Ensamblador Orientado a Objetos

Raúl Martín Rello

El propósito del siguiente texto es el de presentar una nueva metodología de programación, por medio de la cual se consigue diseñar un programa orientado a objetos en lenguaje ensamblador. Basando su funcionamiento en una jerarquía de capas, o niveles de abstracción, el software se engloba en clases (objetos), de forma que el código adquiere una serie de características difícilmente alcanzables con la programación tradicional. Tales como: fácil mantenimiento, alta reutilización del código, rápido desarrollo de nuevas aplicaciones, etc.

Hoy en día, la programación orientada a objetos (POO) es la metodología de programación más utilizada por los programadores de todo el mundo. El desarrollo de aplicaciones en lenguajes de alto nivel en los que el mantenimiento y reutilización del código era un difícil objetivo de conseguir con la programación tradicional, ha sido reemplazado sistemáticamente por aplicaciones orientadas a objetos, en la que estas trabas quedan subsanadas por las características inherentes a esta teoría de programación.

Sin embargo, así como la transición a la orientación a objetos ha sido llevada a cabo con gran éxito en la programación de aplicaciones en lenguajes de alto nivel, en aplicaciones programadas en lenguaje ensamblador (lenguaje de bajo nivel) el paso de la programación tradicional a la POO no ha tenido tanta aceptación. Uno de los principales problemas es que el lenguaje ensamblador es utilizado mayormente para incrementar la velocidad de ejecución del código, y la ejecución de un software basado en la POO trae como consecuencia directa, una ralentización del mismo. Este problema se acrecenta cuando hablamos de software diseñado para aplicaciones embebidas, en la que los recursos del procesador son limitados (cantidad de memoria ROM, RAM, velocidad del ciclo de bus, etc...).

En el presente texto, se sientan las bases de diseño de una nueva metodología de programación, por medio de la cual es posible diseñar un software orientado a objetos en lenguaje ensamblador. Dicha metodología alcanza su máximo rendimiento en aplicaciones embebidas, ya que además de mantener la velocidad de ejecución del código, añade una serie de características adicionales tales como:

- Fácil mantenimiento del código
- Alta reutilización del código
- Gran robustez frente a errores
- Fácil reubicación del código
- Rápido desarrollo de nuevas aplicaciones

La metodología de orientación a objetos en ensamblador (**EOO**) que se presenta en este texto, se basa en una estructura de capas jerárquicas, en la que cada capa puede contener una o más clases.

Dependiendo del tipo de clases que contiene una capa dada, se dirá que ésta pertenece a un nivel de abstracción o a otro. Las clases, a su vez, son objetos que contienen propiedades y métodos (al igual que ocurre en la programación a alto nivel). Puesto que una clase puede ser una propiedad de otra clase superior, la principal característica de la POO, la encapsulación, se mantiene.

Por el contrario, el resto de características tales como herencia, polimorfismo,etc. así como otras de los lenguajes de alto nivel como la asignación dinámica de memoria, presentan serios problemas relacionados con la velocidad de ejecución del código; por ello han sido eliminados de esta teoría.

En la siguiente sección se explica de forma detallada como crear una clase en lenguaje ensamblador. Se explicará como se accede a sus elementos miembro: propiedades y métodos. Más adelante se explica el modelo de capas jerarquizadas de la teoría EOO. Se profundizará en detalle sobre lo que se conoce como capa hardware, capa de librerías y capa de aplicación. Posteriormente se desarrolla un sencillo ejemplo para que la explicación teórica tenga su correspondiente aplicación práctica.

Creando Clases en Ensamblador

Las clases y sus miembros

En primer lugar, hay que tener claro que cuando hablamos de una clase nos referimos a la estructura de datos por excelencia de la POO. Una clase es como una caja cerrada en la que está contenido todo lo necesario para que funcione por sí misma. Las clases están constituidas por sus elementos miembro: las propiedades y los métodos.

Las propiedades son los datos que determinan las características de una clase. Por otra parte, los métodos son los mecanismos para modificar el estado de las propiedades, es decir, para cambiar ciertas características de la propia clase.

Para poder crear objetos de un tipo de clase concreto, es necesario recurrir a un mecanismo llamado instanciación. En l os lenguajes de alto nivel, tales como C++ o Java, la instanciación se reduce a llamar al método *New*. Sin embargo, cuando se programa en ensamblador, esta simple llamada se convierte en un proceso mucho más complejo. Y sobre todo, cuando hablamos de sistemas embebidos donde la memoria RAM está limitada.

Además, la instanciación en alto nivel no se preocupa de la cantidad de memoria RAM que se reserva ni dónde se localiza en

clsSolar.asm

```
;definición de propiedades como si fueran constantes
prpTierra  EQU   1
prpLuna    EQU   2
```

Ejemplo.asm

```
;instancia de la clase Solar, llamada MiClase y situada en la
;posición de memoria 0EBh.
MiClase    EQU   0x0EB

           MOV   X,MiClase      ;X -> 0EBh
           ADD   X,prpTierra    ;X = X+1 = 0EB+1 = 0ECh
           MOV   [X],5          ;[X] = [0EC] = 5

           MOV   X,MiClase      ;X -> 0EBh
           ADD   X,prpLuna      ;X = X+2 = 0EC+1=0EDh
           MOV   [X],4          ;[X] = [0ED] = 4
```

Cuadro 1

concreto una instancia dada, sino que es el propio sistema operativo el que se encarga de gestionar dicha zona de memoria, tanto para ocuparla como para liberarla en el momento preciso. Esto en ensamblador es inabordable por dos razones: la velocidad de ejecución debe ser alta y el espacio de memoria es limitado.

Al programar en lenguaje ensamblador, se trata de que el código sea lo más rápido posible, y que el programador tenga el control absoluto del procesador sobre el que programa.

Es decir, que en todo momento el programador debería saber en que lugar de memoria reside cada clase, ya que de esta forma es muy sencillo corregir posibles errores y realizar un buen mantenimiento del código.

La asignación dinámica de memoria, que se utiliza en los lenguajes de alto nivel, está reñida con las dos razones anteriores, puesto que al gestionar la memoria se pierde velocidad de ejecución y además nunca se conoce a priori donde estará localizada una clase en concreto.

La única manera de huir de la asignación dinámica de memoria, es utilizar un mecanismo de instanciación estática o fija. La manera más sencilla de conseguir esto es reservar el espacio de memoria necesario en la etapa de diseño. O lo que es lo mismo, será el propio programador el que haga de gestor de memoria a la hora de diseñar el software. De esta forma, se consigue evitar la pérdida de tiempo en buscar una zona libre de memoria durante la ejecución del programa y a priori, el programador conocerá exactamente la zona de memoria en la que se ha realizado la instancia.

Definición de propiedades

A la hora de definir las propiedades de una clase, es necesario tener en cuenta, que se haga donde se haga la instancia en memoria, cada una de las propiedades internas debe ser asociada a una posición de memoria RAM concreta. Sin embargo, como una instancia puede ser realizada en cualquier parte de la memoria RAM, y su funcionamiento debe ser totalmente independiente de donde se localice dicha zona de memoria, las propiedades no deben estar asociadas a posiciones fijas, sino que deben situarse en posiciones únicamente relativas a la cabecera de la clase.

El que una propiedad no esté localizada en una posición fija de memoria trae como primera consecuencia importante, que no debe ser definida como una variable, sino como una constante. El valor de la constante asociada a dicha propiedad será igual al desplazamiento a realizar sobre la cabecera de la clase.

De esta manera, es posible definir cada una de las propiedades de una clase sin tener que asignarles una posición de memoria específica. Únicamente cuando se lleve a cabo la instanciación de la clase en concreto, se hará una reserva de memoria para cada una de las propiedades definidas.

La siguiente porción de código define una clase llamada *clsSolar*, que contiene dos propiedades *prpTierra* y *prpLuna*. La clase no está asociada a ninguna posición concreta de memoria RAM, sin embargo la propiedad "*prpTierra*" se ha definido como una constante de valor 1 y la propiedad *prpLuna* como una constante de valor 2. Esto quiere decir que a la hora de realizar una instancia de la clase *clsSolar*, la propiedad *prpTierra* ocupará la siguiente posición de memoria y la propiedad *prpLuna* la posición situada a continuación de la propiedad *prpTierra*.

En el ejemplo siguiente, se muestra el contenido del archivo cls*Solar.asm* que define la clase *clsSolar* y sus propiedades; y el archivo *Ejemplo.asm* en el que se define una instancia de la clase *clsSolar*, que se llama *MiClase* y se sitúa en la posición 0EBH de memoria RAM.

En el cuadro 1 se muestra una porción de código que hace que la propiedad *prpTierra* (RAM ECh) valga 5 y la propiedad *prpLuna* (RAM 0EDh) valga 4.

Como puede verse, definiendo las propiedades de una clase de esta forma, es posible acceder a ellas sin necesidad deasignarles una posición de memoria fija. De esta forma, el código anterior funcionaría igual si la instancia hubiese sido hecha en la posición 100H en vez de haberlo hecho en la posición 0EBh.

Podemos concluir por tanto, que todas las propiedades de una clase se definirán como constantes, cuyo valor indica el desplazamiento en memoria respecto de la cabecera de la clase.

Métodos. La cabecera de clase.

Todos los métodos de una clase tienen en común la característica de que son capaces de acceder a cualquier propiedad de la misma. Para que un método sea capaz de acceder a una propiedad cualquiera, es necesario que conozca la posición de memoria que ocupa. Como se ha visto en el punto 2.2,

las propiedades son declaradas como constantes y por lo tanto es imposible conocer a priori donde serán colocadas en memoria. Puesto que las propiedades se definen como un desplazamiento relativo a la cabecera de la clase, es necesario conocer cual es dicha cabecera. Lógicamente, la cabecera de la clase deberá ser una propiedad definida como una constante de valor 0, ya que tras la instanciación, dicha propiedad estará colocada justo en la misma posición de memoria que la propia instancia.

A la propiedad que define la cabecera de la clase se le llama propiedad *prpThis*, y siempre debe definirse como una constante de valor 0. Además la propiedad *prpThis* siempre deberá existir obligatoriamente.

Una vez que tenemos el mecanismo para referenciar todas las propiedades de una clase (respecto de la propiedad *prpThis*), es necesario implementar un mecanismo capaz de recuperar la cabecera de la clase en cualquier parte del cuerpo del método.

La forma más sencilla de hacer esto es utilizar la pila para guardar la posción de memoria en la que se sitúa la cabecera de la clase. De esta manera, recuperando de la pila la cabecera de la clase, seremos capaces de acceder a cualquier propiedad simplemente realizando un desplazamiento sobre ella; como se vió en el ejemplo anterior. Puesto que la posición de memoria de la instancia no se conoce, y debe ser recuperada de la pila a lo largo del cuerpo del método, es lógico pensar que primero habrá que almacenarlo en la pila. La forma más sencilla de hacerlo es utilizar la instancia como primer argumento del método.

En la porción de código ubicada a la izquierda de estas líneas, perteneciente al archivo *clsSolar.asm*, se define un método público llamado *New*, por medio del cual, se crea un objeto de tipo *clsSolar* con sus propiedades iniciadas a 0 por defecto. En el archivo *Ejemplo.asm* se declara una instancia de la clase *clsSolar* llamada *MiClase* en la posición 100H y posteriormente se crea la instancia con el método *New* de la clase.

Utilizando como paso de argumentos la instancia de la clase, los métodos de la misma tienen la condición necesaria para poder acceder a las propiedades de la misma.

Estructura de Jerárquía de Capas

La jerarquía de capas confiere a esta tería de programación la gran ventaja de que el código resultante es mucho más modular, independiente de la aplicación y fácil de mantener, lo cual hace que las aplicaciones sean diseñadas más rápidamente y sean más robustas frente a errores.

En la metodología EOO se distinguen tres capas fundamentales:

- Capa hardware
- Capa de librerías
- Capa de aplicación

La *Capa Hardware* es la más profunda de la jerarquía, en ella se engloban únicamente las clases que son capaces de acceder de forma directa al hardware de la aplicación. Es por lo tanto donde se realizan todas las instancias (asignación del mapa de memoria) y desde donde arranca toda aplicación.

La *Capa de Librerías* es una capa intermedia y contiene todas las clases independientes de la aplicación que pueden ser necesarias a lo largo del software. Pueden existir clases específicas o clases que sólo contengan métodos de propósito general.

La *Capa de Aplicación* es la capa más superficial de la jerarquía. En ella se encuentra la clase que hace funcionar a la aplicación en cuestión, y que se vale de las clases almacenadas en la Capa de Librerías (clases independientes) para dar vida al software definitivo.

Visto de esta manera, a la hora de diseñar un nuevo proyecto de software basado en un mismo hardware, tan sólo habría que rediseñar la Capa de Aplicación, puesto que el resto de capas serían perfectamente válidas. Y no sólo eso, sino que además, por utilizar una metodología orientada a objetos, las clases que hayan sido verificadas y estén libres de errores seguirán así. Por lo tanto, a la hora de depurar el software sólo será necesario depurar la Capa de Aplicación, puesto que las otras dos capas ya funcionan correctamente.

Un Ejemplo

En esta sección, se muestra un ejemplo de aplicación de la teoría EOO. El ejemplo consiste en desarrollar un software que simule el comportamiento de un reloj en tiempo rea (ver página a la derecha de este texto)l. Para ello, el software hará uso

clsSolar.asm

```
;definicion de propiedades como si fueran constantes
prpThis   EQU   0
prpTierra EQU   1
prpLuna   EQU   2

;método público (global) New
.GLOBAL clsSolar_NEW
clsSolar_NEW:
        ADD     SP,1            ;con estas tres
                                ;instrucciones, recupera
        POP     X               ;el argumento 1 pasado al
        SUB     SP,2            ;método y equilibra la
                                ;pila
                                ;ahora X -> prpThis
        ADD     X,prpTierra     ;X = X+1 -> prpTierra
        MOV     [X],0           ;[X] = [prpTierra] = 0

        ADD     SP,1
        POP     X
        SUB     SP,2
                                ;X -> prpThis
        ADD     X,prpLuna       ;X = X+2 -> prpLuna
        MOV     [X],0           ;[X] = [prpLuna] = 0

        RET
        .end
```

Ejemplo.asm

```
;instancia de la clase clsSolar situada en RAM 100h.
MiClase EQU     0x100

        PUSH    MiClase         ;PUSH 100h
        CALL    clsSolar_NEW    ;[101h] = 0
                                ;[102h] = 0
        ADD     SP,1            ;equilibra la pila.
```

Hw.asm

```
;código fuente de la capa Hardware: Hw.asm
#include "clsMicro.asm"
.end
```

Lib.asm

```
;código fuente de la capa de Librerías: Lib.asm
#include "clsRlj.asm"
#include "clsMetodos.asm"
.end
```

App.asm

```
;código fuente de la capa de Aplicación: App.asm
#include "MiReloj.asm"
.end
```

clsMicro.asm

```
;código fuente del archivo clsMicro.asm
;definición de la clase clsMicro. Mapa de memoria
;instancia de la clase Micro situada en la posición RAM 0h
MiAplicacion    EQU     0x0000      ;instancia de un objeto
                                    ;del tipo aplicación.
                                    ;Reserva 4 posiciones de
                                    ;memoria RAM.
flag_int_clk    EQU     0x0004      ;flag de control para la
                                    ;interrupción de 1Hz.

;código del vector de interrupción de INT_CLK
        ORG     INT_CLK
        MOV     flag_int_clk,1
        RET

;código de inicio del programa principal
        ORG     APP_INI

        PUSH    flag_int_clk        ;argumento 2
        PUSH    MiAplicacion        ;argumento 1, es un
                                    ;objeto de tipo clsMiReloj
        CALL    app_execute         ;llama al método público
                                    ;de la clase clsMiReloj
        ADD     SP,2                ;equilibra la pila
        .end
```

clsRlj.asm

```
;código fuente de la clase clsRlj
;definición de las propiedades
prpThis EQU     0                   ;cabecera de la clase
prpSeg  EQU     1                   ;primera propiedad
prpMin  EQU     2                   ;segunda propiedad
prpHor  EQU     3                   ;tercera propiedad


;métodos de la clase clsRlj
;método privado IncSeg
rlj_IncSeg:
        ADD     SP,1
        POP     X
        SUB     SP,2

        ADD     X,prpSeg            ;X->clsRlj.prpSeg
        CALL    incXde0a59          ;método de propósito
                                    ;general

        RET


;método privado IncMin
rlj_IncMin:
        ADD     SP,1
        POP     X
        SUB     SP,2

        ADD     X,prpMin            ;X->clsRlj.prpMin
        CALL    incXde0a59          ;método de propósito
                                    ;general

        RET


;método privado IncHor
rlj_IncHor:
        ADD     SP,1
        POP     X
        SUB     SP,2

        ADD     X,prpHor            ;X->clsRlj.prpHor
        CALL    incXde0a23          ;método de propósito
                                    ;general

        RET
```

```
;método público TicTac
.GLOBAL rlj_TicTac
rlj_TicTac:
        ADD     SP,1
        POP     X
        SUB     SP,2

        PUSH    X
        CALL    rlj_IncSeg          ;incrementa los segundos
        JUMP    Z,label_ret         ;si segundos!=0, salta.
        CALL    rlj_IncMin          ;sino, incrementa mins.
        JUMP    Z,label_ret         ;si mins != 0, salta
        CALL    rlj_IncHor          ;sino, incrementa horas
label_ret:
        ADD     SP,1                ;equilibra la pila
        RET
        .end
```

clsMetodos.asm

```
;código fuente de la clase de métodos de propósito general
;clsMetodos
;definicion de propiedades.
prpThis EQU     0                   ;cabecera de la clase

;métodos de la clase clsMetodos
;método público incXde0a59
.GLOBAL incXde0a59
incXde0a59:
        ADD     SP,1
        POP     X
        SUB     SP,2

        ADD     [X],1               ;suma 1 al contenido de X
        CMP     [X],60              ;compara con 60 y si es
                                    ;mayor o igual, salta
        JUMP    NC,/set00_
        SET     Z                   ;si es menor, activa Z.
        RET
set00_:
        MOV     [X],0               ;si es menor de 60,
        CLR     Z                   ;borra el flag Z
        RET

;método público incXde0a23
.GLOBAL incXde0a23
incXde0a23:
        ADD     SP,1
        POP     X
        SUB     SP,2

        ADD     [X],1               ;suma 1 al contenido de X
        CMP     [X],24              ;compara con 24 y si es
                                    ;mayor o igual, salta
        JUMP    NC,set00_
        SET     Z                   ;si es menor, activa Z.
        RET
        .end
```

MiReloj.asm

```
;código fuente de la clase MiReloj.asm
;definición de las propiedades
prpThis     EQU     0               ;cabecera de la clase
prpMiReloj  EQU     1               ;instancia tipo clsReloj
                                    ;reserva 3 posiciones
prpCtrl     EQU     4               ;propiedad de control

;métodos de la clase clsMiReloj
;método público app_execute
.GLOBAL app_execute
app_execute:
halt_:  HALT                        ;espera una interrupción
        ADD     SP,1
        POP     Y                   ;recupera arg2
        POP     X                   ;recupera arg1
        SUB     SP,3                ;y equilibra la pila.

        CMP     [Y],1               ;si no es INT_CLK
        JUMP    NZ,halt_            ;vuelve a esperar otra.

        ADD     X,prpMiRlj          ;X->clsApp.prpMiRlj
        PUSH    X                   ;X->clsApp.MiRlj
                                    ;(guarda en pila)
        CALL    rlj_TicTac          ;incrementa la hora
        JUMP    halt_               ;espera a la siguiente
                                    ;interrupción
        .end
```

Ejemplo de aplicación

de una interrupción periódica que se produce en el procesador; dicha interrupción salta cada segundo y se atenderá por medio de un vector de interrupción llamado INT_CLK. De esta manera, nuestro software utilizará esta interrupción para ir incremetando los segundos, los minutos y las horas.

El ejemplo cuenta con una serie de archivos que siguen el patrón de estructura de capas jerárquicas. Los archivos son los siguientes:

Hw.asm, incluye la información de las clases que forman parte de esta capa. Está formada por aquellas clases que tienen acceso directo al hardware.

Lib.asm, incluye la información de las clases que forman parte de la capa de librerías, y que por lo tanto son independientes de la aplicación.

App.asm, incluye la información de las clases que están incluidas en la capa de aplicación.

clsMicro.asm, incluye el código fuente asociado a la clase *clsMicro*, que gestiona el control del microprocesador utilizado. Puesto que es en este archivo donde se define el mapa de memoria y se encuentran todos los métodos que dan acceso al hardware, debe formar parte de la capa más profunda, es decir la del hardware

clsRlj.asm, incluye el código fuente de una clase de tipo *clsRlj*. Puesto que es una clase completamente independiente de la aplicación, debe estar incluida en la capa de librerías.

clsMetodos.asm, incluye el código fuente de una clase que únicamente contiene métodos de propósito general, y que se ha llamado *clsMetodos*. Por ser independiente de la aplicación formará parte de la capa de librerías.

clsMiReloj.asm, incluye el código fuente de una clase correspondiente a la capa de aplicación. En este caso, *clsMiReloj*, hace uso de las capas incluidas en los niveles inferiores de la jerarquía de capas para crear una aplicación completa.

Conclusiones

Como se ha podido ver en el ejemplo expuesto, hay cuatro clases perfectamente definidas, y agrupadas en tres archivos que se corresponden con cada una de las capas de la jerarquía. Se ha conseguido crear un código en ensamblador con una metodología orientada a objetos que presenta las siguientes características:

- Alta velocidad de ejecución
- Estructura modular
- Fácil mantenimiento
- Gran robustez frente a errores
- Altamente reutilizable
- Código 100% reubicable. ❑

Referencias

[1] Introducción a los microcontroladores. - J.A. González Vazquez. ISBN: 8476158033 (McGraw Hill).
[2] Teoría y diseños con microcontroladores PIC. – Antonio R. Tafanera. ISBN: 9874318686 (Autores Editores).
[3] Ensamblador básico. - A. Rojas. ISBN: 9701500989 (Alfaomega Grupo Editor).
[4] Short Course 8051/8032 Microcontroller Assembler. – M. Oshmann. ISBN: 0905705386 (Elektor electr.)
[5] http://www.embedded.com
[6] http://www.computer.org/micro/index.htm