad media sin multiplicador PLL y un cristal de frecuencia 10 MHz. El valor **_CLKFREQ** y por lo tanto la frecuencia del reloj del sistema se activan automáticamente a 10 MHz también, debido a estas declaraciones.

Capítulo 3: Referencia Lenguaje Ensamblador

Este capitulo describe todos los elementos del Lenguaje Ensamblador del chip Propeller y se usa como una referencia a elementos individuales del lenguaje ensamblador. Muchas instrucciones tienen su correspondiente instrucción Spin así que una referencia al lenguaje spin se recomienda igualmente.

La referencia del Lenguaje Ensamblador se divide en tres secciones principales:

1) **La Estructura del Ensamblador Propeller:** El código Ensamblador Propeller es una parte opcional de los Objetos Propeller. Esta sección describe la estructura general del código Ensamblador Propeller y como encaja en los objetos.
2) **La Lista categórica del Lenguaje Ensamblador Propeller.** Todos los elementos, incluyendo operadores, están agrupados por la función. Esta es una forma rápida de ampliar el uso del lenguaje y las características disponibles para usos específicos. Cada elemento tiene una pagina de referencia para mayor información. Algunos elementos están marcados con un súper script "s" que indica que están disponibles en Spin, sin embargo la sintaxis puede variar.
3) **Los Elementos del Lenguaje Ensamblador.** Todas las instrucciones están incluidas en una Tabla Maestra al inicio, y la mayoría de los elementos tienen su subsección dedicada, acomodada alfabéticamente. Los elementos individuales sin subsección dedicada, tales como operadores, están agrupados en otras sub secciones pero pueden localizarse fácilmente siguiendo las paginas de referencia de la Lista Categórica.

La Estructura del Ensamblador Propeller

Cada Objeto propeller contiene código Spin mas código y datos opcionales ensamblador. Un código Spin de Objeto proporciona una estructura, que consiste en bloques de propósito especial. El código de datos y Ensamblador Propeller se localizan en el bloque DAT Pág. 102.

EL código Spin se ejecuta de RAM principal por un cog que usa el interprete Spin, sin embargo, el ensamblador Propeller se ejecuta directamente del mismo cog. Por esta naturaleza, el código ensamblador Propeller y cualquier dato perteneciente a el debe cargarse (enteramente) en un cog en el orden que se ejecutara. De esta forma, ambos códigos ensamblador y datos se tratan igual durante el proceso de carga del cog.

Aquí hay un ejemplo de un Objeto Propeller. Su código spin en el bloque **PUB**, `Main`, inicia otro cog para correr la rutina del Ensamblador Propeller del bloque **DAT**, `Toggle`.

```
{{ AssemblyToggle.spin }}

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


PUB Main
{Inicia cog para cambiar P16 sin fin}

  cognew(@Toggle, 0)                        'Inicia Nuevo cog


DAT
{Toggle P16}
                    org     0               'Inicia a Cog RAM dir 0
Toggle              mov     dira, Pin       'Activa Pin a Salida
                    mov     Time, cnt       'Calcula tiempo de retraso
                    add     Time, #9        'Activa mínimo retraso
aqui
:loop               waitcnt Time, Delay     'Espera
                    xor     outa, Pin       'Cambia Pin
                    jmp     #:loop          'Ciclo sin final

Pin     long        |< 16                   'Numero de Pin
Delay   long        6_000_000               'Ciclos de Reloj  a
retrasar
Time                res  1                  'Espacio de Trabajo del
Contador del Sistema
```

Cuando el método `Main` de la instrucción **COGNEW** se ejecuta, un Nuevo cog comienza a llenar la RAM del cog con 496 longs consecutivos de Memoria Principal, iniciando con la instrucción en la dirección de `Toggle`. Posteriormente el cog inicia sus registros de Propósito Especial y comienza a ejecutar código iniciando en el registro RAM del cog 0.

Ambos datos y ensamblador deben mezclarse con el bloque **DAT** pero se debe tener cuidado para acomodar los elementos críticos que se cargan en el cog en el orden apropiado para ejecutar. Se recomienda escribir en el siguiente orden: 1) código Ensamblador, 2) Inicializar

datos simbólicos (Ej.: LONG), 3) memoria reservada simbólica (Ej. RES). Esto hace que el cog cargue el lenguaje ensamblador primero, seguido inmediatamente de datos inicializados y datos de aplicación después, aun si no es requerido por el código. Ver las secciones que hablan de ORG (Pág. 338), RES (Pág. 352), y DAT (Pág. 102) para mayor información.

Memoria de Cog

La RAM del Cog es similar a la RAM principal en lo siguiente:

- Cada contenedor tiene instrucciones de programa y/o datos.
- Cada contenedor se modifica en tiempo de ejecución (Ej.: Variable residente RAM)

La RAM del cog es diferente de RAM principal en:

- La RAM del cog es mas pequeña y rápida que la RAM principal
- La RAM de cog es un grupo de registros direcccionables solo como longs (cuatro bytes) la RAM principal son localidades direccionable en bytes, words o longs.
- El Ensamblador Propeller ejecuta desde la RAM del cog mientras Spin se obtiene y ejecuta de la RAM principal.
- La RAM del cog esta disponible solo para su cog, RAM principal es compartida.

Una vez que el ensamblador se carga en la RAM del cog, el cog lo ejecuta leyendo un long (32 bits) del registro (iniciando con el registro 0), resolviendo su destino y fuente, ejecutando la instrucción (posiblemente escribiendo el resultado en otro registro) y luego moviéndose a la siguiente dirección para repetir el proceso. El registro de la RAM del cog puede contener instrucciones o datos puros, y cada uno puede modificarse como resultado de la ejecución de otra instrucción.

¿De donde obtiene sus datos una instrucción?

La mayoría de las instrucciones tienen dos operandos de datos; un valor destino y un valor fuente. Por ejemplo, el formato para una instrucción ADD es:

add *destination*, ⟨#⟩*source*

El operando *destination* son la dirección de 9 bits de un registro que contiene el valor deseado para operar. El operando *source* es un valor literal de 9 bits (constante) o una dirección de 9 bits de un registro que contiene el valor deseado. El significado del operando source depende de si hay o no un indicador "#". Por ejemplo:

```
   add  X, #25           'Suma 25 a X
   add  X, Y             'Suma Y a X

X  long  50
Y  long  10
```

La primera instrucción suma el valor literal 25 al valor almacenado en el registro X. La segunda instrucción suma el valor almacenado en el registro Y al valor almacenado en el registro X. En ambos casos, el resultado de la suma se almacena de regreso en X.

Las dos ultimas líneas definen símbolos de datos X y Y como valores long 50 y 10, respectivamente. Como el código ensamblador iniciado en el cog hace que estos datos entren en la RAM del cog justo despues del as instrucciones, X naturalmente es un símbolo que apunta al registro conteniendo 50, y Y es un símbolo que apunta al registro que contiene 10.

Por lo tanto el resultado de la primer instrucción **ADD** es 75 (Ej.: X + 25 → 50 + 25 = 75) y ese valor se almacena de regreso en X. Similar a esto el resultado de la segunda instrucción **ADD** es 85 (Ej.: X + Y → 75 + 10 = 85) y así X se activa a 85.

No olvide el indicador literal '#'

Asegúrese de ingresar el indicador literal, #, cuando la intención es un valor literal (también conocido como valor inmediato). Modificando la primer línea del ejemplo anterior al omitir el carácter # (Ej.: ADD X, 25) hace que el valor en el registro 25 se sume a X en vez de agregar el valor 25 a X.

Otro posible error es omitir el # en instrucciones como **JMP** y **DJNZ**. Si el destino deseado es una etiqueta llamada MyRoutine, una instrucción **JMP** debería normalmente verlo como JMP #MyRoutine en vez de JMP MyRoutine. La causa final es el valor almacenado en el registro MyRoutine para usarse como dirección para brincar a; esto es útil para saltos indirectos pero no es normalmente la intención del desarrollador.

Literales Deben Ajustarse en 9 Bits

La fuente del operando es solo 9 bits de ancho; puede tener un valor de 0 a 511 ($000 a $1FF). Tenga en cuenta cuando especifica valores literales. Si un valor es mas grande que los 9 bits, debe almacenarse en un registro y accesarse con la dirección de registro. Por ejemplo:

```
   add  X, BigValue      'Suma BigValue a X

X         long  50
BigValue  long  1024
```

Etiquetas Globales y Locales

Para dar nombre a rutinas especiales, el código Ensamblador Propeller puede hacer uso de dos tipos de etiquetas: Global y local.

Las etiquetas globales se ven como otros símbolos y siguen las mismas reglas que los símbolos; comienzan con guión bajo '_' o una letra y son seguidos por mas letras, guiones bajos y/o números. Ver Reglas de , Pág. 47, para mas información.

Las etiquetas locales son similares a las globales excepto que empiezan con dos puntos ':' y deben separarse de otras etiquetas locales de mismo nombre por al menos una etiqueta global. Aqui un ejemplo:

```
Addition            mov     Count, #9       'Act 'Add' Cont de ciclo
:loop               add     Temp, X         'Iterativa Temp+X
                    djnz    Count, #:loop   'Dec counter, ciclo back

Subtraction         mov     Count, #15      'Act 'Sub' Cont de ciclo
:loop               sub     Temp, Y         'Iterativa Temp-Y
                    djnz    Count, #:loop   'Dec counter, ciclo back

                    jmp     #Addition       'Brinca a add
```

Este ejemplo tiene dos etiquetas globales, `Addition` y `Subtraction`, y dos etiquetas locales, ambas llamadas `:loop`. Las etiquetas locales pueden tener exactamente el mismo nombre, `:loop`, porque el menos una etiqueta global las separa. De hecho este es el punto de las etiquetas locales; indican cosas comunes, genéricas como ciclos sin requerir nombres únicos para cada uno de ellos.

Las dos instrucciones **DJNZ** son exactamente las mismas, pero van a diferentes lugares. El **DJNZ** de la rutina `Addition` va de regreso a la etiqueta local `:loop` en `Addition`, y la **DJNZ** de la rutina `Subtraction` va de regreso a la etiqueta local `:loop` en `Subtraction`.

Observe que las instrucciones **DJNZ** usan el indicador literal, `#`, y el mismo nombre de la etiqueta local, incluyendo los dos puntos. Sin el `#` el código se ejecutaría inapropiadamente, y sin los dos puntos el código daría un error. Para mas información del formato Ensamblador Propeller, ver Elementos Comunes de Sintaxis, Pág. 255.

Lista Categórica de Lenguaje Ensamblador Propeller

Instrucciones

ORG	Ajusta en tiempo de compilación la dirección del apuntador; p 338.
FIT	Valida que la instrucción o datos previos se ajustan al cog; p 299.
RES	Reserva los siguientes longs para símbolos; p 351.

Configuración

CLKSET[s]	Activa el modo clock a tiempo d ejecución; p 277.

Control de Cog

COGID[s]	Obtiene el ID del Cog actual; p 289.
COGINIT[s]	Inicia o reinicia un cog por ID; p 290.
COGSTOP[s]	Detiene un cog por ID; p 292.

Control de Proceso

LOCKNEW[s]	Activa un nuevo seguro; p 313.
LOCKRET[s]	Regresa un seguro; p 314.
LOCKCLR[s]	Limpia un seguro por ID; p 312.
LOCKSET[s]	Activa un seguro por ID; p 316.
WAITCNT[s]	Detiene una ejecución temporalmente; p 383.
WAITPEQ[s]	Detiene ejecución hasta que pin coincide con estado asignado; p 384.
WAITPNE[s]	Detiene ejecución hasta que pin no coincide con estado asignado; p 385.
WAITVID[s]	Detiene ejecución hasta que Video Generador esta listo para datos píxel; p 386.

Condiciones

IF_ALWAYS	Siempre; p 303.
IF_NEVER	nunca; p 303.
IF_E	Si es igual (Z = 1); p 303.
IF_NE	Si no es igual (Z = 0); p 303.

IF_A	Si por encima (!C & !Z = 1); p 303.
IF_B	Si por debajo (C = 1); p 303.
IF_AE	Si por encima o igual (C = 0); p 303.
IF_BE	Si por debajo o igual (C | Z = 1); p 303.
IF_C	Si C Activa; p 303.
IF_NC	Si C Limpia; p 303.
IF_Z	Si Z Activa; p303.
IF_NZ	Si Z Limpia; p 303.
IF_C_EQ_Z	Si C es igual a Z; p 303.
IF_C_NE_Z	Si C no es igual a Z; p 303.
IF_C_AND_Z	Si C activa y Z activa; p 303.
IF_C_AND_NZ	Si C activa y Z limpia; p 303.
IF_NC_AND_Z	Si C limpia y Z activa; p 303.
IF_NC_AND_NZ	Si C limpia y Z limpia; p 303.
IF_C_OR_Z	Si C o Z activan; p 303.
IF_C_OR_NZ	Si C o Z limpian; p 303.
IF_NC_OR_Z	Si C limpia o Z activa; p 303.
IF_NC_OR_NZ	Si C limpia o Z limpia; p 303.
IF_Z_EQ_C	Si Z es igual a C; p 303.
IF_Z_NE_C	Si Z no es igual a C; p 303.
IF_Z_AND_C	Si Z activa y C activa; p 303.
IF_Z_AND_NC	Si Z activa y C limpia; p 303.
IF_NZ_AND_C	Si Z limpia y C activa; p 303.
IF_NZ_AND_NC	Si Z limpia y C limpia; p 303.
IF_Z_OR_C	Si Z activa o C activa; p 303.
IF_Z_OR_NC	Si Z activa o C limpia; p 303.
IF_NZ_OR_C	Si Z limpia o C activa; p 303.
IF_NZ_OR_NC	Si Z limpia o C limpia; p 303.

Control de Flujo

CALL	Va a dirección con intención de regresar a la siguiente instrucción; p 274.
DJNZ	Decrementa valor y va a dirección si no es cero; p 297.
JMP	Va a dirección incondicionalmente; p 306.
JMPRET	Va a dirección con intención a “regresar” a otra dirección; p 308.
TJNZ	Prueba valor y va a dirección si no es cero; p 377.
TJZ	Prueba valor y va a dirección si es cero; p 380.
RET	Regresa a dirección almacenada; p 356.

Efectos

NR	No resultado (no escribe resultado); p 298.
WR	Escribe resultado; p 298.
WC	Escribe estado C; p 298.
WZ	Escribe estado Z; p 298.

Acceso de Memoria Principal

RDBYTE	Lee byte de memoria principal; p 348.
RDWORD	Lee word de memoria principal; p 350.
RDLONG	Lee long de memoria principal; p 349.
WRBYTE	Escribe byte a memoria principal; p 389.
WRWORD	Escribe word a memoria principal; p 391.
WRLONG	Escribe long a memoria principal; p 390.

Operaciones Comunes

ABS	Obtiene valor absoluto de un numero; p 263.
ABSNEG	Obtiene valor absoluto de un numero negativo; p 264.
NEG	Obtiene el negativo de un numero; p 329.
NEGC	Obtiene un valor o su inverso aditivo, basado en C; p 330.
NEGNC	Obtiene un valor o su inverso aditivo, basado en !C; p 331.
NEGZ	Obtiene un valor o su inverso aditivo, basado en Z; p 333.

NEGNZ	Obtiene un valor o su inverso aditivo, basado en !Z; p 332.
MIN	Limite mínimo de valor no signado a otro valor no signado; p 319.
MINS	Limite mínimo de valor signado a otro valor signado; p 320.
MAX	Limite máximo de valor no signado a otro valor no signado; p 317.
MAXS	Limite máximo de valor signado a otro valor signado; p 318.
ADD	Suma dos valores no signados; p 265.
ADDABS	Suma valor absoluto a otro valor; p 266.
ADDS	Suma dos valores signados; p 268.
ADDX	Suma dos valores no signados mas C; p 270.
ADDSX	Suma dos valores signados mas C; p 268.
SUB	Resta dos valores no signados; p 363.
SUBABS	Resta un valor absoluto de otro valor; p 364.
SUBS	Resta dos valores signados; p 365.
SUBX	Resta valor no signado mas C de otro valor no signado; p 368.
SUBSX	Resta valor signado mas C de otro valor signado; p 366.
SUMC	Valor Sum signado con otro de signo afectado C; p 370.
SUMNC	Valor Sum signado con otro de signo afectado !C; p 371.
SUMZ	Valor Sum signado con otro de signo afectado Z; p 374.
SUMNZ	Valor Sum signado con otro de signo afectado !Z; p 373.
MUL	<reservado para uso futuro>
MULS	<reservado para uso futuro>
AND	Bitwise AND de dos valores; p 272.
ANDN	Bitwise valor AND con NOT de otro; p 273.
OR	Bitwise OR de dos valores; p 337.
XOR	Bitwise XOR de dos valores; p 393.
ONES	<reservado para uso futuro>
ENC	<reservado para uso futuro>
RCL	Rota C a la izquierda en un valor por el numero especificado de bits; p 346.
RCR	Rota C a la derecha en un valor por el numero especificado de bits; p 347.

REV	Reversa LSB del valor y cero extendido; p 357.
ROL	Rota valor a la izquierda por un numero especifico de bits; p 358.
ROR	Rota valor a la derecha por un numero especifico de bits; p 359.
SHL	Mueve valor a la izquierda por un numero especifico de bits; p 361.
SHR	Mueve valor a la derecha por un numero especifico de bits; p 362.
SAR	Mueve valor aritméticamente a la derecha por un numero especifico de bits; p 360.
CMP	Compara dos valores no signados; p 278.
CMPS	Compara dos valores signados; p 280.
CMPX	Compara dos valores no signados mas C; p 286.
CMPSX	Compara dos valores signados mas C; p 283.
CMPSUB	Compara valores no signados, resta el segundo si es menor igual; p 282.
TEST	Bitwise AND de dos valores solo para afectar banderas; p 377.
TESTN	Bitwise AND de un valor con NOT de otro valor solo para afectar banderas; p 378.
MOV	Activa un registro a un valor; p 321.
MOVS	Activa un campo fuente de registro a un valor; p 323.
MOVD	Activa un campo destino de registro a un valor; p 322.
MOVI	Activa un campo instrucción de registro a un valor; p 322.
MUXC	Activa bits discretos de un valor al estado de C; p 325.
MUXNC	Activa bits discretos de un valor al estado de !C; p 326.
MUXZ	Activa bits discretos de un valor al estado de Z; p 328.
MUXNZ	Activa bits discretos de un valor al estado de !Z; p 327.
HUBOP	Desarrolla una operación de Hub; p 301.
NOP	No operación, solo deja pasar cuatro ciclos; p 334.

Constantes

NOTA: Referencia a Constantes (pre-definidas) en Capítulo 2: Introducción al Chip Propeller

TRUE[s]	Lógico verdadero: -1 ($FFFFFFFF); p 96.
FALSE[s]	Lógico falso: 0 ($00000000); p 96.
POSX[s]	Máximo entero positivo: 2,147,483,647 ($7FFFFFFF); p 97.

NEGX[s]	Máximo entero negativo: -2,147,483,648 ($80000000); p 97.
PI[s]	Valor Flotante de PI: ~3.141593 ($40490FDB); p 97.

Registros

DIRA[s]	Registro Dirección puerto A 32 bits; p 351.
DIRB[s]	Registro Dirección puerto A 32 bits (uso futuro); p 351.
INA[s]	Registro Entrada puerto A 32 bits (solo lectura); p 351.
INB[s]	Registro Entrada puerto A 32 bits (solo lectura) (uso futuro); p 351.
OUTA[s]	Registro Salida puerto A 32 bits; p 351.
OUTB[s]	Registro Salida puerto A 32 bits (uso futuro); p 351.
CNT[s]	Registro Contador del Sistema 32 bits (solo lectura); p 351.
CTRA[s]	Registro de Control Contador A; p 351.
CTRB[s]	Registro de Control Contador B; p 351.
FRQA[s]	Registro de Frecuencia Contador A; p 351.
FRQB[s]	Registro de Frecuencia Contador B; p 351.
PHSA[s]	Registro de Fase Cerrada Contador A (PLL); p 351.
PHSB[s]	Registro de Fase Cerrada Contador B (PLL); p 351.
VCFG[s]	Registro de configuración de Video; p 351.
VSCL[s]	Registro de Escala de Video; p 351.
PAR[s]	Registro de parámetros de Inicio de Cog (solo lectura); p 351.

Operadores Unarios

NOTA: Todos los operadores mostrados son operadores de expresiones constantes.

+	Positivo (+X) forma unaria de Suma; p 336.
-	Negativo (−X); forma unaria de Resta; p 336.
^^	Raíz Cuadrada; p 336.
||	Valor Absoluto; p 336.
|<	Valor Decodificado 90-31) en bit simple long alto; p 336.
>|	Long Codificado en valor (0 - 32) como bit alta prioridad; p 336.
!	Bitwise: NOT; p 336.
@	símbolo de Dirección; p 336.

Operadores Binarios

NOTA: Todos los operadores mostrados son operadores de expresiones constantes.

+	Suma; p 336.
-	Resta; p 336.
*	Multiplica y regresa los 32 bits mas bajos (signado); p 336.
**	Multiplica y regresa los 32 bits mas altos (signado); p 336.
/	Divide y regresa el cociente (signado); p 336.
//	Divide y regresa el remanente (signado); p 336.
#>	Limite mínimo (signado); p 336.
<#	Limite máximo (signado); p 336.
~>	Mueve aritméticamente a la derecha; p 336.
<<	Bitwise: Movimiento a la izquierda; p 336.
>>	Bitwise: Movimiento a la derecha; p 336.
<-	Bitwise: Rota a la izquierda; p 336.
->	Bitwise: Rota a la derecha; p 336
><	Bitwise: Reversa; p 336.
&	Bitwise: AND; p 336.
|	Bitwise: OR; p 336.
^	Bitwise: XOR; p 336.
AND	Booleano: AND (promotes non-0 to -1); p 336.
OR	Booleano: OR (promotes non-0 to -1); p 336.
==	Booleano: Es igual; p 336.
<>	Booleano: No es igual; p 336.
<	Booleano: Es menor que (signado); p 336.
>	Booleano: Es mayor que (signado); p 336.
=<	Booleano: Es igual o menor (signado); p 336.
=>	Booleano: Es igual o mayor (signado); p 336.

Elementos de Lenguaje Ensamblador

Definiciones de Sintaxis

Además de las descripciones detalladas, las siguientes paginas contienen definiciones de sintaxis para muchos elementos que describe, en términos cortos, todas las opciones de ese elemento. Las definiciones de sintaxis usan símbolos especiales para indicar cuando y como se usaran ciertas características de elementos.

BOLDCAPS	Las negritas mayúsculas se escribirse tal y como se muestran.
Bold Italics	Las itálicas negritas deben reemplazarse por texto de usuario; símbolos, operadores, expresiones, etc.
. : , #	Comas, dos puntos, puntos y signo de numero se escriben como se muestran.
〈 〉	Corchetes angulares encierran partes opcionales. Escriba la parte encerrada si lo desea, no ingrese los corchetes angulados.
Doble línea	Separa instrucciones del resto del valor.

Elementos Comunes de Sintaxis

Cuando lea las definiciones de sintaxis en este capitulo, tenga en cuenta que todas las instrucciones propeller tienen tres elementos comunes opcionales: una etiqueta, una condición y efectos. Cada instrucción tiene la siguiente sintaxis básica:

〈*Label*〉 〈*Condition*〉 *Instruction Operands* 〈*Effects*〉

- *Label* — una etiqueta declaratoria opcional. *Label* puede ser global (iniciando con un guión bajo '_' o una letra) o puede ser local (iniciando con dos puntos ':'). Las *Labels* locales deben separarse de otras labels del mismo nombre por al menos una etiqueta global. *Label* se usa par instrucciones como JMP, CALL y COGINIT para asignar el destino objetivo. Ver Etiquetas Globales y Locales en Pág. 247 para mayor información.
- *Condition* — una condición opcional d ejecución (IF_C, IF_Z, etc.) que hace que *Instruction* se ejecute o no. Ver IF_x (Condiciones) en la Pág. 302 para mas información.

- *Instruction* y *Operands* — Una instrucción de ensamblador Propeller (**MOV**, **ADD**, **COGINIT**, etc.) y su cero, uno o dos operandos requeridos por *Instruction*.
- *Effects* — una lista opcional de una a tres efectos de ejecución (**WZ**, **WC**, **WR**, y **NR**) para aplicar a la instrucción, si s ejecuto. Hacen que *Instruction* modifique la bandera Z, bandera C, y que escriba o no el valor del resultado de la instrucción al registro destino respectivamente. Ver Efectos en Pág. 298 para mayor información..

Como cada instrucción puede incluir estos tres campos opcionales (*Label*, *Condition*, y *Effects*), para simplificar esos campos comunes se dejan intencionalmente fuera de la descripción de la sintaxis de la instrucción.

así que cuando usted lea la descripción de la sintaxis como:

WAITCNT *Target*, ⟨#⟩ *Delta*

...recuerde que la verdadera sintaxis es esta:

⟨*Label*⟩ ⟨*Condition*⟩ **WAITCNT** *Target*, ⟨#⟩ *Delta* ⟨*Effects*⟩

Esta regla aplica solo para instrucciones del Ensamblador Propeller; no aplica para Directivas del Ensamblador Propeller.

Las declaraciones de sintaxis siempre dan nombres descriptivos a los operandos de la instrucción, tales como los operandos WAITCNT *Target* y *Delta* en el ejemplo anterior. Las descripciones detalladas se refieren a los operandos por estos nombres, sin embargo, las tablas del código operacional y las tablas de verdad siempre usan los nombres genéricos (D, DEST, Destino, y S, SRC, Fuente) para referirse a los bits de la instrucción que almacenan sus respectivos valores.

códigos Operacionales y sus Tablas

La mayoría de las sintaxis incluyen una tabla de código operacional similar al que se muestra abajo. Esta tabla enlista las instrucciones del código operacional de 32 bits, salidas y numero de ciclos de reloj.

La primer columna de la tabla del código operacional contiene el código operacional de las Instrucciones del Ensamblador Propeller, consiste de los siguientes campos:

- INSTR (bits 31:26) - Indica la instrucción a ser ejecutada.
- ZCRI (bits 25:22) - Indica el estado del efecto de la instrucción y el significado del campo SRC.

- CON (bits 21:18) - Indica la condición en la cual ejecuta la instrucción.
- DEST (bits 17:9) - Contiene la dirección del registro destino.
- SRC (bits 8:0) - Contiene la fuente de la dirección del registro o el valor literal de 9 bits.

Cada uno de los bits del campo ZCRI contienen un 1 o un 0 para indicar si deberán escribirse o no las banderas 'Z', 'C', y 'R', así como si el campo SRC contiene o no un valor 'Inmediato (en vez de una dirección de registro). El bit Z y C del campo ZCRI se limpian a (0) por defecto y se activan (1) si la instrucción se especifico con un efecto WZ y/o WC. Ver Efectos en Pág. 298. El estado por defecto del bit R depende del tipo de instrucción, pero también se afecta si la instrucción se especifico con el efecto WR o NR. El estado por defecto del campo I depende del tipo de instrucción y se afecta por la suma o falta del indicador literal # en el campo fuente de la instrucción.

Los bits del campo CON usualmente son todos por defecto (1111) pero se afectan si la instrucción se especifico con una condición. VerIF_x (Condiciones) en Pág. 302.

Las ultimas cuatro columnas de la tabla del código operacional indican el significado de la salida de la instrucción Z y C, por defecto el comportamiento para escribir o no escribir el valor del resultado, y el numero de ciclos que la instrucción requiere para su ejecución.

Tabla de código Operacional (Opcode) CLKSET:

–INSTR– ZCRI –CON– –DEST– –SRC–	Z Resultado	C Resultado	Resultado	Ciclos
000011 0001 1111 ddddddddd ------000	---	---	Not Written	7..22

Tablas de verdad Concisas

Después de la tabla opcode, hay una tabla de verdad concisa. La tabla de verdad concisa demuestra ejemplos de entrada y salidas resultantes para la instrucción correspondiente. En vez de mostrar cada caso de posibles entradas/salidas, la tabla de verdad toma ventaja de las fronteras numéricas o lógicas que resultan en una actividad de banderas y destinos de salidas. Esta información puede ayudar para aprender o verifican la función y comportamiento intrínseco de la instrucción.

Generalmente la tabla de verdad deberán leerse con cuidado desde el renglón superior hacia el de abajo. Cuando muchos casos de fronteras son posibles, los renglones relacionados están agrupados para enfatizar y separar de otros grupos por una línea gruesa horizontal.

Se utilizan las siguientes convenciones:

$FFFF_FFFE; -2	Valores de números en hexadecimal (izquierda de ‘;’) y decimal (derecha de ‘;’).
%0_00000011; 3	Valores de números en binario (izquierda de ‘;’) y decimal (derecha de ‘;’).
0 –o– 1	Cero individual (0) o uno (1) significan binario 0 o 1.
wr, wz, wc	Efectos ensamblador indican estado de ejecución; escribe-resultado, escribe-a, escribe-c.
x	Minúscula “x” indica piezas donde un valor posible aplica.
---	líneas indican piezas que no aplican o no son importantes.

Un buen ejemplo de una tabla de verdad para la instrucción ADDS:

Tabla de verdad ADDS

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	-	-	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	-	-	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	-	-	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	-	-	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

En la tabla de verdad de ADDS hay ocho renglones de datos agrupados en cuatro pares. Cada grupo desarrolla una condición diferente. Las primeras cinco columnas de cada renglón indican las entradas a la instrucción y las ultimas tres columnas muestran los resultados de salida.

- El primer par de datos demuestra una suma simple con signo (-1 + 1) que da por resultado cero (bandera z activa) y un ejemplo (-1 + 2) que da como resultado un no-cero (bandera Z limpia).
- El segundo par de renglones es el mismo concepto pero con signos invertidos en los valores; (1 + -1) y (1 + -2).
- El tercer par de renglones muestra una suma cerca de la frontera del entero mas alto signado (2,147,482,646 + 1) seguido por otro que cruza esa frontera (2,147,482,646 + 2) el cual resulta en un sobre flujo de signo (bandera C activa).

- El cuarto par de renglones muestran el mismo concepto pero alcanzando y cruzando la frontera de los enteros signados por el lado negativo, resultando también en un sobre flujo de signo (bandera C activa).

Observe que un campo destino de instrucción contiene actualmente la dirección del registro que tiene el valor del operando, y el campo fuente es comúnmente codificado de forma similar, pero las tablas de verdad siempre simplifican este detalle mostrando solo el valor del operando deseado para cada fuente y destino.

Tabla Maestra de Instrucciones de Ensamblador Propeller

Una tabla maestra para todas las instrucciones de Ensamblador propeller se proporciona en las siguientes dos paginas. En esta tabla D y S se refieren al destino de las instrucciones y campos fuente, también conocidos como campo-d y campo-s respectivamente. Por favor asegúrese de leer las notas en la pagina siguiente a la tabla.

instrucción		-INSTR- ZCRI -CON- -DEST- -SRC-	Resultado Z	Resultado C	Resultado	Ciclos
ABS	D, S	101010 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
ABSNEG	D, S	101011 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Escrito	4
ADD	D, S	100000 001i 1111 ddddddddd sssssssss	D + S = 0	Acarreo sin signo	Escrito	4
ADDABS	D, S	100010 001i 1111 ddddddddd sssssssss	D + |S| = 0	Acareo sin signo [3]	Escrito	4
ADDS	D, S	110100 001i 1111 ddddddddd sssssssss	D + S = 0	Signo sobre flujo	Escrito	4
ADDSX	D, S	110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Signo sobre flujo	Escrito	4
ADDX	D, S	110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	Acarreo sin signo	Escrito	4
AND	D, S	011000 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
ANDN	D, S	011001 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
CALL	#S	010111 0011 1111 ????????? sssssssss	Resultado = 0	---	Escrito	4
CLKSET	D	000011 0001 1111 ddddddddd ------000	---	---	No Escrito	7..22 [1]
CMP	D, S	100001 000i 1111 ddddddddd sssssssss	D = S	Sin signo (D < S)	No Escrito	4
CMPS	D, S	110000 000i 1111 ddddddddd sssssssss	D = S	Signo (D < S)	No Escrito	4
CMPSUB	D, S	111000 001i 1111 ddddddddd sssssssss	D = S	Sin signo (D => S)	Escrito	4
CMPSX	D, S	110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Signo (D < S+C)	No Escrito	4
CMPX	D, S	110011 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	Sin signo (D < S+C)	No Escrito	4
COGID	D	000011 0011 1111 ddddddddd ------001	ID = 0	0	Escrito	7..22 [1]
COGINIT	D	000011 0001 1111 ddddddddd ------010	ID = 0	No cog libre	No Escrito	7..22 [1]
COGSTOP	D	000011 0001 1111 ddddddddd ------011	Detenido ID = 0	No cog libre	No Escrito	7..22 [1]
DJNZ	D, S	111001 001i 1111 ddddddddd sssssssss	Resultado = 0	no signado Borrow	Escrito	4 or 8 [2]
HUBOP	D, S	000011 000i 1111 ddddddddd sssssssss	Resultado = 0	---	No Escrito	7..22 [1]
JMP	S	010111 000i 1111 --------- sssssssss	Resultado = 0	---	No Escrito	4
JMPRET	D, S	010111 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	4
LOCKCLR	D	000011 0001 1111 ddddddddd ------111	ID = 0	Estado seguro previo	No Escrito	7..22 [1]
LOCKNEW	D	000011 0011 1111 ddddddddd ------100	ID = 0	No seguro libre	Escrito	7..22 [1]
LOCKRET	D	000011 0001 1111 ddddddddd ------101	ID = 0	No seguro libre	No Escrito	7..22 [1]
LOCKSET	D	000011 0001 1111 ddddddddd ------110	ID = 0	Estado seguro previo	No Escrito	7..22 [1]
MAX	D, S	010011 001i 1111 ddddddddd sssssssss	S = 0	Sin signo (D < S)	Escrito	4
MAXS	D, S	010001 001i 1111 ddddddddd sssssssss	S = 0	Signo (D < S)	Escrito	4
MIN	D, S	010010 001i 1111 ddddddddd sssssssss	S = 0	Sin signo (D < S)	Escrito	4
MINS	D, S	010000 001i 1111 ddddddddd sssssssss	S = 0	Signo (D < S)	Escrito	4
MOV	D, S	101000 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
MOVD	D, S	010101 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	4
MOVI	D, S	010110 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	4
MOVS	D, S	010100 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	4
MUXC	D, S	011100 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
MUXNC	D, S	011101 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
MUXNZ	D, S	011111 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
MUXZ	D, S	011110 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4

instrucción	-INSTR- ZCRI -CON- -DEST- -SRC-	Resultado Z	Resultado C	Resultado	Ciclos
NEG D, S	101001 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
NEGC D, S	101100 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
NEGNC D, S	101101 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
NEGNZ D, S	101111 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
NEGZ D, S	101110 001i 1111 ddddddddd sssssssss	Resultado = 0	S[31]	Escrito	4
NOP	------ ---- 0000 --------- ---------	---	---	---	4
OR D, S	011010 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4
RCL D, S	001101 001i 1111 ddddddddd sssssssss	Resultado = 0	D[31]	Escrito	4
RCR D, S	001100 001i 1111 ddddddddd sssssssss	Resultado = 0	D[0]	Escrito	4
RDBYTE D, S	000000 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	7..22 [1]
RDLONG D, S	000010 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	7..22 [1]
RDWORD D, S	000001 001i 1111 ddddddddd sssssssss	Resultado = 0	---	Escrito	7..22 [1]
RET	010111 0001 1111 --------- ---------	Resultado = 0	---	No Escrito	4
REV D, S	001111 001i 1111 ddddddddd sssssssss	Resultado = 0	D[0]	Escrito	4
ROL D, S	001001 001i 1111 ddddddddd sssssssss	Resultado = 0	D[31]	Escrito	4
ROR D, S	001000 001i 1111 ddddddddd sssssssss	Resultado = 0	D[0]	Escrito	4
SAR D, S	001110 001i 1111 ddddddddd sssssssss	Resultado = 0	D[0]	Escrito	4
SHL D, S	001011 001i 1111 ddddddddd sssssssss	Resultado = 0	D[31]	Escrito	4
SHR D, S	001010 001i 1111 ddddddddd sssssssss	Resultado = 0	D[0]	Escrito	4
SUB D, S	100001 001i 1111 ddddddddd sssssssss	D - S = 0	Sin signo Borrow	Escrito	4
SUBABS D, S	100011 001i 1111 ddddddddd sssssssss	D - |S| = 0	Sin signo Borrow [4]	Escrito	4
SUBS D, S	110101 001i 1111 ddddddddd sssssssss	D - S = 0	Signo Overflow	Escrito	4
SUBSX D, S	110111 001i 1111 ddddddddd sssssssss	Z & (D-(S+C) = 0)	Signo Overflow	Escrito	4
SUBX D, S	110011 001i 1111 ddddddddd sssssssss	Z & (D-(S+C) = 0)	Sin signo Borrow	Escrito	4
SUMC D, S	100100 001i 1111 ddddddddd sssssssss	D ± S = 0	Signo Overflow	Escrito	4
SUMNC D, S	100101 001i 1111 ddddddddd sssssssss	D ± S = 0	Signo Overflow	Escrito	4
SUMNZ D, S	100111 001i 1111 ddddddddd sssssssss	D ± S = 0	Signo Overflow	Escrito	4
SUMZ D, S	100110 001i 1111 ddddddddd sssssssss	D ± S = 0	Signo Overflow	Escrito	4
TEST D, S	011000 000i 1111 ddddddddd sssssssss	D = 0	Paridad de resultado	No Escrito	4
TESTN D, S	011001 000i 1111 ddddddddd sssssssss	D = 0	Paridad de resultado	No Escrito	4
TJNZ D, S	111010 000i 1111 ddddddddd sssssssss	D = 0	0	No Escrito	4 or 8 [2]
TJZ D, S	111011 000i 1111 ddddddddd sssssssss	D = 0	0	No Escrito	4 or 8 [2]
WAITCNT D, S	111110 001i 1111 ddddddddd sssssssss	Resultado = 0	Acarreo sin signo	Escrito	5+
WAITPEQ D, S	111100 000i 1111 ddddddddd sssssssss	Resultado = 0	---	No Escrito	5+
WAITPNE D, S	111101 000i 1111 ddddddddd sssssssss	Resultado = 0	---	No Escrito	5+
WAITVID D, S	111111 000i 1111 ddddddddd sssssssss	Resultado = 0	---	No Escrito	5+
WRBYTE D, S	000000 000i 1111 ddddddddd sssssssss	---	---	No Escrito	7..22 [1]
WRLONG D, S	000010 000i 1111 ddddddddd sssssssss	---	---	No Escrito	7..22 [1]
WRWORD D, S	000001 000i 1111 ddddddddd sssssssss	---	---	No Escrito	7..22 [1]
XOR D, S	011011 001i 1111 ddddddddd sssssssss	Resultado = 0	Paridad de resultado	Escrito	4

Notas para Tabla Maestra

Nota 1: Ciclos de Reloj para Instrucciones de Hub

Las instrucciones d Hub requieres de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al cog y el momento de la ejecución. El hub proporciona una ventana de acceso al cog cada 16 ciclos. Debido a que cada cog corre independientemente del hub, debe estar sincronizado al hub cuando se ejecuta una instrucción de hub. La primer instrucción hub en una secuencia tomara de 0 a 15 ciclos para sincronizar la ventana de acceso del hub, y 7 ciclos posteriormente para ejecutar; por lo tanto 7 a 22 (15 + 7) ciclos de reloj para ejecutar. Después de la primer instrucción de hub habrá 9 (16-7) ciclos libres antes de que el siguiente acceso al hub llegue para ese cog; suficiente tiempo para ejecutar dos instrucciones de 4 ciclos sin dejar pasar el siguiente acceso al hub. Para minimizar el desperdicio de ciclos, puedes insertar dos instrucciones normales entre cualquier instrucción de hub siguiente sin que se incremente el tiempo de ejecución. Tenga cuidado ya que las instrucciones pueden ocasionar tiempos indeterminados; particularmente la primer instrucción en secuencia.

Nota 2: Ciclos de Reloj para Modificar Subdivisión de Instrucciones

Las instrucciones que modifican un valor y posiblemente saltan, basados en un resultado, requieren un monto diferente de ciclos d reloj dependiendo si el salto es necesario o no. Estas instrucciones tomas 4 ciclos de reloj si se requiere el salto y 8 ciclos de reloj si no se requiere el salto. Como los ciclos que inicializan estas instrucciones típicamente necesitan ser rápidos, están optimizados de esta forma para velocidad.

Nota 3: `ADDABS` Salida C: Si S es negativo, C = al inverso no signado *borrow* (para D-S).

Nota 4: `SUBABS` Salida C: If S es negativo, C = al inverso no signado *carry* (para D+S).

ABS

instrucción: Obtiene el valor absoluto de un numero.

ABS *AValue*, ⟨#⟩ *SValue*

Resultado: Absoluto *SValue* se almacena en *AValue*.

- *AValue* (campo-d) es el registro en el cual se escribe el valor absoluto de *SValue*.
- *SValue* (campo-s) es un registro o un literal de 9-bit cuyo valor absoluto se escribe en *AValue*.

Tabla Opcode :

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101010 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] The smallest negative number (-2,147,483,648) has no corresponding positive value in 32-bit two's-complement math.

Explicación

ABS toma el valor absoluto de *SValue* y escribe el resultado en *AValue*.

Si se especifica el efecto WZ, se activa la bandera Z (1) si *SValue* es cero. Si el efecto WC se especifica, la bandera C se activa (1) si *SValue* es negativo, o se limpia (0) si *SValue* es positivo. El resultado se escribe en *AValue* a menos que s especifique el efecto NR.

Literales *SValues* son cero-extendidos, así ABS es realmente usado con registros *SValues*.

ABSNEG

instrucción: Obtiene el negativo de un numero de valor absoluto.

ABSNEG *NValue*, ⟨#⟩ *SValue*

Resultado: Negativo absoluto de *SValue* se almacena en *NValue*.

- *NValue* (campo-d) es el registro en el cual se escribe el negativo del valor absoluto *SValue*.
- *SValue* (campo-s) es un registro o un literal de 9-bit cuyo valor absoluto negado será escrito en *NValue*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101011 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	1

Explicación

ABSNEG niega el valor absoluto de *SValue* y escribe el resultado en *NValue*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *SValue* es cero. Si el efecto WC se especifica, la bandera C se activa (1) si *SValue* es negativo, o se limpia () si *SValue* es positivo. El resultado se escribe en *NValue* a menos que se especifique el efecto NR.

Literales *SValues* son cero-extendidos, así que ABSNEG se usa realmente con registros *SValues*.

ADD

instrucción: Suma dos valores no signados.

ADD *Value1*, ⟨#⟩ *Value2*

Resultado: Suma de no signado *Value1* y no signado *Value2* y se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a sumar al *Value2* y es el destino en el cual se escribe el resultado.
- *Value2* (campo-s) es un registro o un literal de 9-bit cuyo valor se suma en *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100000 001i 1111 ddddddddd sssssssss	D + S = 0	no signado Carry	Written	4

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino	Z	C
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFE; 4,294,967,294	$0000_0002; 2	-	-	wz wc	$0000_0000; 0	1	1
$FFFF_FFFE; 4,294,967,294	$0000_0003; 3	-	-	wz wc	$0000_0001; 1	0	1

[1] Ambos fuente y destino se tratan como valores no signados.

Explicación

ADD suma los dos valores no signados de *Value1* y *Value2* juntos y almacena el resultado en el registro *Value1*.

Si el efecto WZ se especifica, la bandera Z se activa (1) si el valor de *Value1* + *Value2* es igual a cero. Si el efecto WC se especifica, la bandera C se activa (1) si el resultado de la suma en un acarreo no signado (sobre flujo 32-bit). El resultado se escribe en *Value1* a menos que se especifique el efecto NR.

Para sumar valores no signados, multi-long, use ADD seguido de ADDX. Ver ADDX en Pág. 270 para mayor información.

ADDABS

instrucción: Suma un valor absoluto a otro valor.

ADDABS *Value*, ⟨#⟩ *SValue*

Resultado: La suma de *Value* y valor absoluto no signado de *SValue* se almacena en *Value*.

- *Value* (campo-d) es el registro que contiene el valor a sumar con el valor absoluto de *SValue* y es el destino dond se escribe el resultado.
- *SValue* (campo-s) es un registro o literal de 9-bit cuyo valor absoluto se suma en *Value*. Literales *SValues* son cero extendidos (siempre valores positivos) así ADDABS es mejor usado con registros *SValues*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100010 001i 1111	ddddddddd	sssssssss	D + |S| = 0	no signado Carry[1]	Written	4

[1] If S is negative, C Result is the inverse of unsigned *borrow* (for D - S).

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino	Z	C
$FFFF_FFFD; 4,294,967,293	$0000_0004; 4	-	-	wz wc	$0000_0001; 1	0	1
$FFFF_FFFD; 4,294,967,293	$0000_0003; 3	-	-	wz wc	$0000_0000; 0	1	1
$FFFF_FFFD; 4,294,967,293	$0000_0002; 2	-	-	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFE; 4,294,967,294	0	1
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFE; -2	-	-	wz wc	$FFFF_FFFF; 4,294,967,295	0	1
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFD; -3	-	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFD; 4,294,967,293	$FFFF_FFFC; -4	-	-	wz wc	$0000_0001; 1	0	0

[1] Destino se trata como valor no signado.

Explicación

ADDABS suma *Value* junto con el valor absoluto de *SValue* y almacena el resultado en el registro *Value*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value* + |*SValue*| es igual a cero. Si el efecto WC se especifica, la bandera C se activa (1) si la suma resultante es un acarreo no signado (sobre flujo 32-bit). El resultado se escribe a *Value* a menos que se especifique NR.

ADDS

instrucción: Suma dos valores signados.

ADDS *SValue1*, ⟨#⟩ *SValue2*

Resultado: Suma de sigando*SValue1* y signado *SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor a sumar con *SValue2* y es el destino en el cual se escribe el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor se suma en *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110100 001i 1111 ddddddddd sssssssss	D + S = 0	signado Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$FFFF_FFFF; -1	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0002; 2	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$FFFF_FFFF; -1	-	-	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	-	-	wz wc	$FFFF_FFFF; -1	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	-	-	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	-	-	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

Explicación

ADDS suma los dos valores signados de *SValue1* y *SValue2* juntos y almacena el resultado en el registro *SValue1*.

Si el efecto WZ se especifico, la bandera Z se activa (1) si *SValue1* + *SValue2* son cero. Si se especifica el efecto WC la bandera C se activa (1) si el resultado es un sobre flujo signado. El resultado se escribe a *SValue1* a menos que se especifique el efecto NR.

Para sumar valores multi-long signados use ADD, posiblemente ADDX, y finalmente ADDSX. Ver ADDSX en Pág. 268 para mayor información.

ADDSX

instrucción: Suma dos valores signados mas C.

ADDSX *SValue1*, ⟨#⟩ *SValue2*

Result: Suma de *SValue1* y *SValue2* signados mas la bandera C se almacenan en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor a sumar con *SValue2* mas C, y es el destino donde se escribe el resultado.
- *SValue2* (campo-s) es un registro o literal de 9-bit cuyo valor mas C se suman en *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110110 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	signado Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$FFFF_FFFE; -2	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFE; -2	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFE; -2	x	0	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$FFFF_FFFE; -2	0	1	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$FFFF_FFFE; -2	1	1	wz wc	$0000_0000; 0	1	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	x	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$0000_0001; 1	x	1	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFE; 2,147,483,646	$0000_0002; 2	x	0	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	x	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0001; -2,147,483,647	$FFFF_FFFE; -2	x	1	wz wc	$8000_0000; -2,147,483,648	0	0

Explicación

ADDSX (Add signado, Extended) suma los dos valores signados de *SValue1* y *SValue2* mas C, y almacena el resultado en el registro *SValue1*. La instrucción ADDSX se usa para desarrollar sumas signadas multi-long ; sumas de 64-bit, por ejemplo.

En una operación signada multi-long, la primer instrucción es no signada (Ej.: ADD), cualquier instrucción intermedia son no signadas, extendidas (Ej.: ADDX), y la ultima instrucción es signada, extendida (Ej.: ADDSX). Asegúrese de usar los efectos WC, y opcionalmente WZ, en las instrucciones ADD y ADDX.

Por ejemplo, una suma signada doble-long (64-bit)Por ejemplo, un signado doble-long (64-bit) podría verse de la siguiente forma:

```
add     XLow,  YLow   wc wz   'Suma longs bajos; guarda C y Z
addsx   XHigh, YHigh          'Suma longs altos
```

Después de ejecutar el código, el resultado doble long (64-bit) esta en los registros long *XHigh:XLow*. Si *XHigh:XLow* inicio como $0000_0001:0000_0000 (4,294,967,296) y *YHigh:YLow* fue $FFFF_FFFF:FFFF_FFFF (-1) el resultado en *XHigh:XLow* deberá ser $0000_0000:FFFF_FFFF (4,294,967,295). Esto se demuestra abajo.

```
                        Hexadecimal              Decimal
                      (alto)     (bajo)
   (XHigh:XLow)     $0000_0001:0000_0000       4,294,967,296
 + (YHigh:YLow)   + $FFFF_FFFF:FFFF_FFFF     +            -1
                  ---------------------     ---------------
                  = $0000_0000:FFFF_FFFF     = 4,294,967,295
```

Una suma signada de triple-long (96-bit) se vera similar pero con una instrucción ADDX insertada entre las instrucciones ADD y ADDSX:

```
add     XLow,  YLow   wc wz   'Suma longs bajos; guarda C y Z
addx    XMid,  YMid   wc wz   'Suma longs medios; guarda C y Z
addsx   XHigh, YHigh          'Suma longs altos
```

Por supuesto puede ser necesario especificar los efectos WC y WZ en la instrucion final, ADDSX, para ver el resultado del cero o la condición de sobre flujo signada. Observe que durante esta operación multi paso la bandera Z siempre indica si el resultado es cero, pero la bandera C indica un acarreamiento no signado hasta el final de la instrucción, ADDSX, en el cual se indica en sobre flujo signado.

Para ADDSX, si el efecto WZ se especifica, la bandera Z se activa (1) si Z se activo previamente y *SValue1* + *SValue2* + C iguala a cero (use WC y WZ antes de las instrucciones ADD y ADDX). Si el efecto WC se especifica, la bandera C se activa (1) si el resultado d la suma es un sobre flujo signado. El resultado se escribe a *SValue1* a menos que se especifique el efecto NR.

ADDX

instrucción: Suma dos valore no signados mas C.

ADDX *Value1*, ⟨#⟩ *Value2*

Resultado: La suma de un valor *Value1* y *Value2* no signado mas C se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a sumar con *Value2* mas C, y es el destino en el cual se escribe el resultado.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor mas C se suman en *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110010 001i 1111 ddddddddd sssssssss	Z & (D+S+C = 0)	no signado Carry	Written	4

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino	Z	C
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	x	0	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	0	1	wz wc	$0000_0000; 0	0	1
$FFFF_FFFE; 4,294,967,294	$0000_0001; 1	1	1	wz wc	$0000_0000; 0	1	1

[1] Ambas fuente y destino se tratan como valores no signados.

Explicación

ADDX (Add Extended) suma los dos valores no signados de *Value1* y *Value2* mas C, y almacena el resultado en el registro *Value1*. La instrucción ADDX se usa para desarrollar sumas multi-long ; sumas de 64-bit, por ejemplo.

En una operación multi-long, la primer instrucción es no signada (Ej.: ADD), cualquier instrucción media son no signadas, extendidas (Ej.: ADDX), y la ultima instrucción es no signada, extendida (ADDX) o signada, extendida (ADDSX) dependiendo de la naturaleza de los valores multi-long originales. Discutiremos los valores no signados multi-long aqui; ver ADDSX en Pág. 268 para ejemplos con valores multi-long signados. Asegúrese de usar WC, y opcionalmente WZ, en las instrucciones ADD y ADDX.

Por ejemplo, una suma doble long no signada (64-bit) podría verse así:

```
add     XLow,  YLow   wc wz    'Suma longs bajos; guarda C y Z
addx    XHigh, YHigh          'Suma longs altos
```

Después de ejecutar el código, el resultado del doble long (64-bit) esta en los registros long *XHigh:XLow*. Si *XHigh:XLow* inicio como $0000_0000:FFFF_FFFF (4,294,967,295) y *YHigh:YLow* como $0000_0000:0000_0001 (1) del resultado en *XHigh:XLow* será $0000_0001:0000_0000 (4,294,967,296). Esto se demuestra a continuación:

```
                          Hexadecimal                Decimal
                        (alto)     (bajo)
   (XHigh:XLow)      $0000_0000:FFFF_FFFF        4,294,967,295
 + (YHigh:YLow)    + $0000_0000:0000_0001      +             1
                   ----------------------      ---------------
                   = $0000_0001:0000_0000      = 4,294,967,296
```

Por supuesto será necesario especificarlos efectos **WC** y **WZ** en la instrucción final, **ADDX**, para poder observar el resultado de cero o la condición de sobre flujo no signada.

Para **ADDX**, si el efecto **WZ** se especifico, la bandera Z se activa (1) si Z se activo previamente y *Value1* + *Value2* + C son igual a cero (use **WC** y **WZ** en las instrucciones **ADD** y **ADDX**). Si el efecto **WC** se especifico, la bandera C se activa (1) si el resultado de la suma es un acarreamiento no signado (sobre flujo 32-bit). El resultado se escribe en *Value1* a menos que se especifique el efecto **NR**.

AND

instrucción: Bitwise, AND de dos valores.

AND *Value1*, ⟨#⟩ *Value2*

Resultado: *Value1* AND *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor que se hace AND con *Value2* y es el destino en el cual se escribe el resultado.
- *Value2* (campo-s) es un registro o literal de 9-bit cuyo valor es el resultado AND con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011000 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_000A; 10	$0000_0005; 5	-	-	wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_0007; 7	-	-	wz wc	$0000_0002; 2	0	1
$0000_000A; 10	$0000_000F; 15	-	-	wz wc	$0000_000A; 10	0	0

Explicación

AND (bitwise AND) genera una operación AND del valor *Value2* con el que esta en *Value1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* AND *Value2* son igual a cero. Si se especifica el efecto WC , la bandera C se activa(1) si el resultado contiene un numero non de bits altos (1). El resultado se escribe en *Value1* a menos que el efecto NR se especifique.

ANDN

instrucción: Bitwise, AND de un valor con el NOT de otro.

ANDN *Value1*, ⟨#⟩ *Value2*

Resultado: *Value1* AND !*Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a hacer AND con el !*Value2* y es el destino donde se escribe el resultado.
- *Value2* (campo-s) es un registro o literal de 9-bit cuto valor es el inverso (bitwise NOT) y se hace AND con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011001 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$F731_125A; -147,778,982	$FFFF_FFFA; -6	-	-	wz wc	$0000_0000; 0	1	0
$F731_125A; -147,778,982	$FFFF_FFF8; -8	-	-	wz wc	$0000_0002; 2	0	1
$F731_125A; -147,778,982	$FFFF_FFF0; -16	-	-	wz wc	$0000_000A; 10	0	0

Explicación

ANDN (bitwise AND NOT) genera una operación AND con el valor inverso (bitwise NOT) del *Value2* en el *Value1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* AND !*Value2* son igual a cero. Si el efecto WC se especifica, la bandera C se activa (1) si el resultado contiene un numero impar de bits altos (1). El resultado se escribe en *Value1* a menos que se especifique el efecto NR.

CALL

instrucción: Salta a una dirección con la intención de regresar a la siguiente instrucción.

CALL #*Symbol*

Resultado: PC + 1 se escribe en el campo-s del registro indicado por el campo-d.

- *Symbol* (campo-s) es una literal de 9-bit cuyo valor es la dirección a saltar. Este campo debe contener un símbolo DAT especificado como literal (#symbol) y el correspondiente código deberá eventualmente ejecutar una instrucción RET etiquetada con el mismo símbolo mas un sufijo de "_ret" (*Symbol*_ret RET).

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010111 0011 1111 ????????? sssssssss	Result = 0	---	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C[2]
$----_----; -	$----_----; -	-	-	WZ WC	31:9 unchanged, 8:0 = PC+1	0	1

[1] El campo-s del registro destino (9 bits mas bajos) se sobrescriben con la dirección de regreso (PC+1) en la ejecución.
[2] La bandera C se activa (1) a menos que PC+1 sea igual a 0; muy improbable ya que requiere que CALL se ejecute desde el inicio de la RAM del cog RAM ($1FF; registro de propósito especial VSCL).

Explicación

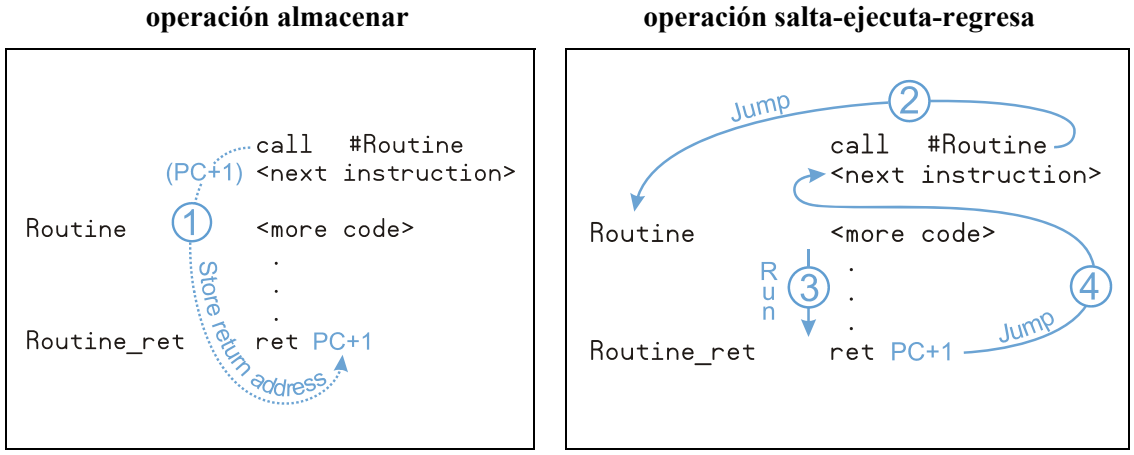
CALL graba la dirección de la siguiente instrucción (PC + 1) y salta a *Symbol*. La rutina en *Symbol* deberá eventualmente ejecutar una instrucción RET para regresar a la dirección grabada (PC+1; la instrucción que sigue de CALL). Para compilar y correr efectivamente CALL, la rutina de *Symbol* RET debe estar etiquetada en la forma de *Symbol* con el "_ret" integrado. La razón para esto se explica a continuación.

El hardware propeller no usa llamada a pila, así que la dirección de regreso debe almacenarse de forma diferente. En tiempo de compilación el ensamblador localiza la rutina destino así como la instrucción RET (etiquetadas *Symbol* y *Symbol*_ret, respectivamente) y codifica esas direcciones en el campo-s y campo-d de la instrucción CALL. Esto proporciona al conocimiento a la instrucción CALL de ambos datos, a donde ir y a donde regresara exactamente.

En tiempo de ejecución la que hace la instrucion **CALL** es almacenar la instrucción de regreso (PC+1) en la localidad de donde regresara; el lugar de la instrucción "*Symbol*_ret **RET**". La instrucción **RET** es solo una instrucción **JMP** sin una dirección destino codificada, y esta acción de tiempo de ejecución proporciona la dirección de regreso a la cual se moverá. Después de almacenar la dirección de regreso, **CALL** salta a la dirección destino; *Symbol*.

El diagrama de abajo usa un programa corto para demostrar el comportamiento de la instrucción **CALL** en tempo de ejecución; la operación almacenar (izquierda) y la operación salta-ejecuta-regresa (derecha).

Figura 3-1: Procedimiento CALL en Tiempo de Ejecución



En este ejemplo ocurre lo siguiente cuando la instrucción **CALL** se realiza en tiempo de ejecución:

① El cog almacena la dirección de regreso (PC+1; `<next instruction>`) en la fuente (campo-s) del registro en `Routine_ret` (ver imagen izquierda).

② El cog salta a `Routine` (ver imagen derecha).

③ Las instrucciones `Routine` se ejecutan, eventualmente llevan a la línea `Routine_ret`.

④ Como la localidad `Routine_ret` contiene una instrucción **RET** con una fuente actualizada (campo-s), la cual es la dirección de regreso escrita en el paso 1, regresa o salta a la línea `<next instruction>`.

Esta naturaleza de la instrucción CALL dicta lo siguiente:

- La rutina referenciada debe tener solo una instrucción RET asociada. Si la rutina necesita mas de un punto de salida, hacer uno de esos puntos de salida con la instrucción RET y hacer los otros puntos de salida con subdivisiones (Ej.:, JMP) a esa instrucion RET.
- La rutina referenciada no puede ser recursiva. Haciendo una llamada anidada a la rutina se sobrescribirá la dirección de regreso de la llamada previa.

CALL es realmente un subgrupo de la instrucción JMPRET; de hecho, es el mismo opcode de JMPRET pero con el campo-I activo (como CALL usa un valor inmediato solamente) y el campo-d activa el ensamblador a la dirección de la etiqueta llamada *Symbol*_ret.

la dirección de regreso (PC + 1) se escribe a la fuente (campo-s) del registro *Symbol*_ret a menos que el efecto NR se especifique. Por supuesto, especificar NR no se recomienda para la instrucción CALL ya que regresa con una instrucción JMP, o RET.

CLKSET

instrucción: Activa el modo clock en tiempo de ejecución.

CLKSET *Mode*

- *Mode* (campo-d) registro que contiene el patrón de 8-bit a escribir en el registro CLK.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111 ddddddddd ------000	---	---	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2]	Z	C
$0000_006F; 111	%0_00000000; 0	-	-	wr wz wc	$0000_0007; 7	0	0

[1] La fuente se activa automáticamente al valor inmediato 0 por el ensamblador para indicar que esta es una instrucción de hub CLKSET.

[2] El destino no se escribe a menos que se de el efecto WR.

Explicación

CLKSET cambia el modo del reloj del sistema durante la ejecución. La instrucción CLKSET se comporta similar al comando Spin del mismo nombre (ver CLKSET en Pág. 74) excepto que solo activa el modo clock, no la frecuencia.

Después de proporcionar una instrucción CLKSET, es importante actualizar el valor de la frecuencia del reloj del sistema escribiendo a su localidad en RAM principal (long 0): WRLONG freqaddr, #0. Si el valor de frecuencia del reloj del sistema no se actualiza, otros objetos no se comportaran adecuadamente debido al dato invalido del reloj de frecuencia.

CLKSET es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al hub y a instrucción al momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

CMP

instrucción: Compara dos valores no signados.

CMP *Value1*, ⟨#⟩ *Value2*

Resultado: opcionalmente, estado de igualdad y mayor que/menor que se escriben en Z y C.

- *Value1* (campo-d) es el registro que contiene el valor a comparar con *Value2*.
- *Value2* (campo-s) es un registro o literal de 9-bit cuyo valor se compara con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100001 000i 1111 ddddddddd sssssssss	D = S	no signado (D < S)	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino[2]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$8000_0000; 2,147,483,648	$7FFF_FFFF; 2,147,483,647	–	–	wr wz wc	$0000_0001; 1	0	0[4]
$7FFF_FFFF; 2,147,483,647	$8000_0000; 2,147,483,648	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1[4]
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	–	–	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	–	–	wr wz wc	$0000_0001; 1	0	0

[1] fuente y destino se tratan como valores no signados.

[2] El destino no se escribe a menos que se de el efecto **WR**.

[3] Destino Salida (destino escrito) puede ser signado o no signado; se muestra aqui como signado para propósitos de demostración solamente.

[4] La bandera C resulta de CMP (Compare no signado) puede diferir de CMPS (Compare signado) donde el "signo interpretado" de la fuente y destino son opuestos. El primer ejemplo en el segundo grupo, arriba, muestra que CMP limpia C porque es no signado $8000_0000 (2,147,483,648) no es menor que el no signado $7FFF_FFFF (2,147,483,647). CMPS, sin embargo debería activar C porque el signado $8000_0000 (-2,147,483,648) es menor que el signado $7FFF_FFFF (2,147,483,647). El segundo ejemplo es el caso complementario donde fuente y destino se cambian.

Explicación

CMP (Compare no signado) compara los valores no signados de *Value1* y *Value2*. Las banderas Z y C, si están escritas, indican el igual relativo, y relación mayor o menor entre los dos.

Si el efecto WZ se especifica, la bandera Z se activa (1) si *Value1* es igual a *Value2*. Si se especifica el efecto WC, la bandera C se activa (1) I *Value1* es menor que *Value2*.

El resultado no se escribe a menos que se especifique el efecto WR. Si s especifica WR, CMP se convierte exactamente en una instrucción SUB.

Para comparar valores multi-long no signados, use CMP seguido de CMPX. Ver CMPX en Pág. 286 para mayor información.

CMPS

instrucción: Compara dos valores signados.

CMPS *SValue1*, ⟨#⟩ *SValue2*

Resultado: Opcionalmente, igualdad y estado mayor/menor que se escribe en Z y C.

- *SValue1* (campo-d) es el registro que contiene el valor a comparar con *SValue2*.
- *SValue2* (campo-s) es el registro o literal 9-bit cuyo valor se compara con *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110000 000i 1111 ddddddddd sssssssss	D = S	signado (D < S)	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$8000_0000; -2,147,483,648	$7FFF_FFFF; 2,147,483,647	–	–	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2,147,483,647	$8000_0000; -2,147,483,648	–	–	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2,147,483,648	$0000_0001; 1	–	–	wr wz wc	$7FFF_FFFF; 2,147,483,647[3]	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	–	wr wz wc	$8000_0000; -2,147,483,648[3]	0	0
$FFFF_FFFE; -2	$FFFF_FFFF; -1	–	–	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	–	–	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	–	–	wr wz wc	$0000_0001; 1	0	0

[1] El destino no se escribe a menos que se den efectos WR.

[2] El resultado de la bandera C de **CMPS** (Compare signado) puede diferir de **CMP** (Compare no signado) donde el signo interpretado de la fuente y destino son opuestos. El primer ejemplo en el segundo grupo, muestra que **CMPS** activa C porque el signado $8000_0000 (-2,147,483,648) es menor que el signado $7FFF_FFFF (2,147,483,647). **CMP**, sin embargo, limpiara C debido a que el no signado $8000_0000 (2,147,483,648) no es menor que el no signado$7FFF_FFFF (2,147,483,647). El segundo ejemplo es el caso complementario donde la fuente y destino se intercambian.

[3] El ejemplo del tercer grupo, arriba, demuestra casos donde la comparación se refleja apropiadamente en las banderas pero la salida de destino cruzo la frontera del signo (error de sobre flujo de signo) en cualquiera de las direcciones negativo o positivo. Esta condición de sobre flujo signado no puede reflejarse en las banderas. Si esta condición es importante en alguna aplicación, desarrolle un **CMPS** sin efecto **WR**, observe el estado de C (signado borrow), luego desarrolle un **SUBS** y observe el estado de C (signado overflow).

Explicación

CMPS (Compare signado) compara los valores signados de *SValue1* y *SValue2*. Las banderas Z y C, si se escriben, indican la relación relativa de igual, mayor o menor que entre los dos.

Si se especifica el efecto WZ la bandera Z se activa (1) si *SValue1* es igual a *SValue2*. Si el efecto WC se especifica, la bandera C se activa (1) si *SValue1* es menor que *SValue2*.

Para comparar valores signados multi-long, en vez de usar CMPS, use CMP, posiblemente CMPX, y finalmente CMPSX. Ver CMPSX en Pág. 283 para mayor información.

CMPSUB

instrucción: Compara dos valores no signados y resta el segundo si es menor o igual.

CMPSUB *Value1*, ⟨#⟩ *Value2*

Resultado: Opcionalmente, *Value1* = *Value1*–*Value2*, y Z y C = resultado comparación.

- *Value1* (campo-d) es el registro que contiene el valor a comparar con *Value2* y es el destino en el cual se escribe el resultado si se desarrolla una resta.
- *Value2* (campo-s) es un registro o literal de 9-bit cuyo valor se compara y posiblemente se resta de *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111000 001i 1111 ddddddddd sssssssss	D = S	no signado (D => S)	Written	4

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino	Z	C
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$0000_0003; 3	0	0

[1] Fuente y destino se tratan como valores no signados

Explicación

CMPSUB compara los valores no signados de *Value1* y *Value2*, y si *Value2* es igual o menor que *Value1* entonces se resta de *Value1*.

Si se activo el efecto WZ, la bandera Z se activa (1) si *Value1* es igual a *Value2*. Si se activo el efecto WC, la bandera C se activa (1) si una resta es posible (*Value1* es igual o menor que *Value2*). El resultado, si existe, se escribe a *Value1* a menos que se especifique NR.

CMPSX

instrucción: Compara dos valores signados mas C.

CMPSX *SValue1*, ⟨#⟩ *SValue2*

Resultado: Opcionalmente, estado de igualdad, mayor/menor se escribe en Z y C.

- *SValue1* (campo-d) es el registro que contiene el valor a comparar con *SValue2*.
- *SValue2* (campo-s) es el registro o literal de 9-bit cuyo valor se copiara con *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110001 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	signado (D < S+C)	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0003; 3	$0000_0002; 2	×	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	×	1	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	×	0	wr wz wc	$FFFF_FFFF; -1	0	1
$0000_0003; 3	$0000_0004; 4	×	1	wr wz wc	$FFFF_FFFE; -2	0	1
$8000_0000; -2,147,483,648	$7FFF_FFFF; 2,147,483,647	0	0	wr wz wc	$0000_0001; 1	0	1[2]
$7FFF_FFFF; 2,147,483,647	$8000_0000; -2,147,483,648	0	0	wr wz wc	$FFFF_FFFF; -1	0	0[2]
$8000_0000; -2,147,483,648	$0000_0001; 1	0	0	wr wz wc	$7FFF_FFFF; 2,147,483,647[3]	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	0	wr wz wc	$8000_0000; -2,147,483,648[3]	0	0
$FFFF_FFFE; -2	$FFFF_FFFF; -1	×	0	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFF; -1	×	1	wr wz wc	$FFFF_FFFE; -2	0	1
$FFFF_FFFE; -2	$FFFF_FFFE; -2	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; -2	$FFFF_FFFE; -2	×	1	wr wz wc	$FFFF_FFFF; -1	0	1
$FFFF_FFFE; -2	$FFFF_FFFD; -3	×	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; -2	$FFFF_FFFD; -3	1	1	wr wz wc	$0000_0000; 0	1	0

[1] El destino no se escribe a menos que se de el efecto **WR**.

[2] El resultado de la bandera C de **CMPS** (Compare signado) puede diferir de **CMP** (Compare no signado) donde el signo interpretado de la fuente y destino son opuestos. El primer ejemplo en el segundo grupo, muestra que **CMPS** activa C porque el signado $8000_0000 (-2,147,483,648) es menor que el signado $7FFF_FFFF (2,147,483,647). **CMP**, sin embargo, limpiara C debido a que el no signado $8000_0000 (2,147,483,648) no es menor que el no signado$7FFF_FFFF (2,147,483,647). El segundo ejemplo es el caso complementario donde la fuente y destino se intercambian.

[3] El ejemplo del tercer grupo, arriba, demuestra casos donde la comparación se refleja apropiadamente en las banderas pero la salida de destino cruzo la frontera del signo (error de sobre flujo de signo) en cualquiera de las direcciones negativo o positivo. Esta condición de sobre flujo signado no puede reflejarse en las banderas. Si esta condición es importante en alguna aplicación, desarrolle un **CMPS** sin efecto **WR**, observe el estado de C (signado borrow), luego desarrolle un **SUBS** y observe el estado de C (signado overflow).

Explicación

CMPSX (Compare signado, Extended) compara los valores signados de *SValue1* y *SValue2* mas C. Las banderas Z y C , si se escribieron, indican la relación relativa igual, mayor o menor entre los dos. La instrucción **CMPSX** se usa para desarrollar comparaciones signadas multi-long ; comparaciones de 64-bit, por ejemplo.

En una operación multi-long signada, la primera instrucción es no signada (Ej.: **CMP**), cualquier instrucción intermedia son no signadas, extendidas (Ej.: **CMPX**), y la ultima instrucción es signada, extendida (Ej.: **CMPSX**). Asegúrese de usar **WC**, y opcionalmente **WZ**, en todas las operaciones de comparación.

por ejemplo, una comparación doble long signado (64-bit) deberá verse así:

```
cmp     XLow,  YLow   wc wz   'Compara long bajo; guarda C y Z
cmpsx   XHigh, YHigh  wc wz   'Compara long alto; guarda C y Z
```

Después de ejecutar lo anterior, las banderas Z y C indican la relación entre los dos valores doble-long (64-bit). Si *XHigh:XLow* inicio como $FFFF_FFFF:FFFF_FFFF (-1) y *YHigh:YLow* era $0000_0000:0000_0001 (1) el resultado de las banderas será: Z = 0 y C = 1; (Value1 < Value2). Esto se demuestra abajo. Note que la comparación es realmente solo una resta y el resultado no se escribe; los resultados de las banderas Z y C son importantes.

```
                          Hexadecimal          Decimal        Banderas
                        (alto)     (bajo)
  (XHigh:XLow)      $FFFF_FFFF:FFFF_FFFF              -1      n/a
- (YHigh:YLow)    - $0000_0000:0000_0001    -          1      n/a
                   ----------------------    ----------    --------
                   = $FFFF_FFFF:FFFF_FFFE    =         -2    Z=0, C=1
```

Una comparación de un triple-long signado (96-bit) deberá verse similar pero con la instrucción **CMPX** insertada entre las instrucciones **CMP** y **CMPSX**:

```
cmp     XLow,  YLow   wc wz   'Compare low longs; save C y Z
cmpx    XMid,  YMid   wc wz   'Compare middle longs; save C y Z
cmpsx   XHigh, YHigh  wc wz   'Compare high longs; save C y Z
```

Para **CMPSX**, si se especifico el efecto **WZ**, la bandera Z se activa (1) si Z se activo previamente y *SValue1* es igual a *SValue2* + C (use **WC** y **WZ** en las instrucciones **CMP** y **CMPX**). Si se especifica el efecto **WC**, la bandera C se activa (1) si *SValue1* es menor que *SValue2* (como valores multi-long).

CMPX

instrucción: Compara dos valores no signados mas C.

CMPX *Value1*, ⟨#⟩ *Value2*

Resultado: Opcionalmente, el estado de igualdad, menor/mayor se escribe en Z y C.

- *Value1* (campo-d) es el registro que contiene el valor a comparar con *Value2*.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor es comparado con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110011 000i 1111 ddddddddd sssssssss	Z & (D = S+C)	no signado (D < S+C)	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino[1]	Fuente[1]	Z	C	Efectos	Destino[2]	Z	C
$0000_0003; 3	$0000_0002; 2	x	0	wr wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0002; 2	0	1	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0002; 2	1	1	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	0	0	wr wz wc	$0000_0000; 0	0	0
$0000_0003; 3	$0000_0003; 3	1	0	wr wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0003; 3	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$0000_0003; 3	$0000_0004; 4	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$8000_0000; 2,147,483,648	$7FFF_FFFF; 2,147,483,647	0	0	wr wz wc	$0000_0001; 1	0	0[4]
$7FFF_FFFF; 2,147,483,647	$8000_0000; 2,147,483,648	0	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1[4]
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	x	0	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFF; 4,294,967,295	x	1	wr wz wc	$FFFF_FFFE; -2[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	0	0	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	1	0	wr wz wc	$0000_0000; 0	1	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFE; 4,294,967,294	x	1	wr wz wc	$FFFF_FFFF; -1[3]	0	1
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	x	0	wr wz wc	$0000_0001; 1	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	0	1	wr wz wc	$0000_0000; 0	0	0
$FFFF_FFFE; 4,294,967,294	$FFFF_FFFD; 4,294,967,293	1	1	wr wz wc	$0000_0000; 0	1	0

[1] Fuente y destino son tratados como valores no signados.
[2] El destino no se escribe a menos que se de el efecto WR.
[3] Salida destino (escrita destino) puede ser signado o no signado; se muestra aqui como signao para propósitos de demostración solamente.

[4] El resultado de la bandera C **CMPX** (Compare no signado, Extended) puede diferir de **CMPSX** (Compare signado, Extended) donde el "signo interpretado" de la fuente y destino son opuestos. El primer ejemplo en el segundo grupo, arriba, muestra que **CMPX** limpia C porque el no signado $8000_0000 (2,147,483,648) no es menor que el no signado $7FFF_FFFF (2,147,483,647). **CMPSX**, sin embargo, tendrá que activar C porque el signado $8000_0000 (2,147,483,648) es menor que el signado $7FFF_FFFF (2,147,483,647). El segundo ejemplo es el caso complementario donde los valores fuente y destino se cambian. Observe que los ejemplos con diferente Z y C no se muestran pero tienen efectos similares a los otros ejemplos.

Explicación

CMPX (Compare Extended) compara los valores no signados de *Value1* y *Value2* mas C. Las banderas Z y C, si están escritas indican la relación relativa igual, mayor o menor entre las dos. La instrucción **CMPX** se usa para desarrollar comparaciones multi-long; comparaciones de 64 bit, por ejemplo.

En una operación multi-long, la primer instrucción es no signada (Ej.: **CMP**), cualquier instrucción intermedia es no signada, extendida (Ej.: **CMPX**), y la ultima instrucción es no signada, extendida (**CMPX**) o signada, extendida (**CMPSX**) dependiendo de la naturaleza de los valores originales multi-long. Vamos a discutir valores no signados multi-long aqui; ver **CMPSX** en Pág. 283 para ejemplos con valores multi-long signados. Asegúrese de usar efectos **WC**, y opcionalmente **WZ**, en todas las instrucciones en la operación de comparación.

Por ejemplo, una comparación doble-long no signado (64-bit) deberá verse como:

```
cmp     XLow,  YLow   wc wz   'Compara longs bajos; Guarda C y Z
cmpx    XHigh, YHigh  wc wz   'Compara longs altos
```

Después de ejecutar la instrucción la banderas C y Z indicaran la relación entre los dos valores dobles longs (64-bit). Si *XHigh:XLow* inicio como $0000_0001:0000_0000 (4,294,967,296) y *YHigh:YLow* fue $0000_0000:0000_0001 (1) el resultado de las banderas será: Z = 0 y C = 0; (Value1 > Value2). Esto se demuestra a continuación. Observe que la comparación es realmente solo una resta con el resultado no escrito; sin embrago el resultado de la bandera Z y C es importante.

```
                      Hexadecimal              Decimal        Banderas
                       (high)     (low)
  (XHigh:XLow)      $0000_0001:0000_0000     4,294,967,296       n/a
- (YHigh:YLow)    - $0000_0000:0000_0001   -             1       n/a
                   ----------------------   ---------------   --------
                   = $0000_0000:FFFF_FFFF   = 4,294,967,295   Z=0, C=0
```

Para **CMPX**, si el efecto **WZ** se especifico, la bandera Z se activa (1) si Z se activo previamente y *Value1* es igual a *Value2* + C (use **WC** y **WZ** en instrucciones **CMP** y **CMPX**). Si se especifico el efecto **WC**, la bandera C se activa (1) si *Value1* es menor que *Value2* (como valor multi-long).

CNT

Registro: Registro contador del sistema.

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   DestOperand, CNT ⟨Effects⟩
```

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* una condición de ejecución opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador deseada. CNT es in registro de solo lectura y por lo tanto solo se usa como operando fuente.
- *DestOperand* es una expresión constante indicando el registro en el que opera, y opcionalmente escrito, usando el valor de CNT en el operando fuente de la instrucción.

Explicación

El registro CNT contiene el valor actual en el contador global 32-bit del contador del sistema. El contador del sistema sirve como referencia de tiempo central para todos los cogs; incrementa su valor d 32-bit una vez cada ciclo del reloj del sistema.

CNT es un pseudo registro de solo lectura; cuando se usa como operando fuente de una instrucción, lee el valor actual del contador del sistema. No use CNT no use como operando destino; que solo resultaría en leer y modificar el registro sombra que ocupa la dirección CNT.

CNT se usa frecuentemente para determinar un valor objetivo inicial para un retraso basado en WAITCNT. El siguiente código desarrolla una operación en un ciclo cada ¼ de segundo. Ver Registros, Pág. 351, y lenguaje spin sección CNT, Pág. 76, para mas información.

```
DAT
                    org 0                       'reinicia apuntador
AsmCode             rdlong   Delay, #0          'Obtiene frec de reloj
                    shr      Delay, #2          'Divide por 4
                    mov      Time,  cnt         'Obtiene tiempo actual
                    add      Time,  Delay       'Ajusta 1/4 de segundo
Loop                waitcnt  Time,  Delay       'Espera 1/4 de segundo
                    '<more code here>           'Desarrolla operación
                    jmp      #Loop              'Ciclo de regreso

Delay     res       1
Time      res       1
```

COGID

instrucción: Obtiene el numero de Cog.

COGID *Destination*

Result: El numero de cog actual "ID" (0-7) se escribe en *Destination*.

- *Destination* (campo-d) es el registro en donde escribir el ID del cog.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0011 1111	ddddddddd	------001	ID = 0	0	Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2]	Z	C
$----_----; -	%0_00000001; 1	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000001; 1	-	-	wz wc	$1; 1 .. $7; 7	0	0

[1] La fuente se activa automáticamente al valor inmediato 1 por el ensamblador para indicar que es instrucción de hub **COGID**.
[2] Salida destino (escribe destino) será de 0 a 7, dependiendo del cog en el que se ejecute la instrucción.

Explicación

COGID regresa el ID del cog en el que se ejecuta la instrucción. La instrucción **COGID** se comporta similar a la instrucción spin del mismo nombre; ver **COGID** en Pág. 78.

Si se especifico el efecto **WZ** la bandera Z se activa si el ID del cog es cero. El resultado se escribe en *Destination* a menos que se especifique el efecto **NR**.

COGID es una instrucción de hub. Las instrucciones de hub requieren 7 a 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al cog y el momento en el que se ejecuta la instrucción. ver Hub en Pág. 24 para mayor información.

COGINIT

instrucción: Inicia o reinicia un cog, opcionalmente por ID, para correr ensamblador propeller o spin.

COGINIT *Destination*

Result: Opcionalmente, el ID del cog iniciado/reiniciado se escribe en *Destination*.

- *Destination* (campo-d) es el registro que contiene información de inicio para el cog objetivo y opcionalmente se convierte en el destino del ID del cog iniciado si un cog se inicia.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111 ddddddddd ------010	ID = 0	No Cog Free	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente [2]	Z	C	Efectos	Destino [3]	Z	C
$0000_0000; 0	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0 [4]	1	0
$0000_0001; 1	%0_00000010; 2	–	–	wr wz wc	$0000_0001; 1 [4]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0000; 0 [5]	1	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$1; 1 .. $7; 7 [5]	0	0
$0000_0008; 8	%0_00000010; 2	–	–	wr wz wc	$0000_0007; 7 [5]	0	1

[1] El destino debe ser un valor PAR (bits 31:18), una dirección código ensamblador (bits 17:4), y un indicador de cog nuevo (bits 3:0).
[2] La fuente es automáticamente activada al valor inmediato valor 2 por el ensamblador para indicar que esta es una instrucción de hub COGINIT.
[3] El destino no se escribe a menos que se de el efecto WR.
[4] Cuando la entrada destino indica iniciar un cog especifico, la salida destino indica el ID del cog; siempre se inicia o reinicia.
[5] Cuando la entrada destino indica iniciar un cog nuevo (siguiente disponible), la salida destino indica el ID del cog que se inicio o 7 (con C activo) si no hay cog disponible.

Explicación

la instrucción COGINIT se comporta similar a las dos instrucciones spin, COGNEW y COGINIT, juntos. La instrucción COGINIT de ensamblador propeller puede usarse para iniciar un cog o reiniciar un cog activo. El registro *Destination* tiene cuatro campos que determinan que cog se inicia, donde comienza el programa en memoria principal y que es lo que contendrá el registro PAR. La tabla de abajo describe estos campos.

Tabla 3-1: Campos de Registro Destino			
31:18	**17:4**	**3**	**2:0**
14-bit Dirección long registro PAR	14-bit dirección long código a cargar	nuevo	Cog ID

El primer campo, serán escritos para el registro PAR del cog bits 15:2. Esto es 14 bits totales que intentan ser los bits altos de una dirección long 16-bit. Similar al campo *Parameter* de la versión spin de COGINIT, el primer campo de *Destination* se usa para pasar ;a dirección de 14-bit de una localidad de memoria o estructura para el cog iniciado.

El segundo campo, bits 17:4, tiene los 14-bits altos de la dirección long 16-bit apuntando al programa ensamblador para cargar en el cog. Los registros cog $000 a $1EF serán cargados secuencialmente iniciando en esta dirección, el registro de propósito especial se limpiara a cero (0), y el cog comenzara a ejecutar código en el registro $000.

El tercer campo, bit 3, debe estar activo (1) si un nuevo cog se inicia, o limpio (0) si un cog especifico debe iniciarse o reiniciarse.

Si el bit del tercer campo esta activo (1), el hub iniciara el siguiente cog disponible (numero mas bajo inactivo) y regresara ese cog ID en *Destination* (si el efecto WR se especifica).

Si el bit del tercer campo esta limpio (0), el hub iniciara o reiniciara el cog identificado en el cuarto campo *Destination*, bits 2:0.

Si se especifica el efecto WZ, la bandera Z se activara (1) si el ID del cog que regresa es cero (0). Si el efecto WC se especifica, la bandera C se activara (1) si no hay cog disponible. Si el efecto WR se especifica, *Destination* se escribe con el ID del cog que el hub inicio, o iniciara, si usted lo deja escoger uno.

No es practico iniciar código spin desde código ensamblador propeller de usuario; recomendamos iniciar código ensamblador solo con esta instrucción.

COGINIT es una instrucción de hub. Las instrucciones de hub requieren 7 a 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al cog y el momento en el que se ejecuta la instrucción. ver Hub en Pág. 24 para mayor información.

COGSTOP

instrucción: Detiene un cog por su ID.

COGSTOP *CogID*

- *CogID* (campo-d) es el registro conteniendo el ID (0 – 7) del cog a detener.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111 ddddddddd ------011	Stopped ID = 0	No Cog Free	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2,3]	Z	C[4]
$0000_0000; 0	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000011; 3	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[5]	%0_00000011; 3	–	–	wr wz wc	$0000_0000; 0	1	0

[1] La fuente es automáticamente activada al valor 3 por el ensamblador para indicar que es una instrucción de hub COGSTOP.
[2] El destino no se escribe a menos que se de el efecto WR.
[3] La salida destino indica el cog a detener
[4] La bandera C se activa (1) si todos los cogs están corriendo antes de ejecutar la instrucción COGSTOP.
[5] Solo los 3 bits mas bajos de la entrada destino se usan así un valor de 8 es visto como cog 0.

Explicación

La instrucción COGSTOP detiene un cog cuyo ID esta en el registro *CogID*, poniendo ese cog en un estado durmiente. En el estado durmiente, el cog deja de recibir pulsos del reloj del sistema así el consumo de potencia se reduce enormemente.

COGSTOP es una instrucción de hub. Las instrucciones de hub requieren 7 a 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al cog y el momento en el que se ejecuta la instrucción. ver Hub en Pág. 24 para mayor información.

Si se especifica el efecto WZ, la bandera Z se activa (1) si el ID del cog detenido es cero (0). Si se especifica el efecto WC, la bandera C se activa (1) si todos los cogs están corriendo antes de ejecutar esta instrucción. Si el efecto WR se especifica, *Destination* se escribe con el ID del cog que se detiene.

Condiciones (IF_x)

Cada instrucción ensamblador propeller tiene un campo "condición" opcional que se usa para determinar dinámicamente si se ejecuta o no cuando se alcanza en tiempo de ejecución. La sintaxis básica para ensamblador propeller es:

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* ⟨*Effects*⟩

El campo opcional *Condition* puede contener una de 32 condiciones (ver **IF_x** (Condiciones), Pág. 302) y por defecto a **IF_ALWAYS** cuando no se especifica condición. Durante la compilación, **Value** d 4-bit representando la condición se usa en lugar de los bits por defecto del campo –CON– en el opcode.

Esta característica, junto con el apropiado uso de los campos *Effects* opcionales de la instrucción hace al ensamblador propeller muy poderoso. por ejemplo, las banderas C y Z pueden afectarse y posteriormente las instrucciones pueden basar su ejecución en estos resultados.

Cuando una condición de instrucción evalúa a **FALSE**, la instrucción dinámicamente se convierte en un **NOP**, pasando 4 ciclos de reloj pero sin afectar banderas o registros. Esto hace que el tiempo del código multi-decisiones sea muy determinante ya que tiene el mismo camino de ejecución (mismo tiempo de ejecución) y puede usarse y alcanzar todavía muchos posibles salidas.

Ver **IF_x** (Condiciones) en Pág. 302 para mas información.

CTRA, CTRB

Registro: Control de registros Contador A y Contador B.

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   CTRA, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   DestOperand, CTRA ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   CTRB, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   DestOperand, CTRB ⟨Effects⟩
```

Resultado: Opcionalmente, el registro del control de contador se actualiza.

- *Label* es una etiqueta opcional. ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* condición de ejecución opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* Es la instrucción ensamblador. CTRA o CTRB puede usarse en *DestOperand* o *SrcOperand*.
- *SrcOperand* una expresión constante usada por *Instruction* para operar, y opcionalmente escribir, el registro CTRA o CTRB en *DestOperand*.
- *DestOperand* una expresión constante indicando el registro en el que opera y opcionalmente escribe, usando el valor de CTRA o CTRB en *SrcOperand*.

Explicación

CTRA y CTRB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores del cog. Cada cog tiene dos módulos idénticos (A y B) que pueden desarrollar tareas repetitivas. Los registros CTRA y CTRB contienen los parámetros de configuración de los módulos contadores A y B respectivamente.

CTRA y CTRB son registros de lecto/escritura y pueden usarse en los campos *DestOperand* o *SrcOperand* en un instrucción ensamblador. El siguiente código activa el contador A a modo NCO en pin 2 E/S. Ver Registros, Pág. 351, y la sección de lenguaje spin CTRA, CTRB, Pág. 98, para mayor información.

```
                mov     ctra, CtrCfg

CtrCfg          long    %0_00100_000_0000000_000000_000_000010
```

DIRA, DIRB

Registro: Registros de dirección para puertos A y B de 32-bit.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* DIRA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, DIRA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* DIRB, *SrcOperand* ⟨*Effects*⟩ (Reservado para uso futuro)

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, DIRB ⟨*Effects*⟩ (Reservado para uso futuro)

Resultado: Opcionalmente, la dirección del registro se actualiza.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. DIRA o DIRB puede usarse en los campos de instrucciones ensamblador *DestOperand* o *SrcOperand* .
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en, y opcionalmente escribir en, los registros DIRA o DIRB en *DestOperand*.
- *DestOperand* es una expresión constante indicando el registro en el que se opera y opcionalmente se escribe usando el valor de DIRA o DIRB en *SrcOperand*.

Explicación

DIRA y DIRB son dos de seis registros de propósito especial (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pins E/S. Los bits del registro DIRA y DIRB indican el estado de dirección para cada uno de los 32 pins de E/S en el Puerto A y el Puerto B respectivamente. El Propeller P8X32A no incluye el Puerto B. DIRB esta reservado para uso futuro por lo que solo se discutirá DIRA.

DIRA es un registro de lecto/escritura y puede usarse en los campos *DestOperand* o *SrcOperand* de una instrucción ensamblador. Un bit bajo (0) activa el correspondiente pin E/S como dirección de entrada y un pin en alto 91) activa el pin como salida. El siguiente código activa los pins E/S P0 a P3 como salidas.

```
mov     dira, #$0F
```

Ver Registros, Pág. 351, y la sección de lenguaje spin DIRA, DIRB, Pág. 107, para mas información. Tenga en cuenta que en Ensamblador Propeller, a diferencia de Spin, los 32 bits de DIRA se accesan al mismo tiempo a menos que se usen instrucciones MUXx.

DJNZ

instrucción: Decrementa un valor y salta a la dirección si no es cero.

DJNZ *Value*, ⟨#⟩ *Address*

Resultado: *Value*-1 se escribe en *Value*.

- *Value* (campo-d) es el registro a decrementar y probar.
- *Address* (campo-s) es el registro o literal de 9-bit cuyo valor es la dirección a saltar cuando el decrementado *Value* no es cero.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111001 001i 1111 ddddddddd sssssssss	Result = 0	Unsigned Borrow	Written	4 or 8

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0002; 2	$----_----; -	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$----_----; -	-	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$----_----; -	-	-	wz wc	$FFFF_FFFF; -1	0	1

Explicación

DJNZ decrementa el registro *Value* y salta a *Address* si el resultado no es cero.

Cuando se específica el efecto WZ, la bandera Z se activa (1) si el *Value* decrementado es cero. Cuando se especifica el efecto WC, la bandera C se activa (1) si el resultado del decremento es un prestado no signado (sobre flujo 32-bit). El resultado decrementado se escribe en *Value* a menos que se especifique NR.

DJNZ requiere un monto diferente de ciclos de reloj dependiendo si tiene o no que saltar. Si tiene que saltar toma 4 ciclos d reloj, si no salta entonces toma ocho ciclos de reloj. Como el ciclo que utiliza DJNZ necesita ser rápido, se optimiza de esta forma para velocidad.

Efectos (WC, WZ, WR, NR)

Cada instrucción de ensamblador Propeller tiene un campo especial de efecto que puede modificar una bandera o registro cuando se ejecuta. La sintaxis básica es:

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* ⟨*Effects*⟩

El campo opcional *Effects* puede contener uno o mas de los cuatro puntos mostrados abajo. Para cualquier efecto no especificado, el comportamiento por defecto permanece como esta indicado por el bit correspondiente (Z, C, o R) en el campo ZCRI de la instrucción del opcode.

Tabla 3-2: Efectos

Efecto	Descripción
WC	Actualiza bandera C. Ver **WC**, Pág. 387.
WZ	Actualiza bandera Z. Ver **WZ**, Pág. 392.
WR	Actualiza registro destino. Ver **WR**, Pág. 388.
NR	Mantiene registro destino. Ver **NR**, Pág. 335.

Seguido de una instrucción con uno a tres *Effects* coma delimitados para hacer que se afecte la instrucción en el punto indicado, por ejemplo:

```
                and     temp1, #$20       wc
                andn    temp2, #$38       wz, nr
    if_c_and_z  jmp     #MoreCode
```

La primera instrucción hace un AND del valor en temp1 con $20, almacena el resultado en temp1 y modifica con la bandera C para indicar la paridad del resultado. La segunda instrucción hace un AND NOT del valor en el registro temp2 con $38, modifica la bandera Z de acuerdo a si el resultado es cero o no, y no escribe el resultado en temp2. Durante la ejecución de la primera instrucción, la bandera Z no se altera. Durante la ejecución de la segunda instrucción, la bandera C no se altera. Si estas instrucciones no incluyen los efectos WC y WZ, esas banderas no se alterarían. La tercera instrucción la cual especifica una *Condition*, salta a la etiqueta MoreCode (no mostrada) pero solo si ambos C y Z están activos; de otra forma la instrucción JMP funciona como una instrucción NOP.

Usando *Effects* en las instrucciones, junto con *Conditions* en instrucciones posteriores, se habilita el código para hacerlo mas poderoso que lo que es posible con lenguajes ensamblador típicos. Ver IF_x (Condiciones) en Pág. 302 para mas información.

FIT

instrucción: Valida que las instrucciones/datos previos se ajusten a una dirección especifica.

FIT ⟨*Address*⟩

Result: Error en tiempo de compilación si la instrucción/dato previo excede *Address*-1.

- *Address* es una dirección opcional RAM de cog (0-$1F0) para lo cual el código ensamblador no debe alcanzar. Si *Address* no se proporciona, se usa el valor $1F0 (la dirección del primer registro de propósito especial).

Explicación

La instrucción FIT verifica la dirección del apuntador del cog actual y genera un error si esta mas allá de *Address*-1 o si esta ms allá de $1EF (el final de la RAM del cog de propósito general). Esta instrucción puede usarse para asegurar que las instrucciones previas y los datos se ajusten en la RAM del cog, o que estén limitados a la región de la RAM del cog. Nota: cualquier instrucción que no entre en la RAM del cog se dejara fuera cuando el código ensamblador se inicia en el cog. Considere el siguiente ejemplo:

```
DAT
                ORG     492
Toggle          mov     dira, Pin
:Loop           mov     outa, Pin
                mov     outa, #0
                jmp     #:Loop

Pin     long    $1000

                FIT
```

Este código se empujo artificialmente por arriba de la RAM del cog con la instrucción ORG, haciendo que el cog se sobrescriba con el prime registro de propósito especial ($1F0) y haciendo que la instrucción FIT ocasione un error en tiempo de compilación.

FRQA, FRQB

Registro: Registros de Frecuencia del Contador A y Contador B.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* FRQA, *SrcOperand* ⟨*Effects*⟩
DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, FRQA ⟨*Effects*⟩
DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* FRQB, *SrcOperand* ⟨*Effects*⟩
DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, FRQB ⟨*Effects*⟩

Resultado: Opcionalmente, el registro contador de frecuencia se actualiza.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. FRQA o FRQB puede usarse en el campo de la instrucion ensamblador *DestOperand* o *SrcOperand*.
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en, y opcionalmente escribir a, el registro FRQA o FRQB en *DestOperand*.
- *DestOperand* es una expresión constante indicando el registro en el que opera, y opcionalmente en el que escribe a, usando el valor de FRQA o FRQB en *SrcOperand*.

Explicación

FRQA y FRQB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores del cog. Cada cog tiene dos módulos contadores idénticos (A y B) que pueden desarrollar muchas tareas repetitivas. Los registros FRQA y FRQB contienen el valor que esta acumulado en os registros PHSA y PHSB, respectivamente, de acuerdo al correspondiente modo del contador y estimulo de entrada. Ver la sección Spin CTRA, CTRB, Pág. 98, para mayor información.

FRQA y FRQB son registros de lecto/escritura y pueden usarse en los campos *DestOperand* o *SrcOperand* de una instrucción ensamblador. El siguiente código activa FRQA a $F. Ver Registros, Pág. 351, y la sección de lenguaje spin FRQA, FRQB, Pág. 114, para mayor información.

```
mov     frqa, #$F
```

HUBOP

instrucción: Desarrolla una operación de hub.

HUBOP *Destination*, ⟨#⟩ *Operation*

Resultado: Varia dependiendo de la operación desarrollada.

- *Destination* (campo-d) es el registro conteniendo un valor a usar en *Operation*.
- *Operation* (campo-s) es un registro o literal 3-bit que indica la operación a realizar.

Tabla opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 000i 1111 ddddddddd sssssssss	Result = 0	---	Not Written	7..22

Tabla de verdad:

(No se especifica porque varia con cada operación de hub. Ver CLKSET, Pág. 277; COGID, Pág. 289; COGINIT, Pág. 290; COGSTOP, Pág. 292; LOCKNEW, Pág. 313; LOCKRET, Pág. 314; LOCKSET, Pág. 316, y LOCKCLR, Pág. 312.)

Explicación

HUBOP es la guía para cada operación de hub en el chip propeller: CLKSET, COGID, COGINIT, COGSTOP, LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR. La instrucción que desarrolla la operación de hub activa el campo *Operation* (campo-s del opcode) a el valor inmediato de 3-bit que representa la operación deseada (ver la descripción de opcode para cada instrucion de hub para mayor información). La instrucción HUBOP por si misma debería usarse raramente, pero puede ser útil para situaciones especiales.

HUBOP es una instrucción de hub. Las instrucciones de hub necesitan de 7 a 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al hub y la instrucción al momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

IF_x (Condiciones)

Cada instrucción Ensamblador propeller tiene una condición especial que se usa para determinar dinámicamente si se ejecuta o no durante el tiempo de ejecución. La sintaxis básica para la instrucción ensamblador Propeller es:

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* ⟨*Effects*⟩

El campo opcional *Condition* puede contener una d 32 condiciones (ver Tabla 3-3) y por defecto a **IF_ALWAYS** cuando no se especifica condición. El valor **Value** de 4 bits mostrado para cada condición es el valor que se usa para el campo CON en la instrucción del opcode.

Esta característica, junto con el uso apropiado de campos *Effects* opcionales. Hacen al Propeller Ensamblador muy poderoso. Se puede afectar las banderas y posteriormente instrucciones ejecutadas condicionalmente basadas en los resultados. Aquí un ejemplo:

```
            test   _pins, #$20      wc
            and    _pins, #$38
            shl    t1, _pins
            shr    _pins, #3
            movd   vcfg, _pins
    if_nc   mov    dira, t1
    if_nc   mov    dirb, #0
    if_c    mov    dira, #0
    if_c    mov    dirb, t1
```

La primera instrucción `test _pins, #$20 wc`, hace su operación y ajusta el estado de la bandera C porque se especifica el efecto **WC**. Las siguientes 4 instrucciones hacen sus operaciones que podrían afectar la bandera C pero no lo hacen porque no se especifica el efecto **WC**. Esto significa que el estado de la bandera C se mantiene desde la ultima modificación en la primera instrucción. Las cuatro instrucciones finales se ejecutan condicionalmente basadas en el estado de la bandera C que se dio 5 instrucciones antes. Entre las cuatro ultimas instrucciones, las primeras dos instrucciones `mov` tienen condiciones `if_nc`, haciendo que se ejecuten solo "si No C" (si C = 0). Las ultimas dos instrucciones `mov` tienen condiciones `if_c`, lo que hace que se ejecuten solo "si C" (si C = 1). En este caso, los dos pares de instrucciones `mov` se ejecutan en modo mutuamente exclusivo.

Cuando una condición de instrucción evalúa a **FALSE**, la instrucción dinámicamente se convierte en **NOP**, pasando 4 ciclos de reloj pero sin afectar banderas o registros. Esto hace que el tiempo de multi-decisión sea muy determinante.

Tabla 3-3: Condiciones			
condición	**Ejecuta instrucción**	**Valor**	**Sinónimo**
IF_ALWAYS	Siempre	1111	
IF_NEVER	Nunca	0000	
IF_E	Si igual (Z = 1)	1010	IF_Z
IF_NE	Si no igual (Z = 0)	0101	IF_NZ
IF_A	Si arriba (!C & !Z = 1)	0001	IF_NC_AND_NZ –and– IF_NZ_AND_NC
IF_B	Si debajo (C = 1)	1100	IF_C
IF_AE	Si arriba o igual (C = 0)	0011	IF_NC
IF_BE	Si abajo o igual (C | Z = 1)	1110	IF_C_OR_Z –and– IF_Z_OR_C
IF_C	si C activo	1100	IF_B
IF_NC	si C limpio	0011	IF_AE
IF_Z	si Z activo	1010	IF_E
IF_NZ	si Z limpio	0101	IF_NE
IF_C_EQ_Z	Si C igual a Z	1001	IF_Z_EQ_C
IF_C_NE_Z	Si C no es igual a Z	0110	IF_Z_NE_C
IF_C_AND_Z	Si C activo y Z activo	1000	IF_Z_AND_C
IF_C_AND_NZ	Si C activo y Z limpio	0100	IF_NZ_AND_C
IF_NC_AND_Z	si C limpio y Z activo	0010	IF_Z_AND_NC
IF_NC_AND_NZ	Si C limpio y Z limpio	0001	IF_A –and– IF_NZ_AND_NC
IF_C_OR_Z	Si C activo o Z activo	1110	IF_BE –and– IF_Z_OR_C
IF_C_OR_NZ	Si C activo o Z limpio	1101	IF_NZ_OR_C
IF_NC_OR_Z	Si C limpio o Z activo	1011	IF_Z_OR_NC
IF_NC_OR_NZ	Si C limpio o Z limpio	0111	IF_NZ_OR_NC
IF_Z_EQ_C	Si Z igual a C	1001	IF_C_EQ_Z
IF_Z_NE_C	Si Z no es igual a C	0110	IF_C_NE_Z
IF_Z_AND_C	Si Z activo y C activo	1000	IF_C_AND_Z
IF_Z_AND_NC	Si Z activo y C limpio	0010	IF_NC_AND_Z
IF_NZ_AND_C	Si Z limpio y C activo	0100	IF_C_AND_NZ
IF_NZ_AND_NC	Si Z limpio y C limpio	0001	IF_A –and– IF_NC_AND_NZ
IF_Z_OR_C	Si Z activo y C activo	1110	IF_BE –and– IF_C_OR_Z
IF_Z_OR_NC	Si Z activo o C limpio	1011	IF_NC_OR_Z
IF_NZ_OR_C	Si Z limpio o C activo	1101	IF_C_OR_NZ
IF_NZ_OR_NC	Si Z limpio o C limpio	0111	IF_NC_OR_NZ

INA, INB

Registro: Registros de entrada para Puertos A y B de 32-bit

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, INA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, INB ⟨*Effects*⟩ (Reservado para uso futuro)

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador deseada. INA y INB son registros de solo lectura por lo tanto solo se deben usar instrucciones de operando fuente.
- *DestOperand* es una expresión constante indicando el registro en el que se opera y opcionalmente se escribe, usando el valor del operando fuente de la instrucción INA o INB.

Explicación

INA y INB son dos de seis registros de propósito especial (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pins E/S. Los bits del registro INA y INB indican el estado lógico actual de cada uno de los 32 pins E/S en el puerto A y B respectivamente. INB esta reservado par uso futuro; el Propeller P8X32A no incluye el Puerto B así que solo INA se discutirá en esta sección.

INA es un pseudo registro de solo lectura, cuando se usa como operando fuente de una instrucción, le el estado lógico del pin E/S correspondiente. No use INA como operando destino; eso solo resultaría en lectura y modificación del registro sombra cuya dirección esta ocupada por INA.

En INA, un bit bajo (0) indica que el pin correspondiente esta sensando tierra y un alto (1) indica que sensa VDD (3.3 volts). El siguiente código escribe el estado actual del pin E/S P0 a P31 en un registro llamado Temp.

```
mov     Temp, ina
```

Ver Registros, Pág. 351, y el lenguaje spin en la sección INA, INB, Pág. 122, para mayor información. Tenga en cuenta que en el ensamblador Propeller a diferencia de Spin, se accesan los 32 bits de INA al mismo tiempo a menos que se use MUX.

JMP

instrucción: Salta a una dirección.

JMP ⟨#⟩ *Address*

- *Address* (campo-s) es el registro o literal 9-bit cuyo valor es la dirección a saltar.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010111 000i 1111	---------	ssssssss	Result = 0	---	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente	Z	C	Efectos	Destino[2]	Z	C[3]
$----_----; -	$----_----; -	-	-	wr wz wc	31:9 unchanged, 8:0 = PC+1	0	1

[1] Destino es normalmente ignorado para uso típico de **JMP**, sin embargo si el efecto **WR** se da la instrucción JMP se convierte en una instrucción **JMPRET** y el campo-s del destino (9 bits bajos) se sobrescriben con la dirección de regreso (PC+1).
[2] Destino no se escribe a menos que se de el efecto **WR**.
[3] La bandera C se active (1) a menos que PC+1 sea igual a 0; muy diferente ya que requerirá que se ejecute JMP desde la parte alta de la RAM ($1FF; registro de propósito especial **VSCL**).

Explicación

JMP activa el contador del programa (PC) a *Address* haciendo que en la ejecución se salte a esa localidad en la RAM del cog. **JMP** esta relacionada a las instrucciones **CALL**, **JMPRET**, y **RET**; de hecho, tienen el mismo opcode pero con diferentes campos-r y campos-i y varían en el manejo del ensamblador y manejo de usuario campo-d y campo-s.

Saltos Condicionales

Los saltos condicionales y ejecuciones condicionales de cualquier instrucción, se alcanzan precediendo la instrucción con la condición en la forma : **IF_x**. Ver **IF_x** (Condiciones) en Pág. 302 para mayor información.

El Símbolo Here '$'

El símbolo 'Here', **$**, representa la dirección actual. Durante el desarrollo y limpieza el símbolo Here '**$**' se usa frecuentemente en la instrucción **JMP** del campo *Address* (Ej.: JMP #$) para hacer que el cog cicla sin fin. También puede usarse como salto relativo hacia atras o adelante por un numero de instrucciones, por ejemplo:

```
Toggle          mov     dira, Pin     'Activa pin E/S a salida
                xor     outa, Pin     'Cambia estado de pin E/S
                jmp     #$-1          'Cicla sin fin
```

La ultima instrucción `JMP #$-1`, hace que la ejecución salte de la segunda a la ultima instrucción (Ej.: “here” menos 1)

JMPRET

instrucción: Salta a la dirección con la intención de "regresar" a otra dirección.

JMPRET *RetInstAddr*, ⟨#⟩ *DestAddress*

Resultado: PC + 1 se escribe el campo-s del registro indicado por el campo-d.

- *RetInstAddr* (campo-d) es el registro en el cual se almacena la dirección de regreso (PC+1) frecuentemente es la dirección de un apropiado RET o instrucción JMP ejecutada por la rutina *DestAddress*.
- *DestAddress* (campo-s) es el registro o literal 9-bit cuyo valor es la dirección de la rutina temporalmente ejecutada.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010111 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente	Z	C	Efectos	Destino[1]	Z	C[2]
$----_----; -	$----_----; -	-	-	WZ WC	31:9 unchanged, 8:0 = PC+1	0	1

[1] El registro destino (9 bits bajos) se sobrescriben con la dirección de regreso (PC+1) en tiempo de ejecución.

[2] La bandera C se active (1) a menos que PC+1 es igual a 0; muy diferente ya que requiere que JMPRET se ejecute desde la parte alta de la RAM del cog ($1FF; registro de propósito especial VSCL).

Explicación

JMPRET (jump and return) proporciona un mecanismo de "llamar" a otras rutinas y eventualmente regresar a la instrucción que seguía al JMPRET. Para llamadas normales a subrutinas use la instrucción CALL ya que la función es similar a su similar en otros procesadores. La instrucción JMPRET proporciona potencia adicional mas allá del simple "llamado" para ejecutar múltiples subrutinas en forma de intercambio de tareas.

El hardware Propeller no usa una llamada a pila, así la dirección de regreso debe almacenarse d forma diferente. En tiempo de ejecución la instrucción JMPRET almacena la dirección de la siguiente instrucción(PC+1) en la fuente (campo-s) del registro en *RetInstAddr*, luego salta a *DestAddress*.

Si el registro *RetInstAddr* contiene una instrucción RET o JMP y se ejecuta eventualmente por la rutina *DestAddress*i, el comportamiento es similar a la instrucción CALL; la dirección de

regreso se almacena, la rutina *DestAddress* se ejecuta y finalmente el control regresa a la instrucción que seguía a JMPRET. Ver CALL en Pág. 274 para mayor información.

Cuando se usa diferente, la instrucción JMPRET puede ayudar en procesos sencillos de multi tareas. Esto se hace definiendo un grupo de registros para colocar diversas direcciones de regreso, especificando los registros para los campos *RetInstAddr* y *DestAddress*.Por ejemplo:

```
Initialize      mov     Task2, #SecondTask      'Inicia 1er destino.

FirstTask       <inicia primer tarea >
                ...
                jmpret  Task1, Task2            'Proporciona el
Segundo ciclo de tareas
                <mas código de tareas  >
                ...
                jmpret  Task1, Task2            'Proporciona el
Segundo ciclo de tareas
                jmp       #FirstTask            'Cicla la primer
tarea

SecondTask      <inicia la segunda tarea>
                ...
                jmpret  Task2, Task1            'Proporciona el
primer ciclo de tareas
                <mas código de segunda tarea >
                ...
                jmpret  Task2, Task1            'Proporciona el
primer ciclo de tareas
                jmp     #SecondTask             'Cicla la segunda
tarea

Task1           res     1                       'Declara dirección de
tareas
Task2           res     1                       'Almacena espacio
```

En este ejemplo hay dos rutinas, FirstTask y SecondTask, las cuales sirven como tareas separadas en el proceso del Cog. La función que cada tarea realice es relativamente irrelevante; estas pueden hacer cosas iguales o no. Task1 y Task2 son longs, declarados al final de código, usados para guardar el destino y dirección de regreso para facilitar el cambio de ejecución entre las dos tareas.

La primer instrucción `mov Task2,#SecondTask`, almacena la dirección de `SecondTask` en el registro `Task2`. Esto lleva al registro de tareas a realizar el primer cambio.

Una vez que `FirstTask` inicio, desarrolla algunas operaciones indicadas por "…" y alcanza la primer instrucción **JMPRET**, `jmpret Task1,Task2`. Primero, **JMPRET** guarda ;a dirección de regreso (PC + 1, la dirección de `<mas código primer tarea>`) en el campo-s del registro `Task1`, luego salta a la dirección indicada por `Task2`. Como inicializamos `Task2` apuntando a `SecondTask`, la segunda tarea se ejecutara ahora.

`SecondTask` desarrolla algunas operaciones dadas en "…" y alcanza otra instrucción **JMPRET**, `jmpret Task2,Task1`. Observe que esta es similar a la instrucción `FirstTask`'s **JMPRET** excepto que el orden de `Task1` y `Task2` es inverso. Esta instrucción **JMPRET** guarda la dirección de regreso (PC + 1, la dirección de `<mas código de la segunda tarea >`) en el campo-s del registro `Task2`, luego salta a la dirección indicada por `Task1`. Como `Task1` contiene la dirección de `<mas código de la primer tarea>`, como se escribió en la instrucción anterior **JMPRET**, la ejecución ahora cambia de regreso a `FirstTask` iniciando con la línea `<mas código de la primer tarea>`.

La ejecución continua cambiando entre `FirstTask` y `SecondTask` siempre que exista la instrucción **JMPRET**, confiadamente regresando a donde previamente se quedo la tarea la ultima vez. Cada instrucción **JMPRET** sobrescribe la dirección destino usada previamente con la nueva dirección de regreso y luego salta al nuevo destino; la dirección de regreso de la ultima vez.

Este concepto de multitareas se aplica en diversas formas. Por ejemplo al remover una de las instrucciones **JMPRET** de `SecondTask` ocasionara que `FirstTask` reciba menos ciclos del cog por unidad de tiempo. Técnicas como esta pueden usarse para obtener mas ciclos de cog a tareas sensitivas al tiempo, o a porciones de tareas sensitivas al tiempo. También es posible introducir mas registros `Task` para hacer multitareas entre tres o mas rutinas en el mismo cog. El tiempo de proceso de cada tarea esta siempre determinado y basado dependiendo de donde se coloca la instrucción **JMPRET** y a que tareas hace referencia.

Observe que el estado de las banderas C y Z no se cambian y no se almacenan entre estos cambios de tareas lógicas. Por esta razón, es importante cambiar entre tareas solo cuando las banderas no se necesitan o no están en peligro de cambiar el estado antes de que la ejecución regrese.

JMPRET es un supergrupo de la instrucción **CALL**; de hecho, es el mismo opcode de **CALL** pero con el campo-i y campo-d configurado por el desarrollador en vez del ensamblador. Ver **CALL** en Pág. 274 para mayor información.

La dirección de regreso (PC+1) se escribe a la fuente (campo-s) del registro *RetInstAddr* a menos que se especifique el efecto NR. Por supuesto al especificar NR no se recomienda para la instrucción JMPRET ya que la convierte en una instrucción JMP, o RET.

LOCKCLR

instrucción: Limpia Lock a falso y obtiene el estado previo.

LOCKCLR *ID*

Resultado: Opcionalmente, el estado previo de Lock se escribe a la bandera C

- *ID* (campo-d) es el registro que contiene el ID (0 – 7) del seguro a limpiar.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111	ddddddddd	------111	ID = 0	Prior Lock State	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2,3]	Z	C
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0005; 5	%0_00000111; 7	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0000; 0	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0008; 8[5]	%0_00000111; 7	–	–	wr wz wc	$0000_0000; 0	1	0

[1] La fuente se active automáticamente al valor inmediato 7 por el ensamblador indicando que esta es una instrucción LOCKCLR hub.
[2] El destino no se escribe a menos que se especifique el efecto WR.
[3] La salida destino indica el ID del seguro que se limpio.
[4] La bandera C indica el estado previo del bit del seguro; en estos casos el bit de seguro se activo por una ejecución previa LOCKSET (no mostrada). El siguiente ejemplo limpia la bandera C porque el bit del seguro se limpio en el ejemplo anterior.
[5] Solo los 3 bits mas bajos del destino se utilizan, así un valor de 8 se puede observar para el bit de seguro ID 0.

Explicación

LOCKCLR es una de cuatro instrucciones de seguro (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usadas para administrar recursos definidos por usuario considerados mutuamente exclusivos. CKCLR limpia el seguro descrito por el registro *ID* a cero (0) y regresa el estado previo de ese seguro en la bandera C, si el efecto WC se especifico. La instrucción LOCKCLR se comporta similar a la instrucción LOCKCLR; Ver LOCKCLR en Pág. 124.

Si el efecto WZ se especifica, la bandera Z se active (1) si el ID del seguro que se limpio es cero (0). Si se especifica el efecto WC, la bandera C se active igual al estado previo del seguro. Si se especifica el efecto WR, el ID del seguro limpiado se escribe en *ID*.

LOCKCLR es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

LOCKNEW

instrucción: Genera un Nuevo seguro y obtiene su ID

LOCKNEW *NewID*

Resultado: El ID del Nuevo seguro (0-7) se escribe en *NewID*.

- *NewID* (campo-d) es el registro donde el ID del nuevo seguro se escribe.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0011 1111 dddddddd ------100	ID = 0	No Lock Free	Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino	Z	C
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0001; 1	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0002; 2	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0003; 3	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0004; 4	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0005; 5	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0006; 6	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	0
$----_----; -	%0_00000100; 4	-	-	wz wc	$0000_0007; 7	0	1

[1] La fuente se activa inmediatamente al valor de 4 para indicar que es una instrucción hub LOCKNEW.

Explicación

LOCKNEW es una de cuatro instrucciones de seguro (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usadas para administrar recursos definidos por usuario mutuamente exclusivos. LOCKNEW genera un seguro único, del hub, y obtiene el ID de ese seguro. La instrucción LOCKNEW se comporta similar a la instruction Spin LOCKNEW; Ver LOCKNEW en Pág. 126.

Si se especifica el efecto WZ, la bandera Z se active (1) si el ID de regreso es cero (0). Si se especifica el efecto WC, la bandera C se active (1) si no hay seguros disponibles. El ID del seguro nuevo se escribe en *NewID* a menos que se especifique el efecto NR.

LOCKNEW es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

LOCKRET

instrucción: Libera un seguro para futuros requerimientos de LOCKNEW

LOCKRET *ID*

- *ID* (campo-d) es el registro que contiene el ID (0 – 7) del seguro a regresar a la bandeja.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111 ddddddddd ------101	ID = 0	No Lock Free	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2]	Z	C[3]
$0000_0000; 0	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0005; 5	%0_00000101; 5	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0008; 8[4]	%0_00000101; 5	–	–	wr wz wc	$0000_0000; 0	1	0

[1] La fuente se active automáticamente al valor de 5 para indicar que es una instrucción LOCKRET de hub.
[2] El destino no s escribe a menos que se especifique el efecto WR
[3] La bandera C se active (1) si todos los bits de seguros están acomodados antes de ejecutar la instrucción LOCKRET.
[4] Solo los 3 bits mas bajos del destino se usan, así el valor de 8 se le da al bit del seguro ID 0.

Explicación

LOCKRET es una de cuatro instrucciones de seguros (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usadas para administrar recursos definidos por usuario mutuamente exclusivos. LOCKRET regresa un seguro, por *ID*, de regreso a la pila de seguro del Hub y pueden se reutilizaos por el cog despues. La instrucción LOCKRET se comporta similar a la instrucción spin LOCKRET; Ver LOCKRET en Pág. 129.

Si el efecto WZ se especifica, la bandera Z se active (1) si el ID del seguro de regreso es cero (0). Si se especifica el efecto WC, la bandera C se active (1) si todos los bits de seguros se acomodaron antes de ejecutar la instrucción. Si se especifica el efecto WR, el ID del seguro de regreso se escribe en *ID*.

LOCKRET es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

LOCKSET

instrucción: Activa un seguro como verdadero y obtiene su estado previo.

LOCKSET *ID*

Resultado: Opcionalmente, el estado previo del seguro se escribe a la bandera C.

- *ID* (campo-d) es el registro que contiene el ID (0 – 7) del seguro a activar.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000011 0001 1111 ddddddddd ------110	ID = 0	Prior Lock State	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente[1]	Z	C	Efectos	Destino[2,3]	Z	C
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	0
$0000_0005; 5	%0_00000110; 6	–	–	wr wz wc	$0000_0005; 5	0	1[4]
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0000; 0	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]
$0000_0008; 8[5]	%0_00000110; 6	–	–	wr wz wc	$0000_0000; 0	1	1[4]

[1] La fuente se active a l valor de 6 automáticamente para indicar que es una instrucción hub LOCKSET.
[2] El destino no se escribe a menos que se de l efecto WR.
[3] La salida destino indica el ID del bit del seguro que se activo.
[4] La bandera C indica el estado previo del bit del seguro; en estos casos el bit del seguro se activo en el ejemplo anterior.
[5] Solo los 3 bits mas bajos del destino se usan; así el valor de 8 se observa para el bit del seguro ID 0.

Explicación

LOCKSET es uno de cuatro instrucciones de seguro (LOCKNEW, LOCKRET, LOCKSET, y LOCKCLR) usados para administrar recursos definidos por usuario mutuamente exclusivos. LOCKSET active el seguro descrito por el registro *ID* a uno (1) y regresa el estado previo de ese seguro en la bandera C; si se especifico el efecto WC. La instrucción LOCKSET se comporta similar a la instrucción Spin LOCKSET; Ver LOCKSET en Pág. 130.

Si el efecto WZ se especifico, la bandera Z se active (1) si el ID del seguro activado es cero (0). Si se especifica el efecto WC, la bandera C se active igual que el estado previo del seguro. Si se especifica el efecto WR, el ID del seguro activado se escribe en *ID*.

LOCKSET es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

MAX

instrucción: Limite máximo de un valor no signado a otro valor no signado.

MAX *Value1*, ⟨#⟩ *Value2*

Resultado: El menor de un valor no signado *Value1* y *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a comparar contra *Value2* y es el destino en el cual se escribe el menor valor de los dos.
- *Value2* (campo-s) es un registro o literal de 9-bit cuyo valor es comparado con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010011 001i 1111 ddddddddd sssssssss	S = 0	Unsigned (D < S)	Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente [1]	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

[1] Ambas, fuente y destino se tratan como valores no signados.

Explicación

MAX compara los valores no signados de *Value1* y *Value2* y almacena el menor de los dos en el registro *Value1*, efectivamente limitando *Value1* a un máximo de *Value2*.

Si se especifica el efecto WZ, la bandera Z se active (1) si *Value2* es cero (0). Si el efecto WC se especifica, la bandera C se active (1) si el no signado *Value1* es menor que el no signado *Value2*. El menor de los dos valores se escribe en *Value1* a menos que se especifique el efecto NR.

MAXS

instrucción: Limite máximo de un valor signado a otro valor signado.

MAXS *SValue1*, ⟨#⟩ *SValue2*

Resultado: El menor de los valores signados en *SValue1* y *SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor a comparar contra *SValue2* y es el destino en el cual se escribe el menor de los dos.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor se compara contra *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010001 001i 1111 ddddddddd sssssssss	S = 0	Signed (D < S)	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$0000_0001; 1	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0002; 2	-	-	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$0000_0000; 0	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	0	1
$0000_0001; 1	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0

Explicación

MAXS compara el valor signado de *SValue1*y *SValue2* y almacena el menor de los dos en el registro *SValue1*, efectivamente limitando *SValue1* a un máximo de *SValue2*.

Si se especifico el efecto WZ, la bandera Z se activa (1) si *SValue2* es cero (0). Si se especifica el efecto WC, la bandera C se active (1) si el valor signado *SValue1* es menor que el valor signado *SValue2*. El menor de los dos valores se escribe en *SValue1* a menos que el efecto NR se especifique.

MIN

instrucción: Limite mínimo de un valor no signado a otro valor no signado.

MIN *Value1*, ⟨#⟩ *Value2*

Resultado: El mayor de los valores no signados de *Value1* y *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a comparar contra *Value2* y es el destino en el cual se escribe el mayor de los dos.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor se compara con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010010 001i 1111	ddddddddd	sssssssss	S = 0	Unsigned (D < S)	Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente [1]	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0001; 2	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0001; 1	1	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1

[1] Fuente y destino se tratan como valores no signados.

Explicación

MIN compara los valores no signados de *Value1* y *Value2* y almacena el mayor de los dos en el registro *Value1*, efectivamente limitando *Value1* a un mínimo de *Value2*.

Si el efecto WZ se especifica, la bandera Z se activa (1) si *Value2* es cero (0). Si el efecto WC se especifica, la bandera C se activa (1) si el no signado *Value1* es menor que el no signado *Value2*. El mayor de los dos valores de escribe a *Value1* a menos que NR este especificado.

MINS

instrucción: Limite mínimo de un valor signado a otro valor signado.

MINS *SValue1*, ⟨#⟩ *SValue2*

Resultado: El mayor de los valores signados *SValue1* y *SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor a comparar contra *SValue2* y es el destino en el cual se escribe el mayor de los dos.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor se compara contra *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010000 001i 1111 ddddddddd sssssssss	S = 0	Signed (D < S)	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$0000_0002; 2	0	1
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0000; 0	–	–	wz wc	$0000_0001; 1	1	0
$0000_0001; 1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1
$FFFF_FFFF; -1	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1

Explicación

MINS compara los valores no signados de S*Value1* y S*Value2* y almacena el mayor de los dos en el registro S*Value1*, efectivamente limitando S*Value1* a un mínimo de *Value2*.

Si el efecto WZ se especifica, la bandera Z se activa (1) si S*Value2* es cero (0). Si el efecto WC se especifica, la bandera C se activa (1) si el no signado S*Value1* es menor que el no signado S*Value2*. El mayor de los dos valores de escribe a S*Value1* a menos que NR este especificado.

MOV

instrucción: Activa un registro a un valor.

MOV *Destination*, ⟨#⟩ *Value*

Resultado: *Value* se almacena en *Destination*.

- *Destination* (campo-d) es el registro en el cual se almacena *Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor se almacena en *Destination*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101000 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$0000_0001; 1	0	0

Explicación

MOV copia o almacena el numero de *Value* en *Destination*.

Si se especifico el efecto WZ, la bandera Z se activa (1) si *Value* es igual a cero. Si el efecto WC se especifica, la bandera C se active al MSB de *Value*. El resultado se escribe en *Destination* a menos que se especifique el efecto NR.

MOVD

instrucción: Activa un campo de registro destino a un valor.

MOVD *Destination*, ⟨#⟩ *Value*

Resultado: *Value* se almacena en el campo-d de *Destination* (bits 17..9).

- *Destination* (campo-d) es el registro cuyo campo destino (bits 17..9) se activa al valor de *Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor se almacena en el campo-d de *Destination*..

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010101 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	–	–	wz wc	$0003_FE00; 261,632	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	–	–	wz wc	$FFFC_01FF; -261,633	0	0

Explicación

MOVD copia el valor 9-bit de *Value* en el campo-d de *Destination* (campo destino) bits 17..9. Los otros bits de *Destination* no cambian. Esta instrucción es practica para activar ciertos registros como CTRA y VCFG, y para actualizar el campo destino de instrucciones en código de modificación propia.

En código de modificación propia, asegúrese que al menos se ejecuta una instrucción entre una instrucción MOVI y la instrucción objetivo que MOVI modifica. Esto le da al cog tiempo de escribir el resultado de MOVI antes de que alcance la instrucción y la ejecute; de otra forma la aun no modificada versión de la instrucción objetivo se alcanzara y se ejecutara.

Si el efecto WZ se especifico, la bandera Z se activa (1) si *Value* es igual a cero. EL resultado se escribe a *Destination* a menos que el efecto NR se especifique.

MOVI

instrucción: Activa una instrucción de registro y campos de efectos a un valor.

MOVI *Destination*, ⟨#⟩ *Value*

Resultado: *Value* se almacena en el campo-i de *Destination* y campo-efectos (bits 31..23).

- *Destination* (campo-d) es el registro cuya instrucción y campos efecto (bits 31..23) se activan al valor de *Value*.
- *Value* (campo-s) es el registro o literal 9-bit cuyo valor se almacena en el campo efecto e i instrucción de *Destination*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010110 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	–	–	wz wc	$FF80_0000; -8,388,608	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	–	–	wz wc	$007F_FFFF; 8,388,607	0	0

Explicación

MOVI copia el valor 9-bit de *Value* en el campo efecto e instrucción de *Destination* bits 31..23. Los otros bits de *Destination* se mantienen sin cambio. Esta instrucción es útil para activar ciertos registros como CTRA y VCFG, y para actualizar la instrucción y campos de efecto de instrucciones en código de modificación propia.

En código de modificación propia, asegúrese que al menos se ejecuta una instrucción entre una instrucción MOVI y la instrucción objetivo que MOVI modifica. Esto le da al cog tiempo de escribir el resultado de MOVI antes de que alcance la instrucción y la ejecute; de otra forma la aun no modificada versión de la instrucción objetivo se alcanzara y se ejecutara.

Si el efecto WZ se especifico, la bandera Z se activa (1) si *Value* es igual a cero. EL resultado se escribe a *Destination* a menos que el efecto NR se especifique.

MOVS

instrucción: Activa una campo fuente de registro a un valor.

MOVS *Destination*, ⟨#⟩ *Value*

Resultado: *Value* se almacena en el campo-s de *Destination* (bits 8..0).

- *Destination* (campo-d) es el registro cuya campo fuente (bits 8..0) se activa al valor de *Value*.
- *Value* (campo-s) es un registro o literal de 9-bit cuyo valor se almacena en el campo fuente de *Destination*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010100 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_01FF; 511	–	–	wz wc	$0000_01FF; 511	0	1
$FFFF_FFFF; -1	$0000_01FF; 511	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$0000_0000; 0	–	–	wz wc	$FFFF_FE00; -512	0	0

Explicación

MOVS copia el valor de 9-bit de *Value* en el campo fuente de *Destination* (campo-s) bits 8..0. Los otros bits de *Destination* quedan sin cambios. Esta instrucción es útil para activar registros como CTRA y VCFG, y para actualizar el campo fuente de instrucciones en código de modificación propia.

En código de modificación propia, asegúrese que al menos se ejecuta una instrucción entre una instrucción MOVI y la instrucción objetivo que MOVI modifica. Esto le da al cog tiempo de escribir el resultado de MOVI antes de que alcance la instrucción y la ejecute; de otra forma la aun no modificada versión de la instrucción objetivo se alcanzara y se ejecutara.

Si el efecto WZ se especifico, la bandera Z se active (1) si *Value* es igual a cero. EL resultado se escribe a *Destination* a menos que el efecto NR se especifique.

MUXC

instrucción: Activa bits discretos de un valor al estado de C.

MUXC *Destination*, ⟨#⟩ *Mask*

Resultado: Los bits *Destination*, indicados por *Mask*, se activan al estado de C.

- *Destination* (campo-d) es el registro cuyos bits descries por *Mask* se afectan por C.
- *Mask* (campo-s) es un registro o literal 9-bit cuyo valor contiene bits altos (1) para cada bit en *Destination* para activar el estado de la bandera C.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011100 001i 1111	ddddddddd	sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	1	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0003; 3	–	0	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	–	1	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	–	0	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	–	1	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	–	0	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	–	1	wz wc	$FFFF_FFFF; -1	0	0

Explicación

MUXC activa cada bit del valor en *Destination*, el cual corresponde a los bits altos (1) de *Mask*, para el estado C. Todos los bits de *Destination* que no están marcados por un alto (1) de *Mask* no se afectan. Esta instrucción es útil para activar o limpiar bits discretos, o grupos de bits, en un valor existente.

Si el efecto WZ se especifica, la bandera Z se activa (1) si el valor final de *Destination* es 0. Si se especifico el efecto WC la bandera C se activa (1) si el resultado *Destination* contiene un numero impar de bits altos (1). El resultado se escribe en *Destination* a menos que se especifique el efecto NR.

MUXNC

instrucción: Activa bits discretos de un valor al estado de !C.

MUXNC *Destination*, ⟨#⟩ *Mask*

Resultado: Los bits *Destination*, indicados por *Mask*, se activan al estado de !C.

- *Destination* (campo-d) es el registro cuyos bits descries por *Mask* se afectan por !C.
- *Mask* (campo-s) es un registro o literal 9-bit cuyo valor contiene bits altos (1) para cada bit en *Destination* para activar el estado inverso de la bandera C.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011101 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	-	1	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	-	0	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	-	1	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	0	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	-	1	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	-	1	wz wc	$0000_0000; 0	1	0

Explicación

MUXNC activa cada bit del valor en *Destination*, el cual corresponde a los bits altos (1) de *Mask*, para el estado !C. Todos los bits de *Destination* que no están marcados por un alto (1) de *Mask* no se afectan. Esta instrucción es útil para activar o limpiar bits discretos, o grupos de bits, en un valor existente.

Si el efecto WZ se especifica, la bandera Z se activa (1) si el valor final de *Destination* es 0. Si se especifico el efecto WC la bandera C se activa (1) si el resultado *Destination* contiene un numero impar de altos (1). El resultado se da en *Destination* a menos que se de el efecto NR.

MUXNZ

instrucción: Activa bits discretos de un valor al estado de !Z.

MUXNZ *Destination*, ⟨#⟩ *Mask*

Resultado: Los bits *Destination*, indicados por *Mask*, se activan al estado de !Z.

- *Destination* (campo-d) es el registro cuyos bits descries por *Mask* se afectan por !Z.
- *Mask* . (campo-s) es un registro o literal 9-bit cuyo valor contiene bits altos (1) para cada bit en *Destination* para activar el estado inverso de la bandera Z

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011111 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	0	–	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	0	–	wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0003; 3	1	–	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	0	–	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	1	–	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	0	–	wz wc	$FFFF_FFFF; -1	0	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0

Explicación

MUXNZ activa cada bit del valor en *Destination*, el cual corresponde a los bits altos (1) de *Mask*, para el estado !Z. Todos los bits de *Destination* que no están marcados por un alto (1) de *Mask* no se afectan. Esta instrucción es útil para activar o limpiar bits discretos, o grupos de bits, en un valor existente.

Si el efecto WZ se especifica, la bandera Z se activa (1) si el valor final de *Destination* es 0. Si se especifico el efecto WC la bandera C se activa (1) si el resultado *Destination* contiene un numero impar de altos (1). El resultado se da en *Destination* a menos que se de el efecto NR.

MUXZ

instrucción: Activa bits discretos de un valor al estado de Z.

MUXZ *Destination*, ⟨#⟩ *Mask*

Resultado: Los bits *Destination*, indicados por *Mask*, se activan al estado de Z.

- *Destination* (campo-d) es el registro cuyos bits descries por *Mask* se afectan por Z.
- *Mask*. (campo-s) es un registro o literal 9-bit cuyo valor contiene bits altos (1) para cada bit en *Destination* para activar el estado de la bandera Z

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011110 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	0	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	1	-	wz wc	$0000_0001; 1	0	1
$0000_0000; 0	$0000_0003; 3	0	-	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0003; 3	1	-	wz wc	$0000_0003; 3	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	0	-	wz wc	$A841_2000; -1,472,126,976	0	0
$AA55_2200; -1,437,261,312	$1234_5678; 305,419,896	1	-	wz wc	$BA75_7678; -1,166,707,080	0	1
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	0	-	wz wc	$0000_0000; 0	1	0
$AA55_2200; -1,437,261,312	$FFFF_FFFF; -1	1	-	wz wc	$FFFF_FFFF; -1	0	0

Explicación

MUXZ activa cada bit del valor en *Destination*, el cual corresponde a los bits altos (1) de *Mask*, para el estado Z. Todos los bits de *Destination* que no están marcados por un alto (1) de *Mask* no se afectan. Esta instrucción es útil para activar o limpiar bits discretos, o grupos de bits, en un valor existente.

Si el efecto WZ se especifica, la bandera Z se activa (1) si el valor final de *Destination* es 0. Si se especifico el efecto WC la bandera C se activa (1) si el resultado *Destination* contiene un numero impar de altos (1). El resultado se da en *Destination* a menos que se de el efecto NR .

NEG

instrucción: Obtiene el negativo de un numero.

NEG *NValue*, ⟨#⟩ *SValue*

Resultado: –*SValue* se almacena en *NValue*.

- *NValue* (campo-d-) es el registro en el cual se escribe el negativo de *SValue*.
- *SValue* (campo-s) es un registro o literal 9-bit cuyo valor negativo se escribirá en *NValue*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101001 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	-	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	-	wz wc	$8000_0000; -2,147,483,648 [1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] El numero negativo mas pequeño (-2,147,483,648) no tiene valor positivo correspondiente en matemática de complemento a dos 32-bit

Explicación

NEG almacena el negativo de *SValue* en *NValue*.

Si se especifico el efecto WZ, la bandera Z se active (1) si *SValue* es cero. Si se especifico el efecto WC, la bandera C se active (1) si *SValue* es negativo o esta limpio (0) si *SValue* es positivo. El resultado se escribe en *NValue* a menos que el efecto NR se especifique.

NEGC

instrucción: Obtiene un valor, o su opuesto, basado en C.

NEGC *RValue*, ⟨#⟩ *Value*

Resultado: *Value* o –*Value* se almacena en *RValue*.

- *RValue* (campo-d) es el registro en el cual se escribe *Value* o –*Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor (si C = 0) o valor opuesto (si C = 1) se escribirá en *RValue*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101100 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	1	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	x	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	0	wz wc	$8000_0001; -2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] El numero negativo mas pequeño (-2,147,483,648) no tiene valor positivo correspondiente en matemática de complemento a dos 32-bit

Explicación

NEGC almacena *Value* (si C = 0) o –*Value* (si C = 1) en *RValue*.

Si el efecto WZ se especifico, la bandera Z se activa (1) so *Value* es cero. Si se especifico el efecto WC, la bandera C se activa (1) si *Value* es negativo o limpio (0) si *Value* es positivo. El resultado se escribe en *RValue* a menos que el efecto NR se especifique.

NEGNC

instrucción: Obtiene un valor, o su opuesto, basado en !C.

NEGNC *RValue*, ⟨#⟩ *Value*

Resultado: –*Value* o *Value* se almacena en *RValue*.

- *RValue* (campo-d) es el registro en el cual se escribe –*Value* o *Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor opuesto (si C = 0) o valor (si C = 1) se escribirá en *RValue*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101101 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	-	0	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	-	x	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	-	0	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	0	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	-	x	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	-	1	wz wc	$8000_0001; -2,147,483,647	0	1

[1] El numero negativo mas pequeño (-2,147,483,648) no tiene valor positivo correspondiente en matemática de complemento a dos 32-bit

Explicación

NEGNC almacena –*Value* (si C = 0) o *Value* (si C = 1) en *RValue*.

Si el efecto WZ se especifico, la bandera Z se activa (1) so *Value* es cero. Si se especifico el efecto WC, la bandera C se activa (1) si *Value* es negativo o limpio (0) si *Value* es positivo. El resultado se escribe en *RValue* a menos que el efecto NR se especifique.

NEGNZ

instrucción: Obtiene un valor, o su opuesto, basado en !Z.

NEGNZ *RValue*, ⟨#⟩ *Value*

Resultado: –*Value* o *Value* se almacena en *RValue*.

- *RValue* (campo-d) es el registro en el cual se escribe –*Value* o *Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor opuesto (si Z = 0) o valor (si Z = 1) se escribirá en *RValue*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101111 001i 1111 ddddddddd sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	0	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	1	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	x	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	0	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	1	-	wz wc	$8000_0001; -2,147,483,647	0	1

[1] El numero negativo mas pequeño (-2,147,483,648) no tiene valor positivo correspondiente en matemática de complemento a dos 32-bit

Explicación

NEGNZ almacena –*Value* (si Z= 0) o *Value* (si Z= 1) en *RValue*.

Si el efecto WZ se especifico, la bandera Z se activa (1) so *Value* es cero. Si se especifico el efecto WC, la bandera C se activa (1) si *Value* es negativo o limpio (0) si *Value* es positivo. El resultado se escribe en *RValue* a menos que el efecto NR se especifique.

NEGZ

instrucción: Obtiene un valor, o su opuesto, basado en Z.

NEGZ *RValue*, ⟨#⟩ *Value*

Resultado: *Value* o –*Value* se almacena en *RValue.*

- *RValue* (campo-d) es el registro en el cual se escribe *Value* o –*Value*.
- *Value* (campo-s) es un registro o literal 9-bit cuyo valor (si Z = 0) o valor opuesto (si Z = 1) se escribirá en *RValue*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
101110 001i 1111	ddddddddd	sssssssss	Result = 0	S[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$----_----; -	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFF; -1	0	1
$----_----; -	$FFFF_FFFF; -1	1	-	wz wc	$0000_0001; 1	0	1
$----_----; -	$0000_0000; 0	x	-	wz wc	$0000_0000; 0	1	0
$----_----; -	$0000_0001; 1	0	-	wz wc	$0000_0001; 1	0	0
$----_----; -	$0000_0001; 1	1	-	wz wc	$FFFF_FFFF; -1	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	0	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$----_----; -	$7FFF_FFFF; 2,147,483,647	1	-	wz wc	$8000_0001; -2,147,483,647	0	0
$----_----; -	$8000_0000; -2,147,483,648	x	-	wz wc	$8000_0000; -2,147,483,648[1]	0	1
$----_----; -	$8000_0001; -2,147,483,647	0	-	wz wc	$8000_0001; -2,147,483,647	0	1
$----_----; -	$8000_0001; -2,147,483,647	1	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1

[1] El numero negativo mas pequeño (-2,147,483,648) no tiene valor positivo correspondiente en matemática de complemento a dos 32-bit

Explicación

NEGZ almacena *Value* (si Z= 0) o –*Value* (si Z= 1) en *RValue*.

Si el efecto WZ se especifico, la bandera Z se activa (1) so *Value* es cero. Si se especifico el efecto WC, la bandera C se activa (1) si *Value* es negativo o limpio (0) si *Value* es positivo. El resultado se escribe en *RValue* a menos que el efecto NR se especifique.

NOP

Instrucción: No operación, solo deja pasar cuatro ciclos de reloj.

```
NOP
```

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
------ ---- 0000 --------- ---------	---	---	---	4

Tabla de verdad:
(No especificada ya que NOP no desarrolla ninguna acción)

Explicación

NOP no desarrolla operaciones pero consume 4 ciclos de reloj. NOP tiene su campo –CON– activo a ceros, la condición NEVER; efectivamente cada instrucción con una condición NEVER es una instrucción NOP. El porque de esto, la instrucción NOP nunca puede ser precedida por una *Condition*, tal como IF_Z o IF_C_AND_Z, ya que nunca puede ejecutarse condicionalmente.

NR

Efecto: Previene que una instrucción ensamblador escriba un resultado.

⟨*Label*⟩ ⟨*Condition*⟩ *Instruction Operands* NR

Resultado: El registro destino de *Instruction* no se afecta.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Instruction* es la instrucción ensamblador deseada.
- *Operands* es cero, uno o dos operandos según sea requerido por *Instruction*.

Explicación

NR (No Result) es uno de cuatro efectos opcionales (WC, WZ, WR, y NR) que influye en el comportamiento de las instrucciones ensamblador. NR hace que se ejecute la instrucción ensamblador sin afectar el registro destino.

Por ejemplo, por defecto la instrucción SHL (Shift Left) mueve el valor destino a la izquierda un numero de bits, escribe el resultado de regreso en el registro destino, y opcionalmente indica el estado a través de las banderas C y Z. Si todo lo que realmente se necesita es el estado de la bandera C de la instrucción SHL, solo especifique ambos efectos WC y NR:

```
shl     value, #1     WC, NR     'Coloca valor MSB en C
```

El ejemplo anterior efectivamente active la bandera C al estado del bit alto de value (bit 31) sin afectar el contenido final de value.

Ver Efectos en Pág. 298 para mayor información.

Operadores

El código Ensamblador Propeller puede contener expresiones usando cualquier operador que este permitido en expresiones constantes. La Tabla 3-4 sumariza todos los operadores permitidos en el Ensamblador Propeller. Revise la referencia del lenguaje Spin en la sección Operadores en Pág. 147 para descripciones detalladas de sus funciones.

Tabla 3-4: Expresiones Constantes Operadores Lógico/Matemáticos

Operador Normal	Es Unario	Descripción , Numero de pagina
+		Suma, 153
+	✓	Positivo (+X); forma unaria de suma, 154
-		Resta, 154
-	✓	Negación (-X); forma unaria de resta, 154
*		Multiplica y regresa bits bajos 32 bits (signados), 157
**		Multiplica y regresa bits altos 32 bits (signados), 157
/		Divide (signado), 158
//		Modulus (signado), 158
#>		Limite mínimo (signado), 159
<#		Limite máximo (signado), 159
^^	✓	Raíz cuadrada, 160
||	✓	Valor absoluto, 160
~>		Corre a la derecha aritmética, 162
|<	✓	Bitwise: Decodifica valor (0-31) en bit alto simple long 164
>|	✓	Bitwise: Codifica long en un valor (0 - 32) bit alta prioridad, 164
<<		Bitwise: Mueve a la izquierda, 165
>>		Bitwise: Mueve a la derecha, 165
<-		Bitwise:Rota a la izquierda, 166
->		Bitwise: Rota a la derecha, 167
><		Bitwise: Reverse, 167
&		Bitwise: AND, 168
|		Bitwise: OR, 169
^		Bitwise: XOR, 169
!	✓	Bitwise: NOT, 170
AND		Boolean: AND (mueve un no 0 a -1), 171
OR		Boolean: OR (mueve un no 0 a -1), 172
NOT	✓	Boolean: NOT (mueve un no 0 a -1), 172
==		Boolean: Igual a, 173
<>		Boolean: No igual a, 174
<		Boolean: Menor que (signado), 174
>		Boolean: Mayor que (signado), 175
=<		Boolean: Igual o menor que (signado), 175
=>		Boolean: Igual o mayor que (signado), 176
@	✓	Símbolo dirección, 177

OR

instrucción: Bitwise hace OR dos valores.

OR *Value1*, ⟨#⟩ *Value2*

Resultado: El resultado de *Value1* OR *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a hacer OR con *Value2* y es el destino en el cual se escribe el resultado.
- *Value2* (campo-s) es el registro o literal 9-bit cuyo valor es el bitwise para hacer OR con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011010 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	1
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_000F; 15	0	0

Explicación

OR (bitwise inclusive OR) desarrolla un OR del valor en *Value2* con *Value1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* OR *Value2* es igual a cero. Si se especifica el efecto WC, la bandera C se activa (1) si el resultado contiene un numero impar de bits altos (1). El resultado se escribe a *Value1* a menos que se especifique el efecto NR.

ORG

instrucción: Ajusta el apuntador ensamblador en tiempo de compilación.

ORG ⟨*Address*⟩

- *Address* es una dirección opcional RAM del Cog (0-495) para ensamblar el siguiente código ensamblador. Si *Address* no se da se usa el valor 0.

Explicación

La instrucción **ORG** (origin) activa el apuntador ensamblador de la herramienta Propeller a un nuevo valor para uso en referencias de direcciones. **ORG** típicamente aparece como `ORG 0`, o solo `ORG`, al inicio de Nuevo código ensamblador que se pretende iniciar en un cog separado.

ORG solo afecta referencias símbolos, no afecta la posicion del código ensamblador en el cog. Cuando se inicia código ensamblador a través de **COGNEW** o **COGINIT**, el destino del cog siempre carga el código en su RAM iniciando en la dirección 0.

Aun pensando que el código siempre se carga d esta forma, el compilador/ensamblador no sabe que parte del código constituye el comienzo ya que los desarrolladores son libres de iniciar cualquier código en cualquier dirección

Para resolver esto, el ensamblador usa un punto de referencia (el valor del apuntador) para calcular la dirección absoluta de los símbolos referenciados. Esas direcciones absolutas están codificadas en el destino de la instrucción ensamblador (campo-d) o la fuente (campo-s) en vez de la referencia simbólica. Por ejemplo:

```
DAT
                org     0               'Inicia Cog en RAM 0
Toggle          mov     dira, Pin       'Activa dirección E/S a salida
:Loop           xor     outa, Pin       'Cambia estado de salida de pin
                jmp     #:Loop          'Ciclo son fin

Pin             long    $0000_0010      'Usa pin 4 E/S ($10 or %1_0000)
```

La sentencia **ORG** en este ejemplo active el apuntador ensamblador a cero (0) así el código que sigue se ensambla con ese punto de referencia en mente. El porque de esto, para el ensamblador el símbolo `Toggle` esta lógicamente en la localidad 0 de la RAM del cog, el simbolo`:Loop` esta en la localidad 1 y el símbolo `Pin` esta en la localidad e de la RAM. El

ensamblador reemplazara cada referencia de estos símbolos con sus respectivas direcciones de código.

Cuando el código `Toggle` se inicia con `COGNEW(@Toggle, 0)`, por ejemplo, el código ejecutara apropiadamente iniciando en la dirección 0 de la RAM ya que todas las direcciones de símbolos se calcularon desde ese punto. Si la instrucción **ORG** fuera `ORG 1` y el código `Toggle` se inicio, no lo ejecutara apropiadamente porque la dirección del símbolo fue calculada desde la referencia incorrecta (1 en vez de 0).

Un Objeto Propeller puede contener múltiples casos de la instrucción **ORG**, cada una puesta inmediatamente antes de una porción ejecutable de código ensamblador. Sin embargo, no es común usar **ORG** con valores diferentes a cero (0) pensando que puede ser útil cuando se crean porciones intercambiables de código ensamblador en tiempo de ejecución.

OUTA, OUTB

Registros: Salida de registros para puertos A y B de 32-bit.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* OUTA, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, OUTA ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* OUTB, *SrcOperand* ⟨*Effects*⟩ (Reservado para uso futuro)

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, OUTB ⟨*Effects*⟩ (Reservado para uso futuro)

Resultado: Opcionalmente, la salida de registro se actualiza.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. OUTA o OUTB puede usarse en cualquier campo *DestOperand* o *SrcOperand* de la instrucción ensamblador.
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en y opcionalmente escribir a, el registro OUTA o OUTB en *DestOperand*.
- *DestOperand* es una expresión constante que indica el registro en el que se opera, y opcionalmente se escribe, usando el valor de OUTA o OUTB en *SrcOperand*.

Explicación

OUTA y OUTB son dos de seis registros de propósito especial (DIRA, DIRB, INA, INB, OUTA y OUTB) que afectan directamente los pins E/S. El bit del registro OUTA y OUTB indica el estado de la salida de cada uno de los 32 pins E/S en Puerto A y Puerto B respectivamente. OUTB esta reservado para uso futuro por lo que solo se discutirá el Puerto OUTA.

OUTA es un registro de lecto/escritura que se usa en los campos *DestOperand* o *SrcOperand*. Si el pin E/S se active como salida, un bit bajo (0) en OUTA hace que la salida este aterrizada y un pin alto (1) hace la salida VDD (3.3 volts). El siguiente código activa los pins P0 a P3 como salida alta.

```
mov     dira, #$0F
mov     outa, #$0F
```

Ver Registros, Pág. 351, y la sección del lenguaje Spin OUTA, OUTB, Pág. 179, para mayor información. Tenga en cuenta que en el Ensamblador Propeller a diferencia del spin todos los 32 bits de OUTA se accesan al mismo tiempo a menos que se usen instrucciones MUXx.

PAR

Registro: Registro de parámetros del Cog de inicio.

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction DestOperand*, PAR ⟨*Effects*⟩

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. PAR es un registro de solo lectura y por lo tanto debe usarse en el operando fuente solamente.
- *DestOperand* es una expresión constante que contiene el registro en el que se opera y opcionalmente se escribe a, por el valor de PAR en la instrucción del operando fuente.

Explicación

El registro PAR contiene el valor de la dirección pasado al campo *Parameter* de la instrucción basada en spin COGINIT o COGNEW, o en los bits altos del campo *Destination* de la instrucción basada en ensamblador COGINIT. Cuando el cog inicia, su código ensamblador propeller puede usar el contenido del registro PAR para acomodar y operar en memoria principal entre el mismo y el código que esta llamando.

Es importante observar que el valor pasado a PAR se pretende que sea dirección long, así solo 14-bits (2 al 15) se retienen; los dos bits bajos se limpian a cero para asegurar que es una dirección long alineada. Otros valores diferentes de direcciones long pueden pasarse, pero deben ser 14 bits o menos movidos a la izquierda y derecha apropiadamente por ambos el llamador y el cog iniciado recientemente.

PAR es un pseudo registro de solo lectura, cuando se usa como fuente, lee el valor pasado al cog durante el inicio. No use PAR como destino ya que solo ocasionara la lectura y modificación del registro sombra cuya dirección esta ocupada por PAR.

El siguiente código mueve el valor de PAR en el registro Addr para uso posterior en el código. Ver Registros, Pág. 351, y la sección de lenguaje spin PAR, Pág. 182, para mayor información.

```
DAT
                org 0                   'Reinicia el apuntador
ensamblador
AsmCode         mov     Addr,  par      'Obtiene la dirección
compartida
```

```
                        '<more code here>           'Desarrolla operaciones

Addr        res         1
```

PHSA, PHSB

Registro: Registro de Fase de Contador A y B.

```
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   PHSA, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   DestOperand, PHSA ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   PHSB, SrcOperand ⟨Effects⟩
DAT
   ⟨Label⟩ ⟨Condition⟩ Instruction   DestOperand, PHSB ⟨Effects⟩
```

Resultado: Opcionalmente, l registro contador de fase se actualiza.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. PHSA y PHSB son pseudo registros de lecto/escritura que actúan diferente en *SrcOperand* y en *DestOperand*.
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en, y opcionalmente escribir a, el registro PHSA o PHSB en *DestOperand*.
- *DestOperand* es una expresión constante indicando el registro en el que se opera y opcionalmente se escribe a, el valor de PHSA o PHSB en *SrcOperand*.

Explicación

PHSA y PHSB son dos de seis registros (CTRA, CTRB, FRQA, FRQB, PHSA, y PHSB) que afectan el comportamiento de los módulos contadores del cog. Cada cog tiene dos módulos contadores idénticos (A y B) que desarrollan tareas repetitivas. Los registros PHSA y PHSB contienen las acumulaciones de los valores de registro de FRQA y FRQB, respectivamente, de acuerdo al correspondiente modo del contador y el estimulo de entrada. Ver la sección del lenguaje spin CTRA, CTRB en Pág. 98 para mayor información.

PHSA y PHSB son pseudo registros de lecto/escritura. En *SrcOperand* leen el valor acumulador del contador . En el *DestOperand* leen el registro sombra del registro que ocupa PHSA o PHSB, pero mas modificaciones afectan ambos a la sombra y al acumulador del registro.

EL siguiente código mueve el valor de PHSA en Result. Ver Registros, Pág. 351, y la sección de lenguaje Spin PHSA, PHSB, Pág. 184, para mayor información.

```
                mov     Result, phsa    'Obtiene el valor actual de
phase
```

RCL

instrucción: Rota C a la izquierda en un valor especificado por bits.

RCL *Value*, ⟨#⟩ *Bits*

Resultado: *Value* tiene *Bits* copias de C rotado a la izquierda.

- *Value* (campo-d) es el registro en el cual se rota C a la izquierda.
- *Bits* (campo-s) es un registro o literal de 5-bit cuyo valor es el numero de bits de *Value* para rotar C a la izquierda

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001101 001i 1111 ddddddddd sssssssss	Result = 0	D[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$8000_0000; -2,147,483,648	$0000_0000; 0	-	x	wz wc	$8000_0000; -2,147,483,648	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	1	wz wc	$0000_0001; 1	0	1
$2108_4048; 554,188,872	$0000_0002; 2	-	0	wz wc	$8421_0120; -2,078,211,808	0	0
$2108_4048; 554,188,872	$0000_0002; 2	-	1	wz wc	$8421_0123; -2,078,211,805	0	0
$8765_4321; -2,023,406,815	$0000_0004; 4	-	0	wz wc	$7654_3210; 1,985,229,328	0	1
$8765_4321; -2,023,406,815	$0000_0004; 4	-	1	wz wc	$7654_321F; 1,985,229,343	0	1

Explicación

RCL (Rotate Carry Left) desarrolla una rotación a la izquierda de *Value*, *Bits* veces, usando el valor de la bandera C original para cada uno de los LSB afectados.

Si se especifica el efecto WZ, la bandera Z se activa (1) si el resultado de *Value* es cero. Si el efecto WC se especifica, al final de la operación , la bandera C se activa igual al bit 31 del original *Value*. El resultado se escribe en *Value* a menos que se especifique el efecto NR.

RCR

instrucción: Rota C a la derecha en un valor especificado por bits.

RCR Value, ⟨#⟩ Bits

Resultado: *Value* tiene *Bits* copias de C rotado ala derecha

- *Value* (campo-d) es el registro en el cual se rota C a la derecha.
- *Bits* (campo-s) es un registro o literal de 5-bit cuyo valor es el numero de bits de *Value* para rotar C a la derecha

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001100 001i 1111 ddddddddd sssssssss	Result = 0	D[0]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0000; 0	-	x	wz wc	$0000_0001; 1	0	1
$0000_0001; 1	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	1
$0000_0001; 1	$0000_0001; 1	-	1	wz wc	$8000_0000; -2,147,483,648	0	1
$18C2_1084; 415,371,396	$0000_0002; 2	-	0	wz wc	$0630_8421; 103,842,849	0	0
$18C2_1084; 415,371,396	$0000_0002; 2	-	1	wz wc	$C630_8421; -969,898,975	0	0
$8765_4321; -2,023,406,815	$0000_0004; 4	-	0	wz wc	$0876_5432; 141,972,530	0	1
$8765_4321; -2,023,406,815	$0000_0004; 4	-	1	wz wc	$F876_5432; -126,462,926	0	1

Explicación

RCR (Rotate Carry Right) desarrolla una rotación a la derecha de *Value*, *Bits* veces, usando el valor original de la bandera C para cada MSB afectado.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el resultado de *Value* es cero. Si se especifico el efecto WC, al final de la operación, la bandera C se hace igual al bit 0 del original *Value*. El resultado se escribe en *Value* a menos que se especifique el efecto NR.

RDBYTE

instrucción: Lee un byte de memoria principal.

RDBYTE *Value*, ⟨#⟩ *Address*

Result: Byte cero extendido se almacena en *Value*.

- *Value* (campo-d) es el registro para almacenar el valor del byte cero extendido.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal a leer.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000000 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	31:8 = 0, 7:0 = byte value	0	0

[1] La salida destino es el byte cero extendido que se lee de la memoria principal; si se genera incluyendo un efecto NR cambiara RDBYTE en una instrucción WRBYTE.

[2] La bandera Z se limpia (0) a menos que la salida destino sea 0.

Explicación

RDBYTE sincroniza al Hub, lee el byte de memoria principal en *Address*, lo hace cero extendido, y lo almacena en el registro *Value*.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el valor leído de memoria principal en cero. El efecto NR no se puede usar con RDBYTE ya que lo cambiaria a una instrucción WRBYTE.

RDBYTE es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

RDLONG

Instruction: Lee un long de memoria principal.

RDLONG *Value*, ⟨#⟩ *Address*

Resultado: Long se almacena en *Value*.

- *Value* (campo-d) es el registro para almacenar el valor del byte cero extendido.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal a leer.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000010 001i 1111	ddddddddd	sssssssss	Result = 0	---	Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	long value	0	0

[1] La salida destino es el byte cero extendido que se lee de la memoria principal; si se genera incluyendo un efecto NR cambiara RDLONG en una instrucción WRLONG.
[2] La bandera Z se limpia (0) a menos que la salida destino sea 0.

Explicación

RDLONG sincroniza al Hub, lee el long de memoria principal en *Address*, y lo almacena en el registro *Value*. *Address* puede apuntar a cualquier byte en el longs deseado, los dos bits mas bajos se limpiaran a cero resultando en una dirección apuntando a la frontera de un long.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el valor que lee de memora principal es cero. El efecto NR no se puede utilizar con RDLONG ya que esto lo cambiaria a una instrucción WRLONG.

RDLONG es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

RDWORD

instrucción: Lee un word de memoria principal.

RDWORD *Value*, ⟨#⟩ *Address*

Resultado: Un word cero extendido se almacena en *Value*.

- *Value* (campo-d) es el registro para almacenar el valor del byte cero extendido.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal a leer.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000001 001i 1111 ddddddddd sssssssss	Result = 0	---	Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	31:16 = 0, 15:0 = word value	0	0

[1] La salida destino es el byte cero extendido que se lee de la memoria principal; si se genera incluyendo un efecto NR cambiara RDWORD en una instrucción WRWORD.

[2] La bandera Z se limpia (0) a menos que la salida destino sea 0.

Explicación

RDWORD sincroniza al Hub, lee un word de memoria principal en *Address*, la hace cero extendida, y la almacena en el registro *Value*. *Address* puede apuntar a cualquier byte dentro del word deseado; los bits mas bajos de la dirección se limpiaran a cero resultando en una dirección que apunta a la frontera de word.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el valor que lee de memora principal es cero. El efecto NR no se puede utilizar con RDWORD ya que esto lo cambiaria a una instrucción WRWORD.

RDWORD es una instrucción de hub. Las instrucciones de hub requieren de 7 a 22 ciclos de reloj para ejecutarse dependiendo de la relación entre la ventana de acceso al hub y el momento de la ejecución. Ver Hub en Pág. 24 para mayor información.

Registros

Cada cog contiene 16 registros de propósito especial para accesar los pins E/S, los contadores el generador de video y los parámetros pasados al momento de iniciar el cog. Todos estos registros se explican en la Referencia de Lenguaje Spin y la mayoría de la información aplica a ambos, Spin y Ensamblador Propeller. La siguiente tabla muestra los 16 registros de propósito especial, indica donde encontrar información, y muestra que detalles, si los hay, no aplican al Ensamblador Propeller. Cada uno de estos registros puede accesarse como cualquier otro registro en los campos fuente y destino de las instrucciones, excepto los que se marcan con la nota de pie 1 o 2. Estos registros especiales solo se pueden leer con el campo fuente de una instrucción; (1) no son de escritura, o (2) no pueden usarse en el campo destino para una operación lecto-modificar-escritura.

Tabla 3-5: Registros

Registro(s)	Descripción
DIRA, DIRB [3]	Registros de dirección para puertos A y B de 32-bit, ver Pág. 295 y 107. El parámetro opcional "[*Pin(s)*]" no aplica ensamblador propeller; todos los bits del registro entero son lecto/escritura una vez a menos que use instrucciones MUXx .
INA [1], INB [1,3]	Registros entrada para puertos A y B de 32-bit (solo lectura) ver Pág. 304 y 122. El parámetro opcional "[*Pin(s)*]" no aplica para ensamblador propeller; todos los bits del registro se leen a la vez.
OUTA, OUTB [3]	Registros de salida para puerto A y B de 32-bit , ver Pág. 340 y 179. El parámetro opcional "[*Pin(s)*]" no aplica a ensamblador propeller; todos los bits del registro entero se leen a la vez a menos que se usen instrucciones MUXx.
CNT [1]	Registro contador del sistema 32-bit (solo lectura) ver Pág. 288 y 76.
CTRA, CTRB	Control de registros contadores A y B ver Pág. 294 y 98.
FRQA, FRQB	Registros de frecuencia de contador A y B ver Pág. 300 y 114.
PHSA [2], PHSB [2]	Registro de fase cerrada de contador A y B, ver Pág. 344 y 184.
VCFG	Registro de configuración de video ver Pág. 381 y 218.
VSCL	Registro de escala de video, ver Pág. 382 y 221.
PAR [1]	Registro de parámetros de inicio de cog (solo lectura), ver Pág. 342 y 182.

Nota 1: Para ensamblador Propeller solo es accesible como fuente (Ej.:, mov *dest*, *source*). Ver las secciones de ensamblador PAR, Pág. 342; CNT, Pág. 288, e INA, INB, Pág. 304.

Nota 2: Para ensamblador Propeller solo lectura como fuente (Ej.: mov *dest, source*); lecto modificación escritura no es posible como registro destino. Ver sección de Lenguaje ensamblador PHSA, PHSB en Pág. 344.

Nota 3: Reservada para uso futuro.

RES

instrucción : Reserva el siguiente long para símbolo

⟨*Symbol*⟩ RES ⟨*Count*⟩

- *Symbol* es un nombre opcional para el long reservado en la RAM del cog
- *Count* es el numero opcional de logs a reservar para *Symbol*. Si no se especifica, RES reserva un long.

Explicación

La instrucción RES (reserve) efectivamente reserve uno o mas longs de RAM del cog incrementando el tiempo de compilación por el apuntador ensamblador por *Count*. Normalmente se usa para reservar memoria para un símbolo que no necesita inicialización a un valor especifico, por ejemplo:

```
DAT
                ORG     0
AsmCode         mov     Time, cnt         'Obtiene contador de
sistema
                add     Time, Delay       'Suma delay
:Loop           waitcnt Time, Delay       'Espera por la ventana
                nop                       'Hace algo útil

                jmp     #:Loop            'Cicla sin fin

Delay    long   6_000_000                 'Tiempo de tamaño de
ventana
Time     RES    1                         'Tiempo de espacio de
trabajo de ventana
```

La ultima línea del ejemplo AsmCode, de arriba, reserve un long de memoria RAM del cog para el símbolo Time sin definir un valor inicial.. AsmCode usa Time como una variable long para esperar al inicio de una ventana de tiempo de 6 millones de ciclos de reloj. Cuando AsmCode se inicia en un cog, se carga en la RAM del cog como se muestra a continuación

Símbolo	Dirección	Instrucción / Datos
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay
	3	nop
	4	jmp #:Loop
Delay	5	6_000_000
Time	6	?

RES simplemente incrementa el tiempo de compilación del apuntador ensamblador que afecta referencias de símbolo posteriores (RAM del cog); no consume espacio en el objeto (RAM principal). Esta distinción es importante en como impacta el código del objeto, inicializando símbolos, y afectando iteraciones en tiempo de ejecución.

- Como incrementa el tiempo de compilación del apuntador solamente, la dirección computada de todos los símbolos que siguen a la instrucción RES se afectan.
- No se consume espacio en la aplicación/objeto (RAM principal). Una ventaja si se definen buffers que solo deben existir en la RAM del Cog pero no en RAM principal.

Precaución: Use RES Solo despues de Instrucciones y Datos

Es importante usar RES solo despues de instrucciones finales y datos en un programa lógico ensamblador. Al colocar RES antes podría arrojar resultados inesperados como se explica.

Recuerde que las instrucciones ensamblador y los datos se colocan en la imagen de la memoria de la aplicación en el orden preciso en el que se ingresan, independientemente del apuntador. Esto es porque el código ensamblador debe cargarse en orden, iniciando con la etiqueta de rutina designada. Sin embargo como RES no genera ningún dato (o código) no tiene ningún efecto en la imagen de memoria; RES solo ajusta el valor del apuntador.

Por naturaleza de la instrucción RES, cualquier dato o código que aparece despues de la instrucción RES se colocara inmediatamente despues de la ultima unidad no-RES, en el mismo espacio lógico que las unidades RES. Considere el siguiente ejemplo donde el código, de arriba, tiene el orden de Time y Delay invertidas.

```
DAT
                ORG     0
AsmCode         mov     Time, cnt         'Obtiene el contador del
sistema
                add     Time, Delay       'Suma Delay
```

```
:Loop           waitcnt Time, Delay       'espera por ventana de
tiempo
                nop                       'Hace algo útil
                jmp     #:Loop            'Ciclo sin fin

Time     RES    1                         'Tiempo de espacio de
trabajo de ventana
Delay    long   6_000_000                 'Tiempo de tamaño de
ventana
```

Este ejemplo iniciara en el cog como sigue:

Símbolo	Dirección	Instrucción / Datos
AsmCode	0	mov Time, cnt
	1	add Time, Delay
:Loop	2	waitcnt Time, Delay
	3	nop
	4	jmp #:Loop
Time	5	6_000_000
Delay	6	?

Observe como se invirtió Time y Delay respecto al ejemplo anterior pero no los datos? Esto es lo que sucedió:

- Primero, el ensamblador coloca todo en la imagen de la memoria del objeto exactamente como lo hizo antes hasta e incluyendo la instrucción **JMP**.
- El ensamblador alcanza el símbolo Time, el cual es declarado con una instrucción **RES**, así que iguala Time a la dirección 5 (el valor actual del apuntador ensamblador) y luego incrementa el apuntador en 1. No se coloca datos en la memoria de la aplicación debido a este paso.
- El ensamblador alcanza el símbolo Delay, el cual es declarado como dato un dato **LONG**, así que iguala Delay a la dirección 6 (el valor actual del apuntador), incrementando el apuntador ensamblador en 1, y colocando después el dato, 6_000_000, en la siguiente localidad disponible en la imagen de memoria despues de la instrucción **JMP** la cual ocurre donde esta apuntando Time lógicamente.

El efecto cuando se inicia un cog es que el símbolo Time ocupa el mismo espacio de RAM del cog que el valor inicial de Delay, y Delay existe en el siguiente registro que contiene un dato desconocido. El código no correera como se pretende.

Por esta razón es mejor colocar las instrucciones RES despues de la ultima instrucción y antes del ultimo dato definido en el que se soporta el código ensamblador, como se muestra en el primer ejemplo.

RET

instrucción: Regresa a la dirección anterior grabada.

RET

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
010111 0001 1111	---------	---------	Result = 0	---	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente	Z	C	Efectos	Destino [2]	Z	C [3]
$----_----; -	$----_----; -	-	-	wr wz wc	31:9 unchanged, 8:0 = PC+1	0	1

[1] El destino es normalmente ignorado por el uso **RET**, sin embargo si el efecto **WR** se da, la instrucción **RET** se convierte en una instrucción **CALL** y el campo destino se sobrescriben con la dirección de regreso (PC+1).
[2] El destino no se escribe a menos que se de el efecto **WR**.
[3] La bandera C se activa (1) a menos que PC+1 sea 0; poco deseado ya que requerirá a **RET** ejecutarse desde el inicio de la RAM del Cog ($1FF; propósito de registro especial **VSCL**).

Explicación

RET regresa la ejecución a la dirección previamente guardada por el contador del programa (PC) a esa dirección. La instrucción **RET** es para ser usada junto con una etiqueta en la forma "label_ret" y una instrucción **CALL** cuyo objetivo es la rutina de **RET**, indicada por "label." Ver **CALL** en Pág. 274 para mayor información.

RET es un subgrupo de la instrucción **JMP** pero con el campo-I y el campo-s no especificado. Esta también relacionada cercanamente a las instrucciones **CALL** y **JMPRET**; de hecho, estos usan el mismo Opcode pero con diferentes campo-r, campo-s y variando en el manejo del ensamblador y manejo de usuario en los valores de campo-d y campo-s.

REV

instrucción: Invierte el valor de LSB y lo hace cero extendido.

REV *Value*, ⟨#⟩ *Bits*

Resultado: *Value* tiene los 32-bit bajos de su LSB invertidos y los bits altos limpios.

- *Value* (campo-d) es el registro que contiene el valor cuyos bits son invertidos.
- *Bits* (campo-s) es el registro o literal de 5-bit cuyo valor se resta de 32 (32-bits), es el numero de *Value* LSB a invertir. Los bits altos de *Bits* MSB de *Value* se limpian.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001111 001i 1111 ddddddddd sssssssss	Result = 0	D[0]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$8421_DECA; -2,078,155,062	$0000_001F; 31	–	–	wz wc	$0000_0000; 0	1	0
$8421_DECA; -2,078,155,062	$0000_001C; 28	–	–	wz wc	$0000_0005; 5	0	0
$8421_DECA; -2,078,155,062	$0000_0018; 24	–	–	wz wc	$0000_0053; 83	0	0
$8421_DECA; -2,078,155,062	$0000_0010; 16	–	–	wz wc	$0000_537B; 21,371	0	0
$8421_DECA; -2,078,155,062	$0000_0000; 0	–	–	wz wc	$537B_8421; 1,400,603,681	0	0
$4321_8765; 1,126,270,821	$0000_001C; 28	–	–	wz wc	$0000_000A; 10	0	1
$4321_8765; 1,126,270,821	$0000_0018; 24	–	–	wz wc	$0000_00A6; 166	0	1
$4321_8765; 1,126,270,821	$0000_0010; 16	–	–	wz wc	$0000_A6E1; 42,721	0	1
$4321_8765; 1,126,270,821	$0000_0000; 0	–	–	wz wc	$A6E1_84C2; -1,495,169,854	0	1

Explicación

REV (Reverse) invierte los bits bajos (32 - *Bits*) de *Value* LSB y limpia los bits altos de *Value* MSB.

Si el efecto WZ se especifico, la bandera Z se activa (1) si el resultado de *Value* es cero. Si se especifico el efecto WC, la bandera C se activa (1) igual al bit 0 del original *Value*. El resultado se escribe en *Value* a menos que el efecto NR se especifique.

ROL

instrucción: Rota el valor a la izquierda por el numero especificado de bits

ROL *Value*, ⟨#⟩ *Bits*

Resultado: *Value* se rota a la izquierda por *Bits*.

- *Value* (campo-d) es el registro a rotar a la izquierda
- *Bits* (campo-s) es un registro o literal de 5-bit cuyo valor es el numero de bits a rotar a la izquierda.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001001 001i 1111 ddddddddd sssssssss	Result = 0	D[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	1	0
$8765_4321; -2,023,406,815	$0000_0004; 4	-	-	wz wc	$7654_3218; 1,985,229,336	0	1
$7654_3218; 1,985,229,336	$0000_000C; 12	-	-	wz wc	$4321_8765; 1,126,270,821	0	0
$4321_8765; 1,126,270,821	$0000_0010; 16	-	-	wz wc	$8765_4321; -2,023,406,815	0	0

Explicación

ROL (Rotate Left) rota *Value* a la izquierda, *Bits* veces. Los MSB que se rotan de *Value* se rotan en sus LSB.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el resultado de *Value* es cero. Si se especifico el efecto WC, al final de la operación, la bandera C se activa (1) igual al bit 31 del original *Value*. El resultado se escribe en *Value* a menos que NR se especifique.

ROR

instrucción: Rota el valor a la derecha por un numero especifico de bits.

ROR *Value*, ⟨#⟩ *Bits*

Resultado: *Value* se rota a la derecha por *Bits*.

- *Value* (campo-d) es el registro a rotar a la derecha.
- *Bits* (campo-s) es un registro o literal de 5-bit cuyo valor es el numero de bits a rotar a la derecha

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001000 001i 1111	ddddddddd	sssssssss	Result = 0	D[0]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0001; 1	-	-	wz wc	$0000_0000; 0	1	0
$1234_5678; 305,419,896	$0000_0004; 4	-	-	wz wc	$8123_4567; -2,128,394,905	0	0
$8123_4567; -2,128,394,905	$0000_000C; 12	-	-	wz wc	$5678_1234; 1,450,709,556	0	1
$5678_1234; 1,450,709,556	$0000_0010; 16	-	-	wz wc	$1234_5678; 305,419,896	0	0

Explicación

ROR (Rotate Right) rota *Value* a la derecha *Bits* veces. Los LSB rotados de *Value* se rotan en sus MSB.

Si se especifico el efecto WZ, la bandera Z se activa (1) si el resultado de *Value* es cero. Si se especifico el efecto WC, al final de la operación, la bandera C se activa (1) igual al bit 0 del original *Value*. El resultado se escribe en *Value* a menos que NR se especifique.

SAR

Instruction: Shift value arithmetically right by specified number of bits.

SAR *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted arithmetically right by *Bits*.

- *Value* (d-field) is the register to shift arithmetically right.
- *Bits* (s-field) is a register or a 5-bit literal whose value is the number of bits to shift arithmetically right.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001110 001i 1111 ddddddddd sssssssss	Result = 0	D[0]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$FFFF_FF9C; -100	$0000_0001; 1	–	–	wz wc	$FFFF_FFCE; -50	0	0
$FFFF_FF9C; -100	$0000_0002; 2	–	–	wz wc	$FFFF_FFE7; -25	0	0
$FFFF_FF9C; -100	$0000_0003; 3	–	–	wz wc	$FFFF_FFF3; -13	0	0
$FFFF_FFF3; -13	$0000_0001; 1	–	–	wz wc	$FFFF_FFF9; -7	0	1
$FFFF_FFF9; -7	$0000_0001; 1	–	–	wz wc	$FFFF_FFFC; -4	0	1
$FFFF_FFFC; -4	$0000_0001; 1	–	–	wz wc	$FFFF_FFFE; -2	0	0
$0000_0006; 6	$0000_0001; 1	–	–	wz wc	$0000_0003; 3	0	0
$0000_0006; 6	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0006; 6	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0

Explicación

SAR (Shift Arithmetic Right) shifts *Value* right by *Bits* places, extending the MSB along the way. This has the effect of preserving the sign in a signed value, thus SAR is a quick divide-by-power-of-two for signed integer values.

Si se especifica el efecto WZ, la bandera Z se activa (1) si el resultado *Value* es cero. Si el efecto WC se especifica, la bandera C se activa igual que el bit 0 original de *Value*. El resultado se escribe a *Value* a menos que el efecto NR se especifique.

SHL

instrucción: Mueve el valor a la izquierda por un especifico numero de bits.

SHL *Value*, ⟨#⟩ *Bits*

Resultado: *Value* se mueve a la izquierda por *Bits*.

- *Value* (campo-d) es el registro a mover a la izquierda.
- *Bits* (campo-s) es un registro o literal 5-bit cuyo valor es el numero de bits a mover a la izquierda.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001011 001i 1111	ddddddddd	sssssssss	Result = 0	D[31]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$8765_4321; -2,023,406,815	$0000_0004; 4	–	–	wz wc	$7654_3210; 1,985,229,328	0	1
$7654_3210; 1,985,229,328	$0000_000C; 12	–	–	wz wc	$4321_0000; 1,126,236,160	0	0
$4321_0000; 1,126,236,160	$0000_0010; 16	–	–	wz wc	$0000_0000; 0	1	0

Explicación

SHL (Shift Left) mueve *Value* a la izquierda por *Bits* lugares y active el nuevo LSB a 0.

Si se especifica el efecto WZ, la bandera Z se activa (1)si el resultado de *Value* es cero. Si el efecto WC se especifica, la bandera C se activa igual al bit 31 del original *Value*. El resultado se escribe en *Value* a menos que NR se especifique.

SHR

instrucción: Mueve el valor a la derecha por un especifico numero de bits.

SHR *Value*, ⟨#⟩ *Bits*

Resultado: *Value* se mueve a la derecha por *Bits*.

- *Value* (campo-d) es el registro a mover a la derecha.
- *Bits* (campo-s) es un registro o literal 5-bit cuyo valor es el numero de bits a mover a la derecha.
-

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
001010 001i 1111 ddddddddd sssssssss	Result = 0	D[0]	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$1234_5678; 305,419,896	$0000_0004; 4	-	-	wz wc	$0123_4567; 19,088,743	0	0
$0123_4567; 19,088,743	$0000_000C; 12	-	-	wz wc	$0000_1234; 4,660	0	1
$0000_1234; 4,660	$0000_0010; 16	-	-	wz wc	$0000_0000; 0	1	0

Explicación

SHR (Shift Right) mueve *Value* a la derecha por *Bits* lugares y active el nuevo MSB a 0.

Si se especifica el efecto WZ, la bandera Z se activa (1)si el resultado de *Value* es cero. Si el efecto WC se especifica, la bandera C se activa igual al bit 0 del original *Value*. El resultado se escribe en *Value* a menos que NR se especifique.

SUB

instrucción: Resta dos valores no signados

SUB *Value1*, ⟨#⟩ *Value2*

Resultado: La diferencia del no signado *Value1* y *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a restar de *Value2*, y es el destino en el cual se escribe el resultado.
- *Value2* (campo-s) es el registro o literal 9-bit cuyo valor es restado de *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100001 001i 1111 ddddddddd sssssssss	D - S = 0	Unsigned Borrow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0002; 2	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wz wc	$0000_0000; 0	1	0
$0000_0002; 2	$0000_0003; 3	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	1

[1] Fuente y destino se tratan como valores no signados.

Explicación

SUB resta el no signado *Value2* del no signado *Value1* y almacena el resultado en el registro *Value1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* − *Value2* es cero. Si se especifica el efecto WC, la bandera C se activa (1) si el resultado de la resta es un prestado no signado (32-bit sobre flujo). El resultado se escribe en *Value1* a menos que el efecto NR se especifique.

Para restar no signados, valores multi-long, use SUB seguido de SUBX. Ver SUBX en Pág. 368 para mayor información.

SUBABS

instrucción: *Resta* un valor absoluto de otro valor.

SUBABS *Value*, ⟨#⟩ *SValue*

Resultado: La diferencia de *Value* y el absolute del signado *SValue* se almacenan en *Value*.

- *Value* (campo-d) es el registro que contiene el valor a restar del absoluto *SValue*, y es el destino en el cual se escribe el resultado.
- *SValue* (campo-s) es un registro o literal 9-bit cuyo valor absolute se resta de *Value*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100011 001i 1111	ddddddddd	sssssssss	D - |S| = 0	Unsigned Borrow[1]	Written	4

[1] If S is negative, C Result is the inverse of unsigned *carry* (for D + S).

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0003; 3	$FFFF_FFFC; -4	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	0
$0000_0003; 3	$FFFF_FFFD; -3	–	–	wz wc	$0000_0000; 0	1	1
$0000_0003; 3	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	1
$0000_0003; 3	$FFFF_FFFF; -1	–	–	wz wc	$0000_0002; 2	0	1
$0000_0003; 3	$0000_0002; 2	–	–	wz wc	$0000_0001; 1	0	0
$0000_0003; 3	$0000_0003; 3	–	–	wz wc	$0000_0000; 0	1	0
$0000_0003; 3	$0000_0004; 4	–	–	wz wc	$FFFF_FFFF; 4,294,967,295	0	1

[1] El destino se trata como un valor no signado

Explicación

SUBABS resta el absoluto de *SValue* de *Value* y almacena el resultado en el registro *Value*.

Si se especifico el efecto WZ, la bandera Z se activa (1) si *Value* − |*SValue*| es igual a cero. Si se especifica el efecto WC, la bandera C se activa (1) si la resta resultante es un prestado no signado (32-bit sobre flujo). El resultado se escribe en *Value* a menos que el efecto NR se especifique.

SUBS

instrucción: Resta dos valores signados.

SUBS *SValue1*, ⟨#⟩ *SValue2*

Resultado: La diferencia del signado *SValue1* y el signado *SValue2* se almacena en *Value1*.

- *SValue1* (campo-d) es el registro que contiene el valor a restar de *SValue2*, y es el destino en el cual se escribe el resultado.
- *SValue2* (campo-s) es un registro o literal de 9-bit cuyo valor se resta de *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110101 001i 1111 ddddddddd sssssssss	D - S = 0	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0002; 2	–	–	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFE; -2	–	–	wz wc	$0000_0001; 1	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	–	–	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$0000_0002; 2	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFE; 2,147,483,646	$FFFF_FFFF; -1	–	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	0
$7FFF_FFFE; 2,147,483,646	$FFFF_FFFE; -2	–	–	wz wc	$8000_0000; -2,147,483,648	0	1

Explicación

SUBS resta el signado *SValue2* del signado *SValue1* y almacena el resultado en el registro *SValue1*.

Si se especifico el efecto WZ, la bandera Z se activa (1) si *SValue1* − *SValue2* es igual a cero. Si se especifico el efecto WC, la bandera C se activa (1) si el resultado de la resta resulta en un sobre flujo signado. El resultado se escribe en *SValue1* a menos que se especifique el efecto NR.

Para restar signados, valores multi-long, use SUB, posiblemente SUBX, y finalmente SUBSX. Ver SUBSX en Pág. 366 para mayor información.

SUBSX

instrucción: Resta el valor signado mas C de otro valor signado.

SUBSX *SValue1*, ⟨#⟩ *SValue2*

Resultado: La diferencia de signado *SValue1*, y signado *SValue2* mas la bandera C, se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor a restar de *SValue2* mas C, y es el destino en el cual se escribe el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor mas C se resta de *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110111 001i 1111 ddddddddd sssssssss	Z & (D–(S+C) = 0)	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	0	0	wz wc	$0000_0000; 0	0	0
$0000_0001; 1	$0000_0001; 1	1	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	0	wz wc	$0000_0000; 0	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	x	1	wz wc	$FFFF_FFFF; -1	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	x	0	wz wc	$8000_0000; -2,147,483,648	0	0
$8000_0001; -2,147,483,647	$0000_0001; 1	x	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	x	0	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	x	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	0

Explicación

SUBSX (Subtract Signed, Extended) resta el valor signado de *SValue2* mas C de *SValue1*, y almacena el resultado en el registro *SValue1*. La instrucción SUBSX se usa para desarrollar restas signadas multi-long; restas de 64-bit, por ejemplo:

En una operación signada multi-long, la primera instrucción es no signada (Ej.: SUB), cualquier instrucción intermedia es no signada, extendida (Ej.: SUBX), t la ultima instrucción

es signada, extendida (Ej.: SUBSX). asegúrese de usar WC, opcionalmente WZ, en las instrucciones SUB y SUBX.

Por ejemplo, una resta signada doble long (64-bit) se podría ver como sigue:

```
  sub     XLow,  YLow   wc wz    'Resta longs bajos; guarda C y Z
  subsx   XHigh, YHigh             'Resta longs altos
```

Despues de ejecutar lo anterior, el resultado doble long (64-bit) esta en los registros *XHigh:XLow*. Si *XHigh:XLow* inicio como $0000_0000:0000_0001 (1) y *YHigh:YLow* fue $0000_0000:0000_0002 (2) el resultado en *High:XLow* será $FFFF_FFFF:FFFF_FFFF (-1). Esto se demuestra a continuación:

```
                        Hexadecimal              Decimal
                      (high)     (low)
   (XHigh:XLow)     $0000_0000:0000_0001                 1
 - (YHigh:YLow)   - $0000_0000:0000_0002     -           2
                  ---------------------     ---------------
                  = $FFFF_FFFF:FFFF_FFFF     =          -1
```

Una resta signada triple-long (96-bit) seria similar pero con una instrucción SUBX insertada entre las instrucciones SUB y SUBSX:

```
  sub     XLow,  YLow   wc wz    'Resta longs bajos; guarda C y Z
  subx    XMid,  YMid   wc wz    'Resta los longs medios; guarda C y
Z
  subsx   XHigh, YHigh             'Resta los longs altos
```

Por supuesto puede ser necesario especificar los efectos WC y WZ en la instrucción final, SUBSX, para poder observar el resultado de cero o la condición de sobre flujo. Observe que durante el multi paso la bandera Z siempre indica si el resultado cambio a cero, pero la bandera C indica el prestado no signado hasta la instrucción final, SUBSX, en la cual se indica el sobre flujo signado.

Para SUBSX, si el efecto WZ se especifica, la bandera Z se activa (1) si Z estaba activa previamente y *SValue1* − (*SValue2* + C) es cero (use WC y WZ antes de SUB y SUBX). Si se especifico el efecto WC, la bandera C se activa (1) si el resultado de la r esta es un sobre flujo signado. El resultado se escribe en *SValue1* a menos que se especifique NR.

SUBX

instrucción: Resta el valor no signado mas C de otro valor no signado

SUBX *Value1*, ⟨#⟩ *Value2*

Resultado: La diferencia de un no signado *Value1*, y un no signado *Value2* mas la bandera C, se almacena en *Value1*.

- *SValue1* (campo-d) es el registro que contiene el valor a restar de *Value2* mas C, y es el destino en el cual se escribe el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor mas C se resta de *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
110011 001i 1111	ddddddddd	sssssssss	Z & (D–(S+C) = 0)	Unsigned Borrow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino [1]	Fuente [1]	Z	C	Efectos	Destino	Z	C
s0000_0001; 1	s0000_0001; 1	0	0	wz wc	s0000_0000; 0	0	0
s0000_0001; 1	s0000_0001; 1	1	0	wz wc	s0000_0000; 0	1	0
s0000_0001; 1	s0000_0001; 1	×	1	wz wc	sFFFF_FFFF; 4,294,967,295	0	1

[1] Fuente y destino se tratan como valores no signados.

Explicación

SUBX (Subtract Extended) reata el valor no signado de *Value2* mas C del valor no signado *Value1* y almacena el resultado en el registro *Value1*. La instrucción SUBX se usa para desarrollar restas multi long; reatas de 64-bit, por ejemplo.

En una operación multi long, la primer instrucción es no signada (Ej.: SUB), cualquier instrucción media es no signada, extendida (Ej.: SUBX), y la ultima instrucción es no signada, extendida (SUBX) o signada, extendida (SUBSX) dependiendo de la naturaleza de los valores originales. Discutiremos los valores multi long signados; Ver SUBSX en Pág. 366 para ejemplos con valores multi-long signados. asegúrese de usar los efectos WC, opcionalmente WZ, antes de las instrucciones SUB y SUBX.

Por ejemplo, una resta doble-long no signado (64-bit) podría verse como sigue::

```
sub     XLow,  YLow   wc wz    'Resta longs bajos; guarda C y Z
subx    XHigh, YHigh            'Resta longs altos
```

Despues de ejecutar lo anterior, el resultado doble long (64-bit) se encuentra en los registros *XHigh:XLow*. Si *XHigh:XLow* inicio como $0000_0001:0000_0000 (4,294,967,296) y *YHigh:YLow* era $0000_0000:0000_0001 (1) el resultado en *High:XLow* será $0000_0000:FFFF_FFFF (4,294,967,295). Se demuestra a continuación.

```
                        Hexadecimal              Decimal
                      (high)     (low)
   (XHigh:XLow)     $0000_0001:0000_0000       4,294,967,296
 - (YHigh:YLow)   - $0000_0000:0000_0001     -             1
                  ----------------------     ---------------
                  = $0000_0000:FFFF_FFFF     = 4,294,967,295
```

Por supuesto, puede ser necesario especificar los efectos **WC** y **WZ** en la instrucción final, **SUBX**, para poder observar el resultado cero o la condición prestado no signado.

Para **SUBX**, si el efecto **WZ** se especifica, la bandera Z se activa (1) si Z estaba activa previamente y *Value1* − (*Value2* + C) es cero (use **WC** y **WZ** antes de **SUB** y **SUBX**). Si se especifico el efecto **WC**, la bandera C se activa (1) si el resultado de la resta es prestado no signado (sobre flujo 32-bit). El resultado se escribe en *Value1* a menos que se especifique **NR**.

SUMC

instrucción: Suma un valor signado con otro cuyo signo se invierte dependiendo de C.

SUMC *SValue1*, ⟨#⟩ *SValue2*

Resultado: La suma de un signado *SValue1* y ±*SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor de la suma con –*SValue2* o *SValue2*, y es el destino donde se almacena el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor es afectado en signo por C y se suma a *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100100 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	–	0	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	1	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	–	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	0	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	–	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	–	0	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	–	0	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$0000_0001; 1	–	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	0	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	–	1	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	–	0	wz wc	$8000_0000; -2,147,483,648	0	1

Explicación

SUMC (Sum with C-affected sign) suma el valor signado de *SValue1* a –*SValue2* (si C = 1) o a *SValue2* (si C = 0) y almacena el resultado en *SValue1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *SValue1* ± *SValue2* es cero. Si se especifica el efecto WC, la bandera C se active si el resultado de la suma en un sobre flujo signado. El resultado se escribe en *SValue1* a menos que el efecto NR se especifique.

SUMNC

instrucción: Suma un valor signado con otro cuyo signo se invierte dependiendo de !C.

SUMNC *SValue1*, ⟨#⟩ *SValue2*

Resultado: La suma de un signado *SValue1* y ±*SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor de la suma con –*SValue2* o *SValue2*, y es el destino donde se almacena el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor es afectado en signo por !C y se suma a *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100101 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	-	0	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	-	1	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	-	1	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	-	0	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	-	1	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	-	1	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	-	0	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	-	1	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	-	1	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	-	0	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	-	1	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	-	1	wz wc	$8000_0000; -2,147,483,648	0	1

Explicación

SUMNC (Sum with !C-affected sign) suma el valor signado de *SValue1* a *SValue2* (si C = 1) o a –*SValue2* (si C = 0) y almacena el resultado en *SValue1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *SValue1* ± *SValue2* es cero. Si se especifica el efecto WC, la bandera C se active si el resultado de la suma en un sobre flujo signado. El resultado se escribe en *SValue1* a menos que el efecto NR se especifique.

SUMNZ

instrucción: Suma un valor signado con otro cuyo signo se invierte dependiendo de !Z.

SUMNZ *SValue1*, ⟨#⟩ *SValue2*

Resultado: La suma de un signado *SValue1* y ±*SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor de la suma con –*SValue2* o *SValue2*, y es el destino donde se almacena el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor es afectado en signo por !Z y se suma a *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100111 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	0	–	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$0000_0001; 1	1	–	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$FFFF_FFFF; -1	1	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	–	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$0000_0001; 1	1	–	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	0	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$0000_0001; 1	1	–	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	–	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	1	–	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	1	–	wz wc	$8000_0000; -2,147,483,648	0	1

Explicación

SUMNZ (Sum with !Z-affected sign) suma el valor signado de *SValue1* a *SValue2* (si Z = 1) o a –*SValue2* (si Z = 0) y almacena el resultado en *SValue1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *SValue1* ± *SValue2* es cero. Si se especifica el efecto WC, la bandera C se activa si el resultado de la suma en un sobre flujo signado. El resultado se escribe en *SValue1* a menos que el efecto NR se especifique.

SUMZ

instrucción: Suma un valor signado con otro cuyo signo se invierte dependiendo de Z.

SUMZ *SValue1*, ⟨#⟩ *SValue2*

Resultado: La suma de un signado *SValue1* y ±*SValue2* se almacena en *SValue1*.

- *SValue1* (campo-d) es el registro que contiene el valor de la suma con –*SValue2* o *SValue2*, y es el destino donde se almacena el resultado.
- *SValue2* (campo-s) es un registro o literal 9-bit cuyo valor es afectado en signo por Z y se suma a *SValue1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
100110 001i 1111 ddddddddd sssssssss	D ± S = 0	Signed Overflow	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0001; 1	$0000_0001; 1	0	-	wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	1	-	wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$FFFF_FFFF; -1	0	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	0	-	wz wc	$FFFF_FFFE; -2	0	0
$FFFF_FFFF; -1	$FFFF_FFFF; -1	1	-	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; -1	$0000_0001; 1	0	-	wz wc	$0000_0000; 0	1	0
$8000_0000; -2,147,483,648	$0000_0001; 1	0	-	wz wc	$8000_0001; -2,147,483,647	0	0
$8000_0000; -2,147,483,648	$0000_0001; 1	1	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$8000_0000; -2,147,483,648	$FFFF_FFFF; -1	0	-	wz wc	$7FFF_FFFF; 2,147,483,647	0	1
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	0	-	wz wc	$7FFF_FFFE; 2,147,483,646	0	0
$7FFF_FFFF; 2,147,483,647	$FFFF_FFFF; -1	1	-	wz wc	$8000_0000; -2,147,483,648	0	1
$7FFF_FFFF; 2,147,483,647	$0000_0001; 1	0	-	wz wc	$8000_0000; -2,147,483,648	0	1

Explicación

SUMZ (Sum with Z-affected sign) suma el valor signado de *SValue1* a –*SValue2* (si Z = 1) o a *SValue2* (si Z = 0) y almacena el resultado en *SValue1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *SValue1* ± *SValue2* es cero. Si se especifica el efecto WC, la bandera C se activa si el resultado de la suma en un sobre flujo signado. El resultado se escribe en *SValue1* a menos que el efecto NR se especifique.

Símbolos

Los símbolos de la Tabla 3-6 sirven a uno o mas propósitos especiales en el código Ensamblador Propeller. Para símbolos Spin Ver, Símbolos en Pág. 211. Cada propósito de símbolo se describe brevemente con referencia a otras secciones que lo describen directamente o se usa en ejemplos

Tabla 3-6: Símbolos

Símbolo	propósito(s)
%	Indicador binario. Se usa para indicar que un valor se expresa en binario (Base-2). Ver Representaciones de Valores en Pág. 47.
%%	Indicador cuaternario: Se usa para indicar que un valor esta expresado en cuaternario (base-4). Ver Representaciones de Valores Representaciones de Valores en Pág. 47.
$	1) Indicador Hexadecimal: se usa para indicar que es valor esta expresado en hexadecimal (base-16). Ver Representaciones de Valores 2) en Pág. 47. 3) Indicador Ensamblador Here: se usa para indicar la dirección actual en instrucciones Ensamblador. Ver **JMP** en Pág. 306.
''	Indicador de cadena: Se usa para iniciar o terminar una cadena de texto. Ver Bloque de Datos en Pág. 102.
_	1) Delimitador: se una como delimitador de grupo en valores constantes (donde una coma o punto normalmente se usaría como delimitador). Ver Representaciones de Valores 2) en Pág. 47. 3) Guión Bajo: Se usa como parte de un símbolo Reglas de en Pág. 47.
#	Indicador literal ensamblador. Se usa para indicar una expresión o símbolo en vez de una dirección de registro Ver ¿De donde obtiene sus datos una instrucción? en Pág. 245.

Tabla 3-6: (continuación)	
Símbolo	**propósito(s)**
:	Indicador de etiqueta local: aparece inmediatamente antes de una etiqueta local. Ver Etiquetas Globales y Locales en Pág. 247.
,	Delimitador de Lista: se usa para separar piezas en una lista. Ver la sección **DAT** Declaración de Data(Sintaxis 1) en Pág. 103.
'	Indicador de comentario: se usa para ingresar líneas sencillas de comentarios de código para propósito de vista de código. Ver "Usando la Herramienta Propeller".
''	Indicador de comentario de documento: se usa para ingresar linease de comentario del documento (texto no compilado) para propósitos de vista.
{ }	Indicador de comentarios multi código: se usa para ingresar multi líneas de comentarios de código (texto no compilado).
{{ }}	Indicador de comentarios multilínea para documento: se usa para ingresar comentarios de documento multilíneas. Ver Herramienta Propeller en el archive de ayuda del software.

TEST

instrucción: Hace la operación AND de dos valores para afectar las banderas únicamente.

TEST *Value1*, ⟨#⟩ *Value2*

Resultado: Opcionalmente el resultado cero y de paridad se escribe a las banderas Z y C

- *Value1* (campo-d) es el registro que contiene el valor a hacer AND con *Value2*.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor se hace AND con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011000 000i 1111 ddddddddd sssssssss	D = 0	Parity of Result	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_0007; 7	–	–	wr wz wc	$0000_0002; 2	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wr wz wc	$0000_000A; 10	0	0

[1] El destino no se escribe a menos que el efecto WR se especifique. NOTA: la instrucción TEST con el efecto WR es una instrucción AND.

Explicación

TEST es similar a AND excepto que no escribe el resultado en *Value1*; hace la operación AND de los valores en *Value1* y *Value2* y opcionalmente almacena el resultado cero y paridad del resultado en las banderas Z y C.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* AND *Value2* es cero. Si se especifica el efecto WC, la bandera C se activa (1) si el resultado contiene un numero impar de bits altos (1).

TESTN

instrucción: Hace la operación AND de un valor con el NOT de otro para afectar banderas.

TESTN *Value1*, ⟨#⟩ *Value2*

Resultado: Opcionalmente se escribe el resultado cero y paridad en las banderas Z y C.

- *Value1* (campo-d) es el registro que contiene el valor a hacer AND con !*Value2*.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor se invierte (bitwise NOT) y se hace AND con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011001 000i 1111	ddddddddd	sssssssss	D = 0	Parity of Result	Not Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$F731_125A; -147,778,982	$FFFF_FFFA; -6	-	-	wr wz wc	$0000_0000; 0	1	0
$F731_125A; -147,778,982	$FFFF_FFF8; -8	-	-	wr wz wc	$0000_0002; 2	0	1
$F731_125A; -147,778,982	$FFFF_FFF0; -16	-	-	wr wz wc	$0000_000A; 10	0	0

[1] El destino no se escribe a menos que el efecto WR se especifique. NOTA: la instrucción TESTN con el efecto WR es una instrucción ANDN.

Explicación

TESTN es similar a ANDN excepto que no escribe el resultado a *Value1*; desarrolla la operación AND NOT de *Value2* en *Value1* y opcionalmente almacena el resultado cero y paridad en las banderas Z y C.

Si se especifico el efecto WZ, la bandera Z se activa (1) si *Value1* AND NOT *Value2* es cero. Si se especifico el efecto WC, la bandera C se activa (1) si el resultado contiene un numero impar de bits altos (1).

TJNZ

Instrucción: Prueba un valor y salta a una dirección si no es cero.

TJNZ *Value*, ⟨#⟩ *Address*

- *Value* (campo-d) es el registro a probar.
- *Address* (campo-s) es el registro o literal 9-bit cuyo valor es la dirección a la cual saltar si *Value* contiene un valor diferente de cero.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111010 000i 1111 ddddddddd sssssssss	D = 0	0	Not Written	4 or 8

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0000; 0	$----_----; -	-	-	wr wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$----_----; -	-	-	wr wz wc	$0000_0001; 1	0	0

[1] El destino no se escribe a menos que se especifique el efecto WR.

Explicación

TJNZ prueba el registro *Value* y salta a *Address* si contiene un numero diferente de cero.

Cuando el efecto WZ se especifica, la bandera Z se activa (1) se el registro *Value* contiene cero.

TJNZ requiere un monto diferente de ciclos de reloj dependiendo de si salta o no. Si tiene que saltar toma 4 ciclos de reloj, si no salta sucede que toma 8 ciclos de reloj. Como el ciclo que esta usando TJNZ necesita ser mas rápido, se optimiza de esta forma para velocidad.

TJZ

instrucción: Prueba un valor y salta a una dirección si es cero.

TJZ *Value*, ⟨#⟩ *Address*

- *Value* (campo-d) es el registro a probar
- *Address* (campo-s) es el registro o literal 9-bit cuyo valor es la dirección a saltar ciando *Value* contiene cero.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111011 000i 1111 ddddddddd sssssssss	D = 0	0	Not Written	4 or 8

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0000; 0	$----_----; -	-	-	wr wz wc	$0000_0000; 0	1	0
$0000_0001; 1	$----_----; -	-	-	wr wz wc	$0000_0001; 1	0	0

[1] El destino no se escribe a menos que se especifique el efecto WR.

Explicación

TJZ prueba el registro *Value* y salta a *Address* si contiene cero.

Cuando el efecto WZ se especifica, la bandera Z se activa (1)si el registro *Value* contiene cero.

TJZ requiere un monto diferente de ciclos de reloj dependiendo de si salta o no. Si tiene que saltar toma 4 ciclos de reloj, si no salta sucede que toma 8 ciclos de reloj.

VCFG

Registro: Registro de configuración de video

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* VCFG, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, VCFG ⟨*Effects*⟩

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador deseada. VCFG puede usarse en cualquiera de los campos *DestOperand* o *SrcOperand* de la instrucción
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en y opcionalmente escribir a, el registro VCFG en *DestOperand*.
- *DestOperand* es una expresión constante que indica el registro en el que se opera y opcionalmente se escribe, usando el valor de VCFG en *SrcOperand*.

Explicación

VCFG es uno de dos registros (VCFG y VSCL) que afectan el comportamiento del generador de video del cog. Cada cog tiene un generador de video que facilita la transmisión de imágenes de video a un rango constante. El registro VCFG contiene la configuración del generador de video.

El siguiente código activa el registro de la configuración de video para habilitar en modo compuesto a con cuatro colores, banda base chroma (color) habilitado) en el grupo de pin 1, 4 pins bajos (los cuales son P11:P8)

```
                mov     vcfg,  VidCfg

VidCfg    long  %0_10_1_0_1_000_00000000000_001_0_00001111
```

Ver Registros, Pág. 351, y la sección de spin VCFG, Pág. 218, para mayor información.

VSCL

Registro: Registro de escala de video

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* VSCL, *SrcOperand* ⟨*Effects*⟩

DAT
⟨*Label*⟩ ⟨*Condition*⟩ *Instruction* *DestOperand*, VSCL ⟨*Effects*⟩

- *Label* es una etiqueta opcional, Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis, Pág. 255.
- *Instruction* es la instrucción ensamblador. VSCL puede usarse en los campos *DestOperand* o *SrcOperand* de la instrucción.
- *SrcOperand* es una expresión constante usada por *Instruction* para operar en y opcionalmente escribir a, el registro VSCL en *DestOperand*.
- *DestOperand* es una expresión constante que indica el registro en el que se opera, y opcionalmente se escribe, usando el valor de VSCL en *SrcOperand*.

Explicación

VSCL es uno de dos registros (VCFG y VSCL) que afectan el comportamiento del generador de video del cog. Cada cog tiene un generador de video que facilita la transmisión de datos de imágenes de video a un rango constante. El registro VSCL active el rango al cual los datos de video son generados.

El siguiente código active el registro de escala de video para 160 pixeles y 2560 frames (por un marco de 16 pixeles por 2 bit de color). Por supuesto el rango actual al cual los pixeles se transmiten dependen de la frecuencia del PLLA en combinación con el factor de escala.

```
                mov     vcfg,  VscCfg

VscCfg    long  %000000000000_10100000_101000000000
```

Ver Registros, Pág. 351, y la sección de lenguaje spin VSCL, Pág. 221, para mayor información.

WAITCNT

instrucción: Detiene la ejecución del cog temporalmente.

WAITCNT *Target*, ⟨#⟩ *Delta*

Result: *Target* + *Delta* se almacena en *Target*.

- *Target* (campo-d) es el registro con el valor objetivo a comparar contra el contador del sistema (CNT). Cuando el contador del sistema alcanza el valor *Target*, *Delta* se suma a *Target* y la ejecución continua en la siguiente instrucción.
- *Delta* (campo-s) es el registro o literal de 9-bit cuyo valor se suma a *Target* en preparación para la siguiente instrucción WAITCNT. Esto genera una ventana sincronizada de retraso.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111110 001i 1111	ddddddddd	sssssssss	Result = 0	Unsigned Carry	Written	5+

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wz wc	$0000_0000; 0	1	0
$FFFF_FFFF; 4,294,967,295	$0000_0001; 1	–	–	wz wc	$0000_0000; 0	1	1
$0000_0000; 0	$0000_0001; 1	–	–	wz wc	$0000_0001; 1	0	0

Explicación

WAITCNT, "Wait for System Counter," es una de cuatro instrucciones (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener un cog hasta cumplir con una condición. La instrucción WAITCNT detiene el cog hasta que el contador global del sistema iguala el valor en *Target*, entonces suma *Delta* a *Target* y continua la ejecución en la siguiente instrucción. La instrucción WAITCNT se comporta similar a la instrucción WAITCNT spin para retrasos sincronizados. Ver WAITCNT en Pág. 223.

Si se especifico el efecto WZ, la bandera Z se active (1) si la suma de *Target* y *Delta* es cero. Si el efecto WC se especifica, la bandera C se activa (1) si la suma de *Target* y *Delta* resulta en un acarreo de 32-bit (sobre flujo). El resultado se escribe en *Target* a menos que el efecto NR se especifique.

WAITPEQ

instrucción: Detiene la ejecución de un cog hasta que un pin E/S coincida con el estado indicado

WAITPEQ *State*, ⟨#⟩ *Mask*

- *State* (campo-d) es el registro con el estado objetivo a comparar contra INx hecho AND con *Mask*.
- *Mask* (campo-s) es el registro o literal 9-bit cuyo valor se hace AND con INx antes de la comparación con *State*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111100 000i 1111 ddddddddd sssssssss	---	---	Not Written	5+

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wr wz wc	$0000_0000; 0	1	0
$0000_0000; 0	$0000_0001; 1	–	–	wr wz wc	$0000_0001; 1	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0000; 0	$0000_0002; 2	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wr wz wc	$0000_0004; 4	0	0

[1] El destino no se escribe a menos que el efecto WR se proporcione.

Explicación

WAITPEQ, "Wait for Pin(s) to Equal," es una de cuatro instrucciones (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener la ejecución del cog hasta que se cumple una condición. La instrucción WAITPEQ detiene el cog hasta que el resultado de INx hecho AND con *Mask* iguala el valor del registro *State*. INx es INA o INB dependiendo del valor de la bandera C en la ejecución; INA si C = 0, INB si C = 1 (el P8X32A es una excepción a la regla; siempre prueba INA).

La instrucción WAITPEQ se comporta similar a la instrucción Spin WAITPEQ; ver WAITPEQ en Pág. 227.

WAITPNE

instrucción: Detiene la ejecución de un cog hasta que los pins E/S no coinciden con el estado indicado.

WAITPNE *State*, ⟨#⟩ *Mask*

- *State* (campo-d) es el registro con el estado objetivo a comparar contra INx hecho AND con *Mask*.
- *Mask* (campo-s) es el registro o literal 9-bit cuyo valor se hace AND con INx antes de la comparación con *State*.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111101 000i 1111	ddddddddd	sssssssss	---	---	Not Written	5+

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0000; 0	$0000_0000; 0	–	–	wr wz wc	$0000_0000; 0	1	1
$0000_0000; 0	$0000_0001; 1	–	–	wr wz wc	$0000_0002; 2	0	0
$0000_0001; 1	$0000_0001; 1	–	–	wr wz wc	$0000_0003; 3	0	0
$0000_0000; 0	$0000_0002; 2	–	–	wr wz wc	$0000_0003; 3	0	0
$0000_0002; 2	$0000_0002; 2	–	–	wr wz wc	$0000_0002; 2	0	0

[1] El destino no se escribe a menos que el efecto WR se proporcione.

Explicación

WAITPNE, "Wait for Pin(s) to Not Equal," es una de cuatro instrucciones (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener el cog hasta que se cumpla una condición. La instrucción WAITPNE detiene la instrucción hasta que el resultado de INx hecho AND con *Mask* no coincide con el valor del registro *State*. INx es INA o INB dependiendo del valor de C durante la ejecución; INA si C = 0, INB si C = 1 (el P8X32A es una excepción a la regla; siempre prueba INA). La instrucción WAITPNE se comporta similar a la instrucción Spin WAITPNE; Ver WAITPNE en Pág. 229.

WAITVID

instrucción: Detiene la ejecución de un cog hasta que el generador de video esta disponible para tomar datos de pixeles.

WAITVID *Colors*, ⟨#⟩ *Pixels*

- *Colors* (campo-d) es el registro cuatro valores de color tamaño byte cada uno describe cuatro posibles colores del patrón de pixeles en *Pixels*.
- *Pixels* (campo-s) es el registro o literal 9-bit cuyo valor es el siguiente patrón de 16 pixeles por 2-bit (o 32 pixeles por 1 bit) a desplegar.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
111111 000i 1111 ddddddddd sssssssss	D + S = 0	Unsigned overflow	Not Written	5+

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z	C
$0000_0002; 2	$FFFF_FFFD; -3	-	-	wr wz wc	$FFFF_FFFF; -1	0	0
$0000_0002; 2	$FFFF_FFFE; -2	-	-	wr wz wc	$0000_0000; 0	1	1
$0000_0002; 2	$FFFF_FFFF; -1	-	-	wr wz wc	$0000_0001; 1	0	1
$0000_0002; 2	$0000_0000; 0	-	-	wr wz wc	$0000_0002; 2	0	0

[1] El destino no se escribe a menos que el efecto WR se proporcione.

Explicación

WAITVID, "Wait for Video Generator," es una de cuatro instrucciones (WAITCNT, WAITPEQ, WAITPNE, y WAITVID) que se usan para detener la ejecución del cog hasta que una condición se cumple. La instrucción WAITVID detiene el cog hasta que el hardware del generador de video esta listo para el siguiente dato de pixeles, entonces el generador de video acepta los datos (*Colors* y *Pixels*) y el cog continua con la ejecución en la siguiente instrucción. La instrucción WAITVID se comporta similar a la instrucción spin WAITVID; Ver WAITVID en Pág. 230.

Si el efecto WZ se especifica, la bandera Z se activa (1) si *Colors* y *Pixels* son iguales.

asegúrese de iniciar el modulo generador de video y el contador A antes de ejecutar la instrucción WAITVID o esperara por siempre. Ver VCFG en Pág. 218, VSCL en Pág. 221, y CTRA, CTRB en Pág. 98.

WC

Efecto: Hace que la instrucción ensamblador modifique la bandera C.

⟨Label⟩ ⟨Condition⟩ Instruction Operands **WC**

Resultado: El estado de la bandera C se actualiza con la ejecución de *Instruction*.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Instruction* es la instrucción ensamblador.
- *Operands* es cero, uno o dos operandos según requiere *Instruction*.

Explicación

WC (Write C flag) es uno de cuatro efectos opcionales (**NR**, **WR**, **WZ**, y **WC**) que influye en el comportamiento de las instrucciones ensamblador. **WC** hace que cuando se ejecuta una instrucción ensamblador se modifique la bandera C de acuerdo al resultado.

Por ejemplo, la instrucción **CMP** (Compare Unsigned) compara dos valores (fuente y destino) pero no escribe automáticamente el resultado a las banderas C y Z. Usted puede determinar si el valor destino es menor que el valor fuente usando la instrucción **CMP** con el efecto **WC**:

```
cmp     value1, value2   WC      'C = 1 si value1 < value2
```

La instrucción anterior **CMP** compara `value1` con `value2` y activa C alto (1) si `value1` es menor que `value2`.

Ver Efectos en Pág. 298 para mayor información.

WR

Efecto: Hace que la instrucción ensamblador escriba un resultado

⟨Label⟩ ⟨Condition⟩ Instruction Operands WR

Resultado: El registro destino de *Instruction* se cambia al valor del resultado.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis en Pág.. 255.
- *Instruction* es la instrucción ensamblador.
- *Operands* es cero, uno o dos operandos según requiere *Instruction*.

Explicación

WR (Write Result) es uno de cuatro efectos opcionales (WC, WZ, NR, y WR) que influyen en el comportamiento de las instrucciones ensamblador. WR hace que cuando se ejecuta una instrucción ensamblador escriba su resultado en el registro destino.

Por ejemplo, por defecto la instrucción COGINIT (Cog Initialize) no escribe un resultado en el registro destino. Esto esta bien cuando se quiere iniciar un cog con un ID especifico, pero cuando esta preguntando por iniciar el siguiente cog disponible (Ej. El bit 3 del registro destino esta en alto) quizá quiera saber cual cog se inicio, por ID. Usted puede obtener el ID del Nuevo cog de la instrucción COGINIT usando el efecto WR:

```
  coginit   launch_value   WR     'Inicia Nuevo cog, obtiene el ID
de regreso
```

Asumiendo que launch_value apunta a un registro que contiene en bit alto en el bit 3, la instrucción COGINIT inicia el siguiente cog disponible y escribe su ID de regreso en el registro launch_value.

Ver Efectos en Pág. 298 para mayor información.

WRBYTE

instrucción: Escribe un byte a memoria principal.

WRBYTE *Value*, ⟨#⟩ *Address*

- *Value* (campo-d) es el registro que contiene el valor 8-bit a escribir en memoria.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal en la cual escribir.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000000 000i 1111 ddddddddd sssssssss	---	---	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	n/a	1	0

[1] Salida destino no existe cuando se incluye el efecto WR ya que cambiara de WRBYTE a RDBYTE.
[2] La bandera Z se activa (1) a menos que la dirección de memoria principal este en la frontera long.

Explicación

WRBYTE Sincroniza al hub y escribe el byte mas bajo en *Value* a la memoria principal *Address*.

El efecto WR no se puede usar con WRBYTE ya que lo cambiaria a una instrucción RDBYTE.

WRBYTE es una instrucción de hub. Las instrucciones de hub toman entre 7 y 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al hub y el momento en el que se ejecuta la instrucción. Ver Hub en Pág. 24 para mayor información.

WRLONG

instrucción: Escribe un long a memoria principal.

WRLONG *Value*, ⟨#⟩ *Address*

- *Value* (campo-d) es el registro que contiene el valor 32-bit a escribir en memoria.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal en la cual escribir.

Tabla Opcode:

–INSTR– ZCRI –CON–	–DEST–	–SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000010 000i 1111	ddddddddd	sssssssss	---	---	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	wz wc	n/a	0	0

[1] Salida destino no existe cuando se incluye el efecto WR ya que cambiara de WRLONG a RDLONG.
[2] La bandera Z se activa (1) a menos que la dirección de memoria principal este en la frontera long.

Explicación

WRLONG sincroniza el hub y escribe el long *Value* a la memoria principal *Address*.

El efecto WR no se puede usar con WRLONG ya que lo cambiaria a una instrucción RDLONG.

WRLONG es una instrucción de hub. Las instrucciones de hub toman entre 7 y 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al hub y el momento en el que se ejecuta la instrucción. Ver Hub en Pág. 24 para mayor información

WRWORD

instrucción: Escribe un word a memoria principal.

WRWORD *Value*, ⟨#⟩ *Address*

- *Value* (campo-d) es el registro que contiene el valor 16-bit a escribir en memoria.
- *Address* (campo-s) es un registro o literal de 9-bit cuyo valor es la dirección de memoria principal en la cual escribir.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
000001 000i 1111 ddddddddd sssssssss	---	---	Not Written	7..22

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino[1]	Z[2]	C
$----_----; -	$----_----; -	-	-	WZ WC	n/a	1	0

[1] Salida destino no existe cuando se incluye el efecto WR ya que cambiara de WRWORD a RDWORD.
[2] La bandera Z se activa (1) a menos que la dirección de memoria principal este en la frontera long.

Explicación

WRWORD sincroniza el hub y escribe el word mas bajo en *Value* a la memoria principal *Address*.

El efecto WR no se puede usar con WRWORD ya que lo cambiaria a una instrucción RDWORD.

WRWORD es una instrucción de hub. Las instrucciones de hub toman entre 7 y 22 ciclos de reloj para ejecutarse, dependiendo de la relación entre la ventana de acceso al hub y el momento en el que se ejecuta la instrucción. Ver Hub en Pág. 24 para mayor información

WZ

Efecto: Hace que la instruction ensamblador modifique la bandera Z.

⟨Label⟩ ⟨Condition⟩ Instruction Operands **WZ**

Resultado: La bandera Z se actualice con el estado de la ejecución de *Instruction*.

- *Label* es una etiqueta opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Condition* es una condición opcional. Ver Elementos Comunes de Sintaxis en Pág. 255.
- *Instruction* es la instruction ensamblador.
- *Operands* es cero, una o dos operandos según requiera *Instruction*.

Explicación

WZ (Write Z flag) es uno de cuatro efector opcionales (**NR**, **WR**, **WC**, y **WZ**) que influye en el comportamiento de las instrucciones ensamblador. **WZ** hace que cuando se ejecuta una instrucción ensamblador modifique la bandera Z de acuerdo al resultado.

Por ejemplo, la instrucción **CMP** (Compare Unsigned) compara dos valores (destino y fuente) pero no escribe automáticamente el resultado a las banderas C y Z. Usted puede determinar si el valor destino es igual a la fuente usando la instrucción **CMP** con el efecto **WZ**:

```
cmp     value1, value2   WZ      'Z = 1 si value1 = value2
```

El ejemplo anterior **CMP** compara `value1` con `value2` y activa Z alto (1) si `value1` es igual a `value2`.

Ver Efectos en Pág. 298 para mayor información.

XOR

instrucción: Hace la operación XOR de dos valores

XOR *Value1*, ⟨#⟩ *Value2*

Resultado: *Value1* XOR *Value2* se almacena en *Value1*.

- *Value1* (campo-d) es el registro que contiene el valor a hacer XOR con *Value2* y es el destino en el cual se escribirá el resultado.
- *Value2* (campo-s) es un registro o literal 9-bit cuyo valor se hace XOR con *Value1*.

Tabla Opcode:

–INSTR– ZCRI –CON– –DEST– –SRC–	Resultado Z	Resultado C	Resultado	Ciclos
011011 001i 1111 ddddddddd sssssssss	Result = 0	Parity of Result	Written	4

Tabla de verdad:

Entrada					Salida		
Destino	Fuente	Z	C	Efectos	Destino	Z	C
$0000_000A; 10	$0000_0005; 5	–	–	wz wc	$0000_000F; 15	0	0
$0000_000A; 10	$0000_0007; 7	–	–	wz wc	$0000_000D; 13	0	1
$0000_000A; 10	$0000_000A; 10	–	–	wz wc	$0000_0000; 0	1	0
$0000_000A; 10	$0000_000D; 13	–	–	wz wc	$0000_0007; 7	0	1
$0000_000A; 10	$0000_000F; 15	–	–	wz wc	$0000_0005; 5	0	0

Explicación

XOR (bitwise exclusive OR) hace la operación XOR del valor en *Value2* con el valor *Value1*.

Si se especifica el efecto WZ, la bandera Z se activa (1) si *Value1* XOR *Value2* es cero. Si el efecto WC se especifica, la bandera C se activa (1) si el resultado contiene un numero impar de bits altos (1). El resultado se escribe a *Value1* a menos que se especifique el efecto NR.

Apéndice A: Lista de Palabras Reservadas

Estas palabras estas siempre reservadas ya sean para programación en Spin o Propeller.

Tabla A-0-1: Lista de Palabras Reservadas Propeller

_CLKFREQ[s]	CONSTANT[s]	IF_NC_AND_NZ[a]	MIN[a]	PLL4X[s]	SUBSX[a]
_CLKMODE[s]	CTRA[d]	IF_NC_AND_Z[a]	MINS[a]	PLL8X[s]	SUBX[a]
_FREE[s]	CTRB[d]	IF_NC_OR_NZ[a]	MOV[a]	PLL16X[s]	SUMC[a]
_STACK[s]	DAT[s]	IF_NC_OR_Z[a]	MOVD[a]	POSX[d]	SUMNC[a]
_XINFREQ[s]	DIRA[d]	IF_NE[a]	MOVI[a]	PRI[s]	SUMNZ[a]
ABORT[s]	DIRB[d#]	IF_NEVER[a]	MOVS[a]	PUB[s]	SUMZ[a]
ABS[a]	DJNZ[a]	IF_NZ[a]	MUL[a#]	QUIT[s]	TEST[a]
ABSNEG[a]	ELSE[s]	IF_NZ_AND_C[a]	MULS[a#]	RCFAST[s]	TESTN[a]
ADD[a]	ELSEIF[s]	IF_NZ_AND_NC[a]	MUXC[a]	RCL[a]	TJNZ[a]
ADDABS[a]	ELSEIFNOT[s]	IF_NZ_OR_C[a]	MUXNC[a]	RCR[a]	TJZ[a]
ADDS[a]	ENC[a]	IF_NZ_OR_NC[a]	MUXNZ[a]	RCSLOW[s]	TO[s]
ADDSX[a]	FALSE[d]	IF_Z[a]	MUXZ[a]	RDBYTE[a]	TRUE[d]
ADDX[a]	FILE[s]	IF_Z_AND_C[a]	NEG[a]	RDLONG[a]	TRUNC[s]
AND[d]	FIT[a]	IF_Z_AND_NC[a]	NEGC[a]	RDWORD[a]	UNTIL[s]
ANDN[a]	FLOAT[s]	IF_Z_EQ_C[a]	NEGNC[a]	REBOOT[s]	VAR[s]
BYTE[s]	FROM[s]	IF_Z_NE_C[a]	NEGNZ[a]	REPEAT[s]	VCFG[d]
BYTEFILL[s]	FRQA[d]	IF_Z_OR_C[a]	NEGX[d]	RES[a]	VSCL[d]
BYTEMOVE[s]	FRQB[d]	IF_Z_OR_NC[a]	NEGZ[a]	RESULT[s]	WAITCNT[d]
CALL[a]	HUBOP[a]	INA[d]	NEXT[s]	RET[a]	WAITPEQ[d]
CASE[s]	IF[s]	INB[d#]	NOP[a]	RETURN[s]	WAITPNE[d]
CHIPVER[s]	IFNOT[s]	JMP[a]	NOT[s]	REV[a]	WAITVID[d]
CLKFREQ[s]	IF_A[a]	JMPRET[a]	NR[a]	ROL[a]	WC[a]
CLKMODE[s]	IF_AE[a]	LOCKCLR[d]	OBJ[s]	ROR[a]	WHILE[s]
CLKSET[d]	IF_ALWAYS[a]	LOCKNEW[d]	ONES[a#]	ROUND[s]	WORD[s]
CMP[a]	IF_B[a]	LOCKRET[d]	OR[d]	SAR[a]	WORDFILL[s]
CMPS[a]	IF_BE[a]	LOCKSET[d]	ORG[a]	SHL[a]	WORDMOVE[s]
CMPSUB[a]	IF_C[a]	LONG[s]	OTHER[s]	SHR[a]	WR[a]
CMPSX[a]	IF_C_AND_NZ[a]	LONGFILL[s]	OUTA[d]	SPR[s]	WRBYTE[a]
CMPX[a]	IF_C_AND_Z[a]	LONGMOVE[s]	OUTB[d#]	STEP[s]	WRLONG[a]
CNT[d]	IF_C_EQ_Z[a]	LOOKDOWN[s]	PAR[d]	STRCOMP[s]	WRWORD[a]
COGID[d]	IF_C_NE_Z[a]	LOOKDOWNZ[s]	PHSA[d]	STRING[s]	WZ[a]
COGINIT[d]	IF_C_OR_NZ[a]	LOOKUP[s]	PHSB[d]	STRSIZE[s]	XINPUT[s]
COGNEW[s]	IF_C_OR_Z[a]	LOOKUPZ[s]	PI[d]	SUB[a]	XOR[a]
COGSTOP[d]	IF_E[a]	MAX[a]	PLL1X[s]	SUBABS[a]	XTAL1[s]
CON[s]	IF_NC[a]	MAXS[a]	PLL2X[s]	SUBS[a]	XTAL2[s]
					XTAL3[s]

a = Elementos Ensamblador s = Elemento Spin; d = Dual (disponible en ambos lenguajes); # = Reservados para uso futuro

Apéndice B: Ejemplos Matemáticos y Tablas

Multiplicación, División, y Raíz Cuadrada

Multiplicación, división, y raíz cuadrada pueden realizarse usando instrucciones de suma, resta y movimiento. Aquí se muestra una rutina multiplicador no signado que multiplica dos valores 16-bit para alcanzar un producto 32-bit:

```
'' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
multiply        shl     x,#16               'obtiene el multiplicando  en x[31..16]
                mov     t,#16                  'listo para multiplicar 16 bits
                shr     y,#1      wc           'obtiene el multiplicador inicial en
c
:loop  if_c     add     y,x       wc        'si c es activo, suma el multiplicando
al producto
                rcr     y,#1      wc           'pone el siguiente multiplicador en
c, mueve el  producto
                djnz    t,#:loop               'cicla hasta completar
multiply_ret    ret                            'regresa con el producto en
y[31..0]
```

El tiempo de la rutina de arriba podría cortarse en ~1/3 si el ciclo no fuera rotado, repitiendo la secuencia **ADD** / **RCR** y descartando la instrucción **DJNZ**.

La división es como la multiplicación, pero en reversa. Es potencialmente mas compleja, porque una prueba de comparación debe hacerse antes de que se realice la resta. Para remediar esto hay una instrucción especial **CMPSUB** *D, S* la cual prueba si una resta se puede hacer sin ocasionar sobre flujo. SI no hay sobre flujo la resta se realice y la salida C es 1. Si hay sobre flujo D se deja solo y la salida C es 0.

Aqui hay una rutina divisora que divide un valor de 32-bit por un valor 16-bit pata obtener in cociente 16-bit y un sobrante 16-bit.

```
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]
'
divide          shl     y,#15               'Obtiene el divisor en  y[30..15]
                mov     t,#16                  'Listo para los cocientes de 16 bits
:loop           cmpsub  x,y       wc        'y =< x? resta, bit cociente en C
```

```
                rcl     x,#1                    'rota c en cociente, mueve
dividendo
                djnz    t,#:loop                'cicla hasta terminar
divide_ret      ret                             'cociente en x[15..0],
                                                        'sobrante en  x[31..16]
```

Al igual que la rutina multiplicador la rutina divisora podría recodificarse con una secuencia de 16 bits **CMPSUB** + **RCL** para eliminar el **DJNZ** y cortar el tiempo de ejecución en ~1/3. Haciendo tales cambios, la velocidad frecuentemente se gana a expensas del tamaño del código.

Aqui hay una rutina que usa la instrucción **CMPSUB**:

```
'
' Compute square-root of y[31..0] into x[15..0]
'
root            mov     a,#0                    'reinicia acumulador
                mov     x,#0                    'reinicia raíz
                mov     t,#16                   'listo para raíz de 16 bits
:loop           shl     y,#1    wc          'rota los dos bits superiors de y a
acumulador
                rcl     a,#1
                shl     y,#1    wc
                rcl     a,#1
                shl     x,#2                    'determina siguiente bit de raíz
                or      x,#1
                cmpsub  a,x     wc
                shr     x,#2
                rcl     x,#1
                djnz    t,#:loop                'cicla hasta completar
root_ret        ret                             'raíz cuadrada en x[15..0]
```

Muchas funciones matemáticas complejas pueden realizarse con sumas, restas y movimientos. Aun con estos ejemplos específicos dados, estos tipos de algoritmos pueden codificarse de muchas formas para ofrecer una mejor solución.

Tablas Log y Anti-Log ($C000-DFFF)

Las tablas Log y Anti Log son útiles para convertir valores entre su forma numérica y exponencial .

Cuando los números están codificados en forma exponente, operaciones matemáticas simples toman un efecto mas complejo. Por ejemplo “sumar” y “restar” se convierten en “multiplicar” y “dividir”, “mover a la izquierda” en “cuadrado” y “mover a la derecha” se convierte en “raíz cuadrada” y “dividir por 3” producirá una “raíz cúbica”. Una vez que el exponente se convierte a un numero, el resultado será aparente. Este proceso es imperfecto, pero rápido.

Para aplicaciones donde se desarrollan muchas multiplicaciones y divisiones en ausencia de sumas y restas, la codificación exponencial puede acelerar las cosas. La codificación exponencial es también útil para comprimir números en pocos bits- sacrificando resolución en mayores magnitudes. En muchas aplicaciones tales como síntesis de audio, la naturaleza de las señales es logarítmica en ambas frecuencia y magnitudes. Procesar tales datos en forma exponencial es natural y eficiente, y tiene a ofrecer una linearidad simple a lo que es actualmente logarítmico.

El ejemplo de código dado abajo usa cada muestra d tablas verbatim. Mayor resolución podría alcanzarse interpolando linearidad entre las tablas muestra, como el cambio de gradiente es ligero entre muestra y muestra.. El costo podría se código mas largo y velocidad de ejecución mas baja.

Tabla Log ($C000-$CFFF)

La tabla Log contiene datos usados para convertir números no signados en exponentes base-2.

La tabla Log esta hecha de 2,048 words no signados los cuales hacen los exponentes fraccionales base-2. Para usar esta tabla debe determinar primero la porción entera del exponente del numero que esta convirtiendo. Esto es simplemente la posición mas importante del bit. Para $60000000 será 30 ($1E). Esta porción entera siempre encajara en 5 bits. Separando esos 5 bits en el resultado para que ocupen las posiciones 20..16. En nuestro caso de $60000000, ahora tendremos un resultado parcial de $001E0000. Despues justificar al inicio y separar los primeros 11 bits debajo dejando los mas importantes bits en las posiciones 11..1. Esto será $0800 para nuestro ejemplo. Sumar $C000 para la tabla Log base y sabrás la dirección actual word del exponente fraccional. Leyendo el word a $C800, tenemos el valor $95C0. Sumando esto en el resultado parcial $001E95C0 – es $60000000 en forma exponente. Observe que los bits 20..16 hacen la porción entera del exponente, mientras que los bits 15..0 hacen la parte fraccional, con el bit 15 siendo ½, bit 14 siendo ¼, y así sucesivamente hasta el bit 0. El exponente ahora puede manipularse al sumar, restar y mover . Siempre asegúrese que las operaciones matemáticas nunca llevaran al exponente debajo de cero o generara un sobre flujo en el bit 20. De otra forma no se podrá convertir a un numero correctamente.

Aqui hay una rutina que convierte un numero no signado en exponente base-2 usando la tabla Log:

```
' Convierte un numero en exponente
'
' on entry: num holds 32-bit unsigned value
' on exit:  exp holds 21-bit exponente con 5 bits enteros y 16 bits fraccionales
'
```

```
numexp          mov     exp,#0                  'limpia exponente

                test    num,num4        wz      'obtiene la porcion entera del
exponente
                muxnz   exp,exp4                'mientras justifica el numero
    if_z        shl     num,#16
                test    num,num3        wz
                muxnz   exp,exp3
    if_z        shl     num,#8
                test    num,num2        wz
                muxnz   exp,exp2
    if_z        shl     num,#4
                test    num,num1        wz
                muxnz   exp,exp1
    if_z        shl     num,#2
                test    num,num0        wz
                muxnz   exp,exp0
    if_z        shl     num,#1

                shr     num,#30-11              'justifica los bits como words
                and     num,table_mask          'separa bits de la tabla
                add     num,table_log           'suma la dirección de la tabla
log
                rdword  num,num                 'lee la porcion fracional del
exponente
                or      exp,num                 'combina la parte fraccional y
entera

numexp_ret      ret                             '91..106 ciclos
                                        '(variacion debido a sinc HUB en RDWORD)

num4            long    $FFFF0000
num3            long    $FF000000
num2            long    $F0000000
num1            long    $C0000000
num0            long    $80000000
exp4            long    $00100000
exp3            long    $00080000
exp2            long    $00040000
exp1            long    $00020000
exp0            long    $00010000
table_mask      long    $0FFE                   'mascara de tabla offset
table_log       long    $C000                   'base tabla log

num             long    0                       'entrada
exp             long    0                       'salida
```

Tabla Anti-Log ($D000-$DFFF)

La tabla Anti-Log contiene datos usados para convertir exponentes base-2 en números no signados.

La tabla Anti-Log consiste de 2,048 words no signadas que son los mas bajos 16-bits de una mantisa de 17-bit (el bit 17 esta implicado y debe activarse separadamente). Para usar esta tabla mueva los 11 bits superiores de la fracción exponente (bits 15..5) en los bits 11..1 y separe. Sume $D000 para la tabla base Anti-Log. Lea el word en la localidad en el resultado-esto es la mantisa. Despues, mueva la mantisa a la izquierda a los bits 30..15 y active el bit 31– el bit 17 faltante de la mantisa. El ultimo paso es mover el resultado a la derecha por 31 menos el exponente entero en los bits 20..16. El exponente es ahora convertido a un numero no signado.

Aqui se muestra una rutina que convertirá un exponente base-2 en un numero no signado usando la tabla Anti-Log:

```
' Convierte exponente a numero
'
' on entry: exp holds exponente 21-bit con 5 bits entero y 16 bits fraccionales
' on exit:  num holds 32-bit valor no signado
'
expnum          mov     num,exp                 'obtiene exponente en el numero
                shr     num,#15-11                  'justifica fraccion exponente
como offset word
                and     num,table_mask          'separa tabla de bits offset
                or      num,table_antilog           'suma direccion tabla anti-log
                rdword  num,num                 'lee mantisa word in un numero
                shl     num,#15                     'mueve mantisa a los bits 30..15
                or      num,num0                'activa bit superior (bit 17 de la
mantisa)
                shr     exp,#20-4                   'mueve entero exponente a bits
4..0
                xor     exp,#$1F                    'invierte bits para obtener
movimiento count
                shr     num,exp                     'mueve numero a posicion final

expnum_ret      ret                                 '47..62 ciclos
                                                        '(variacion debido a sic HUB
en RDWORD)

num0            long    $80000000               'bit 17 de la mantisa
table_mask      long    $0FFE                   'tabla mascara offset
table_antilog   long    $C000                   'tabla base anti-log

exp             long    0                           'entrada
num             long    0                           'salida
```

Tabla de Seno ($E000-$F001)

La tabla de seno proporciona 2,049 muestras de 16-bit no signados en un rango de 0° a 90°, (resolución 0.0439°).

Un pequeño monto de código ensamblador puede hacer espejo y cambiar la tabla de seno para crear una tabla completa de seno/coseno que tiene 13-bit de resolución angular y 17-bit de resolución de muestreo:

```
' Obtiene seno/coseno
'
'     Cuadrante:  1             2             3             4
'        angulo:  $0000..$07FF  $0800..$0FFF  $1000..$17FF  $1800..$1FFF
'  tabla indice:  $0000..$07FF  $0800..$0001  $0000..$07FF  $0800..$0001
'        espejo:  +offset       -offset       +offset       -offset
'        cambio:  +sample       +sample       -sample       -sample
'
' on entry: sin[12..0] tiene angulo (0° a justo debajo de 360°)
' on exit:  sin tiene valor signado en rango de $0000FFFF ('1') a
' $FFFF0001 ('-1')
'
getcos          add     sin,sin_90          'para coseno, suma 90°
getsin          test    sin,sin_90      wc  'obtiene cuadrante 2|4 en c
                test    sin_sin_180     wz  'obtiene cuadrante 3|4 en nz
                negc    sin,sin             'si cuadrante 2|4, niega offset
                or      sin,sin_table       'o en tabla sin dirección >> 1
                shl     sin,#1              'mueve a izquierda para obtener
dirección word final
                rdword  sin,sin             'lee muestra word de $E000 a $F000
                negnz   sin,sin                 'si cuadrante 3|4, niega muestra
getsin_ret
getcos_ret      ret                         '39..54 ciclos de reloj
                                            '(variacion debido a sincronizacion
HUB en RDWORD)


sin_90          long    $0800
sin_180         long    $1000
sin_table       long    $E000 >> 1          'tabla base seno movida a la derecha

sin             long    0
```

Como en la tabla log y Anti-Log, la interpolación linear podría aplicar a la tabla d seno para alcanzar mejores resoluciones.

Indice Alfabético en Inglés

Assignment

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z