DATOS GARANTIZADOS CONTRA FALLAS DE ENERGÍA

■ Uribe Cataño, Luis Guillermo
e-mail: luribe@usb.ve
Universidad Católica Andrés Bello, Venezuela

Fecha de Recepción: 10 de Noviembre de 2010
Fecha de Aceptación: 24 de Febrero de 2011

RESUMEN

Mantener íntegra la información al fallar la energía –o las comunicaciones– es un problema común, que empeora cuando los datos están representados por secuencias de múltiples bytes, lo cual es casi siempre el caso. Se precisan soluciones aplicables a equipos muy pequeños (embebidos), que no pueden emplear técnicas avanzadas a la usanza de los grandes RDBMS, pues incluso ciertos cálculos complicados y arduos, como CRCs, usados para establecer la integridad de los datos, o discrepancias entre valores originales y sus copias de respaldo en sistemas de bases de datos, resultan inviables en entornos de microcontroladores. Se presenta aquí un Algoritmo original, en forma de sencilla Máquina de Estados, con la garantía de que, sin importar cuándo re-arranque un sistema, la información jamás resultará inconsistente. Sí podría ocurrir que el último grupo de bytes no llegara a asentarse, lo que resulta apenas inevitable cuando desaparece el indispensable suministro de energía, o al perderse las comunicaciones más allá de lo que resulte tolerable.

En resumen, este Algoritmo asegura que, al grabar un dato multibyte: *Todos* sus componentes quedarán escritos en su totalidad o, de lo contrario: no se grabará *ninguno*.

Palabras clave: Máquinas de Estados; Finite State Machines, FSM. Algorithm Keeps Data Safe; Diagramas de Estados; State Diagrams. Máquinas Algorít-

micas; Algorithmic State Machines, ASM. Sistemas de Aseguramiento de la Coherencia en Información; Data Coherence Systems. Comportamiento Adecuado ante Fallas de Energía; Data Availability and Transactional Integrity under Server Failure. Operación ante Fallas de Comunicación.

Maintaining Data Integrity Against Power Failures

ABSTRACT

Microcomputer systems normally update data to non-volatile memory (flash or EEPROM), or to Database Servers (both local or remote). This paper specially focuses on small, embedded processors, and presents a robust algorithm that prevents data losses and inconsistencies due to power failures, or similar program interruptions. It is centered on multibyte data which normally makes up the information to be processed on small systems. Interrupting the update procedure in the middle of a record may play havoc to the system.

This procedure avoids data losses by maintaining two (2) separate memory areas, duplicating critical variables in each. Storage subsystems can consist on battery-backed RAM, magnetic disks, flash or EEPROM memories, or they can be local or remote storage subsystems. It is based on a very simple, Finite-State Machine that uses 4 States, Gray coded, to track the status of each main variable and its mirrored value in the storage device. The machine dictates a sequence that the software driver can use to write data to both the main and the backup variables. The driver sets and resets two (2) status bits variables, B0: the Main Variable status bit, and B1: the Mirror status bit. Both status bits must be recorded in the same storage medium as the data they represent.

This algorithm prevents race conditions and ambiguous situations. You can enter the power-up verification routine at any time, interrupting the main flow of the program at any place, without losing any data at all. It is guaranteed that all bytes belonging to a data set are written right, or that none are stored, but non half-processed data will be made available to the user under any circumstances.

Keywords: Finite State Machines, FSM. Algorithm Maintain Data Integrity; Power Failures; State Diagrams. Algorithmic State Machines, ASM. Data Coherence Systems. Data Availability and Transactional Integrity under Server Failure. Data Integrity under Power Fail conditions and Communications Channel Interruptions.

I. INTRODUCCIÓN

Mantener la integridad de la información tiene que ser una premisa fundamental en la casi totalidad de los Sistemas, con independencia de su tamaño; esto significa que cuando un dispositivo obtiene y almacena cierto "registro", conformado por diversos valores, no debe haber ninguna posibilidad de que una sección del mismo llegue a no pertenecer apropiadamente al resto. El problema se agudiza, desde el punto de vista de la tecnología electrónica, cuando los datos que se manipulan están constituidos mediante agregados o grupos de bytes, y casi toda información práctica está conformada de esa manera, tanto en equipos pequeños (embebidos) como en servidores de bases de datos para Procesamiento Electrónico de datos (EDP).

En programas conocidos en general como Manejadores de Bases de Datos Relacionales (RDBMS, por sus siglas en inglés), una situación que podría dar origen a que una sección de una entidad de información no se correspondiera con la otra (corrupción de datos) ocurriría si, por ejemplo, dos procesos diferentes trataran de modificar simultáneamente el mismo elemento de la Base de Datos, operación que normalmente se ejecuta en tres pasos: se adquiere primero el registro completo sobre el cual se va a trabajar (*como por ejemplo, la información de nómina de un empleado de una empresa*), a continuación se le aplican las modificaciones pertinentes (*suponga, por vía de ilustración, que se le hacen créditos y descuentos*), y finalmente se devuelve el registro actualizado al servidor para su resguardo definitivo. El resultado de esta secuencia, cuando se están haciendo simultáneamente dos o más modificaciones sobre los mismos datos, dependerá de una condición fortuita: el *orden* en que ellas se ejecuten. Así, si un operador del departamento de contabilidad lee un registro, y a continuación lo obtiene también el operador número 2; después, el primero modifica la información que recibió y la devuelve al servidor (*piense que agregó un crédito por concepto de horas extras a un miembro del personal de la empresa*) y luego el segundo operario aplica sus propios cambios (*como descontarle, por decir algo, un préstamo*) y procede también él mismo a grabar en el disco sus propias modificaciones, el resultado será que el pago adicional que procesó el primer operador desaparecerá por completo del asiento contable, porque esa información quedará totalmente reemplazada por la que almacenó de último el segundo contabilista. Se generará así, en el mejor de los casos, un reclamo airado del empleado de la empresa cuando cobre su salario.

Pueden avizorarse casos mucho peores. Por ejemplo, si ambos contables tratan *al mismo tiempo* de almacenar en el servidor sus propios registros, podría ocurrir que se esté grabando un valor total de Bs. 9,876 como resultado del primer grupo de modificaciones, y uno de Bs. 1,234 correspondiente a los cambios introducidos por el otro contador, y que en definitiva el empleado termine cobrando Bs. 1,2 76 (¡el servidor habría usado las dos primeras cifras de un resultado y las dos últimas del otro!).

Alguien quedará siempre muy insatisfecho cuando ocurren situaciones que arrojan esta clase de resultados incorrectos.

Aun cuando este artículo no trata de RDBMS, puede indicarse que en esos sistemas se aplican estrategias que mantienen la integridad de la información y garantizan que empresas como bancos o líneas aéreas trabajen correctamente, como ocurre en la actualidad. Para eliminar el problema de escrituras múltiples de los registros, los RDBMS incluyen mecanismos que bloquean ("lock") un registro en el momento en que un operador lo adquiere para modificarlo, tal como lo acabamos de ilustrar, y excluye de su acceso a todos los demás operarios interesados en alterar simultáneamente la misma información. Se elimina así la posibilidad de múltiples escrituras, que fue lo que ocasionó el problema señalado con anterioridad. Este tipo de transacciones pretende por lo común que el almacenamiento de Registros de múltiples bytes sea una operación "atómica", que es como se conoce a este mecanismo de bloqueo para escritura.

Estas medidas ordinarias funcionan bien para ambientes de sistemas de Bases de Datos donde cada usuario ejecuta varias tareas simultáneamente y trabajan, además, múltiples usuarios. Ellas consisten en la aplicación de mecanismos clásicos de protección de zonas problemáticas, denominadas Regiones Críticas, mediante Semáforos, "Mutexes" y técnicas similares, que evitan que algún interesado altere campos de datos que al mismo tiempo estén siendo modificados por otros.

Sin embargo ciertos eventos externos, como fallas súbitas de la energía eléctrica, accidentes impredecibles con el botón de re-arranque u ocasionales errores prolongados en las comunicaciones, pueden hacer que una escritura multibyte quede inconclusa, perdiendo irremisible e inevitablemente su atomicidad, y que por tanto, al restablecerse el sistema, una parte de los datos almacenados corresponda a información nueva en tanto que, en otra sección, ésta sea obsoleta…

Si una eventualidad así ocurriera cuando se estuviera, por ejemplo, en el proceso de reservar un lugar en un vuelo de avión, terminaría el pasajero registrado con su

nombre e identificación correctos, pero ocupando el asiento de otra persona, en diferente fecha, ¡o hacia un destino impensado!

Y si un sistema embebido –que es sobre ellos que versa este artículo– está encargado de registrar, por ejemplo, información de la energía eléctrica empleada por ciertas instalaciones industriales, terminaría alojando el valor correcto del consumo, pero asociado a una marca de tiempo anterior, ¡o con el cliente equivocado!

Ciertamente grandes sistemas RDBMS, como DB2, Sybase, Oracle, Postgres y otros (*los nombres comerciales pertenecen a sus respectivos propietarios*), han resuelto este problema para las corporaciones que los usan, aunque en algún Banco de nacional importancia –no identificado aquí… ocurre de tanto en tanto que un movimiento de dinero se da por concluido exitosamente, y se recibe incluso el correo que así lo hace constar ¡al tiempo que los saldos de las cuentas involucradas no reflejan la transacción! O a veces su sistema automatizado de pagos trata de debitar mucho más de lo que precisa ¡para cancelar una tarjeta de crédito que pertenece a la misma institución bancaria! Esto es inconveniente y podría dar lugar a operaciones incorrectas, y hasta fraudulentas. Dichos sistemas convencionales de RDBMS aplican procedimientos y mecanismos que también garantizan la coherencia de la información ante fallas catastróficas de este tipo, y otras como, por ejemplo, la pérdida de los discos del servidor. Ellos incluyen técnicas apropiadas de respaldo, duplicación de datos, discos espejo (RAID) y sistemas de registro secuencial de todas las transacciones ("*Journaling*"; así se emplea también en los sistemas de archivos EXT3 y EXT4 de Linux, en el ZFS de Sun Microsystems y en otros más). El lector interesado en el área de DBMS puede consultar la bibliografía que se incluye al final de este escrito.

Ahora, cuando se trata de microprocesadores embebidos, la realización de cálculos como los del CRC, empleados por ciertos autores como instrumento para determinar la integridad de un grupo de datos, o discrepancias entre la imagen grabada de un Registro y la de su respaldo, resultan del todo improcedentes, por complicados y laboriosos, y por las exigencias que imponen a los escasos recursos computacionales locales.

Se propone aquí una FSM, sencilla pero promisoria, cuyos tres (3) estados centrales representan un algoritmo garante de que, sin importar la parte del ciclo de escritura en que se re-arranque un sistema ¡*la información jamás será inconsistente*! Sí podría suceder que el Registro que se estuviera procesando al aparecer la falla no se almacenara, pero lo contrario sería mucho pedir ante la ausencia del vital soporte energético, o al fallar las comunicaciones. Lo que este algoritmo garantiza es que, si se almacena un dato de múltiples bytes, todos sus componentes quedarán grabados íntegramente, ¡o que jamás se grabará ninguno!

Si se está en medio de una reservación, el cliente quedará expulsado del sistema si se interrumpe la electricidad o fallan las comunicaciones, y tendrá que reingresar más tarde, al restablecerse el servicio… pero lo que nunca ocurrirá es que él se retire con su trámite al día y se encuentre, al abordar el avión, conque su reservación… no está en los términos en que él la hizo.

En el campo de los microcontroladores se ven toda clase de soluciones a este problema, que van desde ignorarlo olímpicamente por completo: "*si se va la energía, alguien tendrá que corregir a mano las inconsistencias*..." (esto es muy común); o: "*instalen un UPS*"… (lo que resulta costoso y no necesariamente resuelve el problema, sino que más bien lo pospone), hasta realizar –como ya se mencionó– cálculos muy imbricados para evaluar el estado de la información en relación a la de su respaldo. Muchas aplicaciones presentadas en flamantes libros de texto de microcontroladores, no toleran, *en lo absoluto*, fallas de energía, ni se hace en ellos ninguna mención ni consideración al respecto…. Esto podría dar como resultado, que se produjeran dispositivos del todo inadecuados en la era de la computación portátil y de los celulares alimentados con baterías.

2. IMPORTANCIA Y ANTECEDENTES

Este problema es muy significativo, pues hay una gran cantidad de sistemas embebidos que necesitan actualizar datos, compuestos de múltiples bytes, a EEPROM, memorias Flash, sistemas del tipo "Hot Standby" y discos magnéticos instalados en servidores de archivos, siendo la corrupción de la información, ante fallas de energía o de las comunicaciones, una opción inaceptable. Contar con un procedimiento como éste, tan simple, eficaz y robusto, fácil de seguir, de entender y de programar, probado y garantizado como algoritmo, que asegure la información y elimine las que de otra manera serían inconsistencias perversas producidas por interrupciones al fallar la energía o las comunicaciones, redundará en un gran beneficio para los programadores de sistemas embebidos, sensibles a las inconsistencias de datos.

En la literatura técnica se discuten varios procedimientos para resolver esta clase de problemas; véase como ejemplo, en "*Embedded Software, Know It All*", Jack Ganssle et. al., Newnes 2008, pp. 377, un método

que involucra el uso de CRCs, costoso en términos de tiempo y de recursos de CPU.

3. METODOLOGÍA PROPUESTA

La estrategia que se seguirá está orientada a evitar pérdidas de información, o a recuperarse de ellas, y consiste en mantener dos (2) áreas separadas, en memoria no volátil, en donde se almacenan –duplicadas– las variables críticas. Los tipos de memoria pueden ser: RAM respaldadas con baterías; discos magnéticos; unidades Flash; subsistemas dobles de almacenamiento independiente, y otros más, por el estilo, y estos dispositivos pueden ser locales o ¡inclusive remotos! (esto último resulta de un gran valor).

La FSM que aquí se presenta usa 3–4 estados, codificados meticulosamente en código Gray, para seguirle la pista a la situación de cada variable principal y a su respaldo, en el dispositivo de almacenamiento no volátil que se haya elegido para cada una de ellas. La secuencia que el código "manejador" debe ejecutar para escribir los datos de ambas variables, principal y de respaldo, transitará los pasos indicados en la Tabla-1: "Tabla de Estados" (refiérase a la tabla en la columna de enfrente). Note que el "manejador" llevará a "**1**" y a "**0**" dos (2) elementos de status: **B0** y **B1**, de acuerdo a la estricta secuencia exigida por la FSM, siendo **B0** el que representa el estado de la variable Principal, y **B1** el bit de la variable de Respaldo, o "espejo". Ambos se graban en el mismo medio de almacenamiento, local o remoto, que corresponda a su variable asociada.

4. COMENTARIOS Y CONCLUSIONES

Este algoritmo *no* es a prueba de otras fallas de hardware. El mecanismo de memoria no volátil, cualquiera que sea, debe funcionar bien: los datos almacenados tienen que corresponder *siempre* a lo que *ya* se grabó, como es normal que ocurra... Errores de hardware deben resolverse mediante procedimientos aparte. Si existiera la posibilidad de que las variables de respaldo difirieran alguna vez, de la información principal, por daños en los discos o en los chips, tendrían que aplicarse políticas apropiadas de recuperación de datos.

Esta solución *requiere* un Estado Inicial garantizado (*como casi cualquier FSM*): La primera vez que se vaya a manipular una variable, sus bits **B0,B1**, correspondientes a su versión principal y de respaldo, *tienen* que estar ambos en cero –pues ese es el estado de arranque de la FSM– y los datos *iniciales* de la variable también deben estar almacenados, tanto en el área principal como en la de respaldo. Si esto no se garantiza, *y llegara a fallar la energía* a la mitad de esa *primera* vez que se escribe una variable multibyte, el valor asumido por la información sería indeterminado… El estado inicial del dispositivo puede pre-configurarse en la fábrica. Si se usan EEPROM o Flash, este procedimiento es rutinario. Si se prefieren los valores conque esas memorias vienen del proveedor (suelen ser **"1"**), puede recodificarse la FSM, en estricto código Gray, comenzando con **"11"** como Estado Inicial. En el caso de discos magnéticos, tendrá que garantizarse que el espacio de almacenamiento para las variables sujetas a esta FSM esté inicializado en ceros, *antes* de comenzar a ejecutarse este algoritmo.

Los bits **B0** y **B1** *tienen* que almacenarse, como se sugiere en los diagramas, en *celdas* separadas, cuyo tamaño dependerá de la mínima cantidad de información que pueda escribirse, de una sola vez, en el dispositivo de memoria no volátil que ha sido seleccionado (*un byte, un sector, un cilindro*…) En particular, al momento de grabar **B0** o **B1**, *no debe reescribirse ningún otro bit* significativo al problema, ni aún de manera *tácita*, como ocurriría en el caso de operaciones del tipo: **Read-Modify-Write** (**RMW**; como, por ejemplo: INC VAR; también son fuente de problemas en esta área, los buffers y cachés). Hay que garantizar que al escribir uno de estos bits **B0** o **B1**, solo haya dos alternativas al momento de una falla de energía: o que el bit mantenga su estado previo, o que lo cambie de acuerdo al valor nuevo que se está grabando. Pero solo debe alterarse un bit a la vez; si se dedican **B0** y **B1** a la misma *celda*, es posible que más de uno altere su valor al fallar la energía, *y esto invalida el algoritmo*. Asimismo se enfatiza que, al grabar **B0** o **B1**, *tampoco* deben reescribirse simultáneamente sus datos asociados, *ni siquiera de forma implícita*. Deben tomarse medidas que garanticen que los datos en realidad *sí* reposan en el dispositivo físico después de una operación de escritura (por ejemplo, abriendo el archivo en modo "*commit*" e invocando los apropiados "*flush*", o trabajando en modo "*unbuffered*").

Si las fallas esperadas son de comunicaciones, deben transportarse los datos mediante protocolos del tipo TCP (*no* UDP), que garanticen la entrega de cada paquete de información y, además, hay que emplear protocolos a nivel de la capa de aplicación que aseguren la apropiada grabación de la información en el medio de almacenamiento remoto: Que cuando se reporte que un dato se recibió y se grabó en el servidor ¡exista seguridad indubitable al respecto!

Si ocurre una falla de luz cuando se está grabando en memoria una información, las *celdas* no involucradas en la operación de escritura *no deben alterarse*; si lo hacen, se considerará esto como un error de hardware, según ya se indicó.

Este algoritmo emplea, además, una estrategia de "*fall-back*" que funciona así: A partir de una posición en la que se conoce, con certeza, que tanto la información Primaria como la de Respaldo almacenada son correctas, lo peor que puede ocurrir al fallar la energía –o las comunicaciones– es que: la máquina queda invariable, en la misma posición, o en otra en la cual también se conoce que una de las copias de la información almacenada es correcta; esto posibilita restablecer adecuadamente los duplicados. El proceso empieza grabando una variable, pero los bits **B0** y **B1** indican que esto no ha ocurrido. Después de almacenada y asegurada la información, si se comienza a cambiar el estado **B0,B1** y falla la luz: O se sigue indicando que *no* se hizo la grabación, y los datos volverán a re-escribirse en la próxima verificación (*pup*, cfr. Figs. 1 y 2), tomados de la copia apropiada, o se indica que *sí* se grabó, lo cual es correcto. En el primer caso hay que reescribir ¡lo que ya estaba bien!… esto parece un precio muy bajo por *asegurar* la calidad de la información multibyte.

El algoritmo aquí planteado requiere únicamente el soporte normal de hardware que cualquier otro programa; no necesita circuitería especial para su implementación, siempre que se respeten todas las consideraciones que acabamos de hacer, en relación a los medios de almacenamiento. Esta FSM debe implementarse mediante un programa para el micro, que puede estar en Assembler en dispositivos pequeños, o en lenguajes de alto nivel, como el "C", si se cuenta con infraestructura para ellos. Las rutinas que materializan la máquina de estados no tienen que ser atómicas; basta con garantizar que se termina cada una, de manera completa y satisfactoria, *antes* de pasar a la siguiente rutina ("*commit*"). No importa el momento en que se interrumpa el proceso, la máquina siempre se recuperará, mientras se respeten todas las consideraciones aquí planteadas.

Este procedimiento soporta fallas simultáneas, de energía y/o de comunicaciones; es decir, si se estuviera en el proceso de recuperación de una eventualidad cualquiera y, sin que éste haya finalizado, ocurriera otra, o similar, perturbación, el sistema mantendrá la consistencia de la información… ¡a ultranza!

```
B1 B0 --- SIGNIFICADO DEL ESTADO ---
----- ------------------------------------------------
 0 0  VARIABLES PRINCIPAL Y DE RESPALDO, AMBAS SEGURAS
      ---
      Si la rutina de "power up" (pup) encuentra este
      estado, puede asumir que todo está correcto;
      ambos valores, el Principal y el de Respaldo, se
      han almacenado perfectamente, tanto en la
      memoria principal como en la de respaldo.
      Pueden leerse y usarse los datos de la variable
      Principal con entera libertad.
------------------------
Cuando llegue la ocasión de modificar un valor, el
manejador lleva B0 a "1" para pasar a "01" como sigue:
=====================================================
 0 1  COMENZARÁ A GRABARSE SOBRE LA VARIABLE PRINCIPAL
      ---
      Si la rutina "pup" entra en este estado al hacer
      su verificación, puede asumirse que está dañada
      la variable Principal, y la acción que hay que
      ejecutar es: Actualizarla, desde los datos que
      están a buen resguardo en la memoria de Respaldo
------------------------
Cuando el manejador termina de escribir en la variable
Principal lleva B1 a "1" y procede a "11", como sigue:
=====================================================
 1 1  LA VARIABLE PRINCIPAL SE HA GRABADO BIEN; PRONTO
      COMENZARÁN (siguiente estado) A REPLICARSE SUS
      COMPONENTES HACIA LA VARIABLE DE RESPALDO.
      Este es solo un estado transitorio, que se usa
      para evitar cambiar entre las asignaciones "01"
      y "10" de una sola vez, lo que podría terminar
      en un valor indeterminado si fallara la energía.
      ---
      Si pup se tropieza con este estado debe asumirse
      que es errónea la variable de Respaldo y se irá
      (siguiente estado) a Actualizarla, tomando los
      datos perfectamente guardados en la Principal.
------------------------
Cuando termina de replicarse el área Principal sobre
las posiciones de Respaldo, se lleva B0 a "0" y se
continúa a "10", así:
=====================================================
 1 0  LA VARIABLE PRINCIPAL SE HA GRABADO BIEN Y SE
      REPLICARÁN AHORA SUS COMPONENTES HACIA EL ÁREA DE
      RESPALDO (similar al estado Temporal, previo)
      ---
      Si pup ve este estado, asumirá que es errónea la
      variable de Respaldo, y la Actualizará desde los
      datos, correctos, que se encuentran grabados en
      la memoria Principal.
------------------------
Cuando el manejador termina de grabar la variable de
Respaldo, lleva B1 a "0" lo que conduce a la FSM al
Estado Inicial de su secuencia, "00".
=====================================================

No hay carreras o situaciones ambiguas en el algoritmo,
y la rutina de "power up" puede ejecutarse en cualquier
instante, interrumpiendo el flujo principal del programa,
como puede verse en los Diagramas de Estado que se anexan
a continuación, sin perder, jamás, ningún dato.
```

TABLA-1: "TABLA DE ESTADOS"

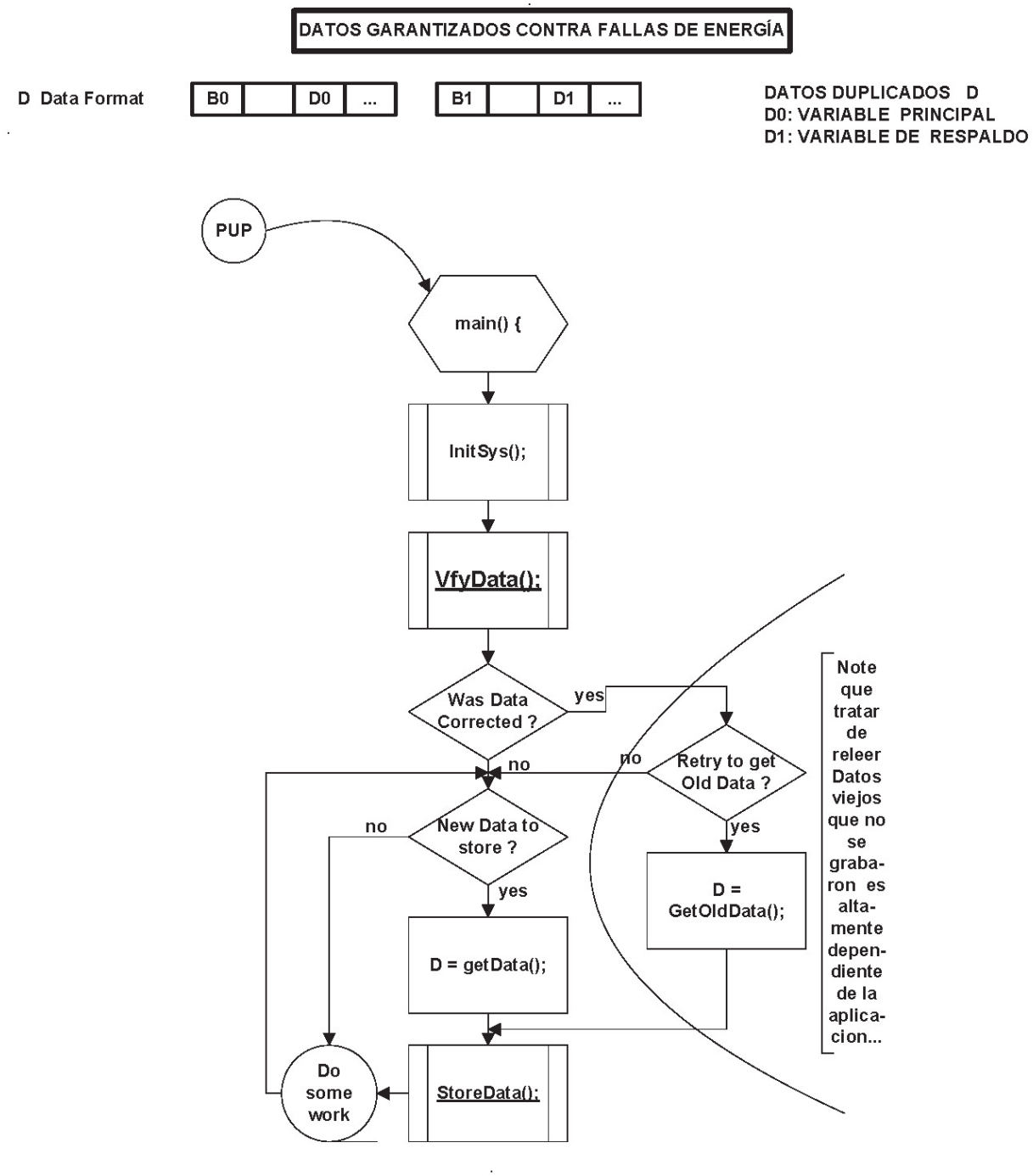

FIGURA-1: DIAGRAMA DE ESTADOS PRINCIPAL

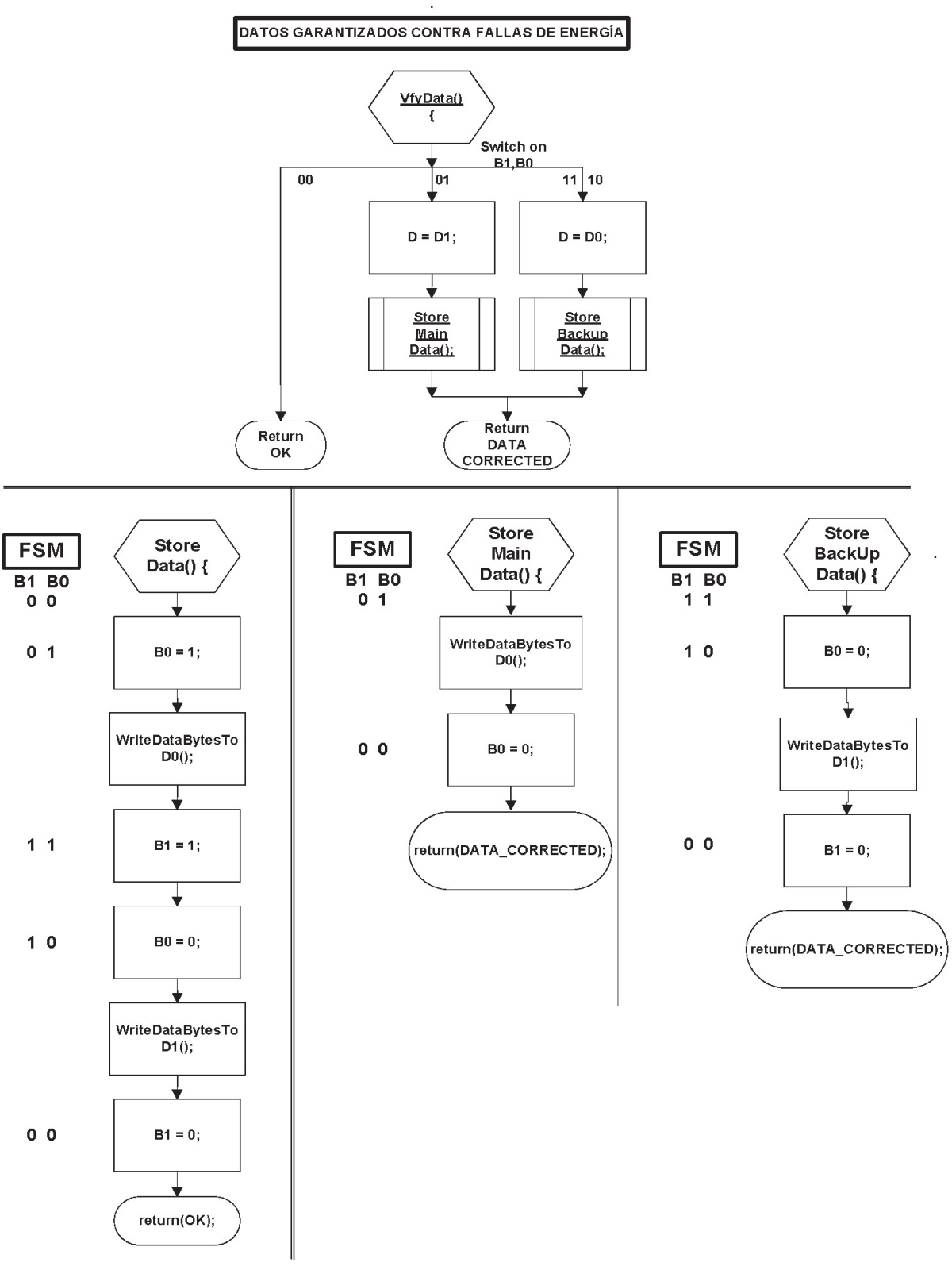

FIGURA-2: DIAGRAMA POR FUNCIONES

5. REFERENCIAS

[1] Ganssle, Jack, et. al., "Embedded Software, Know It All", ISBN: 978-0-7506-8583-2, Newnes 2008, pp. 374 y ss.: "*Multibyte Writes*".

[2] Weikum, Gerhard, "Transactional Information Systems, Theory, Algorithms, and the Practice of Concurrency Control and Recovery", pp. 427 y ss.: "*Crash Recovery: Notion of Correctness*".

[3] Liu, Ling, et. al. (Eds.), "Encyclopedia of Database System", ISBN 978-0-387-49616-0, Springer 2009.

[4] Gustafsson, Thomas, "Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems", Doctoral thesis, ISBN 978-91-85831-33-3, Institutionen för datavetenskap, Linköpings universitet, Sweden, 2007.

[5] Chong, Raul, et. al., "Getting Started WITH DB2 Express-C", 3rd. Ed., IBM Corp., 2010, pp. 199 y ss.: "*Database Crash or restart recovery*".

[6] Aue, Axel (Korntal, DE), "Method for Recognizing a Power Failure in a Data Memory and Recovering the Data Memory", United States Patent 20090158089, 2009, pp. 3 y ss.: "*… CRC is time-intensive…*".

[7] CSA (Canadian Standards Association) N290.14-07, "Qualification of pre-developed software for use in safety-related instrumentation and control applications in nuclear power plants", 2007, pp. 23: "*…product retains state data …during any power failure/restart event*".

[8] Ferraggine, Viviana, et. al., "*Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*", ISBN 978-91-85831-33-3, Publ.: Information Science Reference, 2009.

[9] Atkinson, Colin, et. al., "*Component-Based Software Development for Embedded Systems*", Springer 2005.