Universidad Europea

LAUREATE INTERNATIONAL UNIVERSITIES

La pila y funciones

Contenido

Presentación

La **programación estructurada**, base de cualquier otra metodología de programación, también está soportada en ensamblador, aunque de una manera muy simple pero robusta.

La programación estructurada está fundamentada en las **funciones, trozos de código independiente, autónomo y reutilizable**. Es decir, trazos de código que resuelven un problema y que son independientes del resto de código del programa, lo que lo hace muy portable y reutilizable. La única comunicación que mantiene con el resto del programa es a través de parámetros, por los que se le pasa la información de entrada y por la que sale la información calculada.

Las funciones, además, fundamentan su funcionamiento en la pila. Un concepto de almacenamiento de información, cuyo principio de funcionamiento es "último en entrar, primero en salir (LIFO)". Es decir, podemos entender una pila, como un **almacenamiento** donde los elementos están **apilados verticalmente**, de manera que solo la cima de la pila es accesible, siendo solo el elemento último en poner en la pila, el único que puede ser sacado. Los demás elementos (aunque visibles) no pueden ser sacados de la pila, mientras encima de ellos haya otros elementos.

Objetivos

Los **objetivos** que se pretenden alcanzar con este recurso son los siguientes:

- Analizar el **funcionamiento** de la pila software que incluyen todos los procesos.
- Reproducir el **manejo** de la pila software en ensamblador.

Funciones (o rutinas)

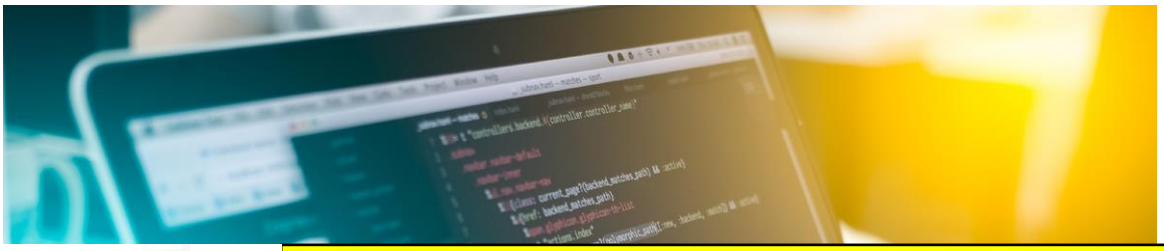
Cuanto mayor es una organización empresarial, más crecen las responsabilidades y el volumen de ésas. Por tanto, la evolución natural de las organizaciones les lleva a ir repartiendo las responsabilidades sobre las actividades que se van desarrollando.

Las rutinas y funciones (que se diferencian porque las primeras no tienen parámetros y las segundas sí) son **conceptos básicos** en todos los lenguajes de programación y base de la programación estructurada.

Permiten la **reutilización de código**, la separación/división de problemas y la reutilización de una manera muy simple. Las funciones se pueden agrupar, formado bibliotecas, de modo que puedan volver a ser utilizadas en el futuro, sin necesidad de reescribir el código que las implementa de forma directa en nuestro propio proyecto.

Además, las funciones (o rutinas) son la base de la **división de trabajo** en equipos de desarrollo de varios programadores. Permitiendo una división del trabajo y una fácil unificación posterior. Dando lugar, al primer elemento básico necesario para un primer nivel de abstracción en el diseño de software (*Software Engineering*)

Sea cual sea el lenguaje de programación que se use, o el nivel de abstracción que permita, por ejemplo, en programación orientado a objetos, al final, todo el código es traducido a estas pequeñas instrucciones de código máquina y a sus simples, pero efectivas funciones. No importan el tamaño del objeto, la forma de los parámetros de una función u otras abstracciones, que al final, serán la pila y las llamadas a las funciones las encargadas de dar soporte a todo este código de alto nivel.



Declaración de una función

Una función, es sólo un grupo de instrucciones que **realizan una tarea determinada y fija**, y luego de terminar su labor, la ejecución del programa vuelve al punto de llamada.

Por tanto, el uso típico que se le da a una función, es el de solucionar un **problema aislado**, o el de encapsular un trozo de código que se va a repetir muchas veces a lo largo de nuestro programa.

Ejemplo de repetición

Crear una rutina (no tiene parámetros de entrada ni salida) que permita:

- Borrar todos los bits del puerto B.
- Inicializar el temporizador 0.
- Activar la interrupción de tiempo.

Si esta fuese una tarea que se realiza varias veces en el programa, es buena idea **encapsularlo en una rutina**, de manera que cada vez que se quiera hacer esta actividad, solo haya que invocar (llamar) la función, no teniendo que reescribir el código de nuevo.

La declaración de funciones en ensamblador, **no existe**, es simplemente una dirección de memoria de programa, que coincide con a la primera instrucción que componen el grupo de instrucciones. Para simplificar este cálculo de la primera instrucción, generalmente se identifica con una etiqueta para que el cálculo y la memorización de la funciona sea más simple. Esta etiqueta representa el nombre de la rutina.

Llamada a una función

La llamada a una función, desde el punto de vista del microprocesador, es solo un salto (incondicional) a un punto del código. Por tanto, se podría implementar con instrucciones tipo JMP.

Pero entonces, ¿cómo se podría gestionar el regreso al punto de llamada? Hay que tener en cuenta que una misma función puede ser llamada desde varios puntos de código, por lo que saber desde el punto que ha sido llamada, para luego continuar desde ahí, se convierte en un problema.

Una posible solución, es almacenar el punto de llamada de alguna manera (se recuerda que el punto desde el que se llama una función es el valor del registro PC antes del salto), por ejemplo en un **registro**, luego, consultando ese punto valor almacenado, se puede saber cuál es el punto de retorno. El problema, es que una función puede ser invocada desde otra función, con lo que ya hay que recordar dos puntos de retorno (¿dos registros?).

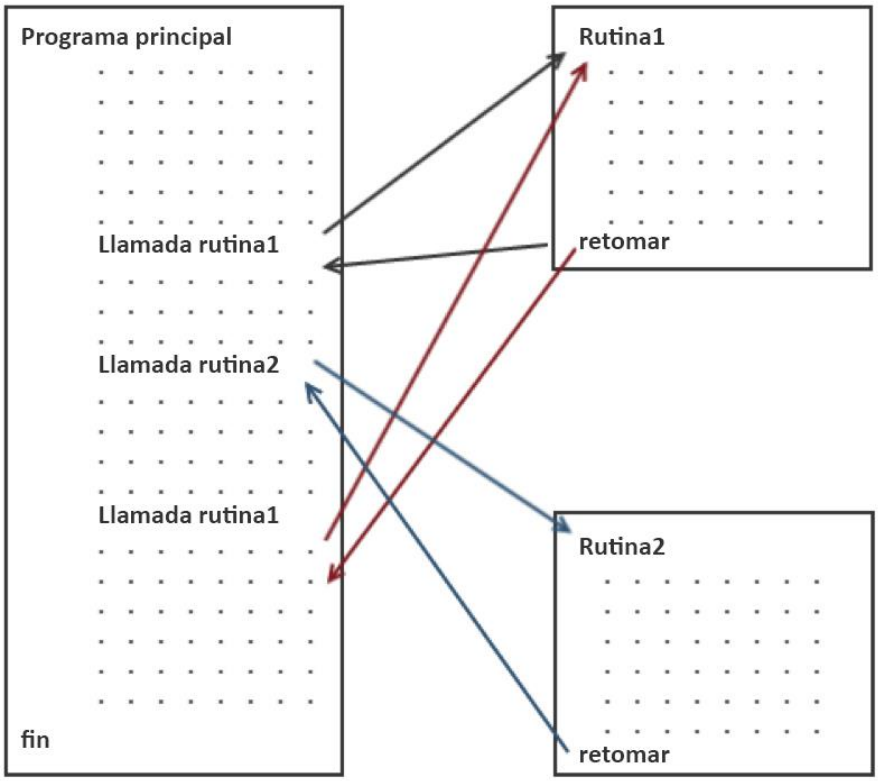
En detalle - ¿Qué pasa si se extiende esto hasta N niveles de retorno?

La solución a esto es la pila de datos. De manera que ya no solo se puede almacenar todos los que se necesiten (mientras se tenga memoria libre), sino que además se tienen **organizados** los puntos de retorno del **más actual al más antiguo**. De manera que se puede deshacer el camino de llamadas a funciones deshaciendo los elementos guardados en la pila.

La solución, pasa, porque cada vez que se llama a una rutina, la dirección de memoria del punto de programa en ejecución (dirección de retorno) debe ser almacenado. Recordemos que ese punto de retorno, es el valor del contador de programa justo antes de hacer el salto a la función.

La instrucción en ensamblador que hace estas dos tareas es la función call:

- Almacenar el valor actual del contador de programa.
- Saltar al inicio de la función.



Como el almacenamiento del punto de retorno, ya se realiza en memoria, da la posibilidad de manejar una lista ilimitada de llamadas de rutina, incluyendo también la recursividad (llamando a un procedimiento desde si mismo).

Memoria
Este espacio de memoria, **funcionará como una pila** (*stack* en inglés), donde los nuevos datos se amontonan en la cima (push) sobre los datos anteriores, manteniendo el orden de llegada. Al recuperar datos (pop), el primer valor obtenido es el de la parte superior de la pila (cima), que coincide con el último valor en ser almacenado.

La pila (STACK)

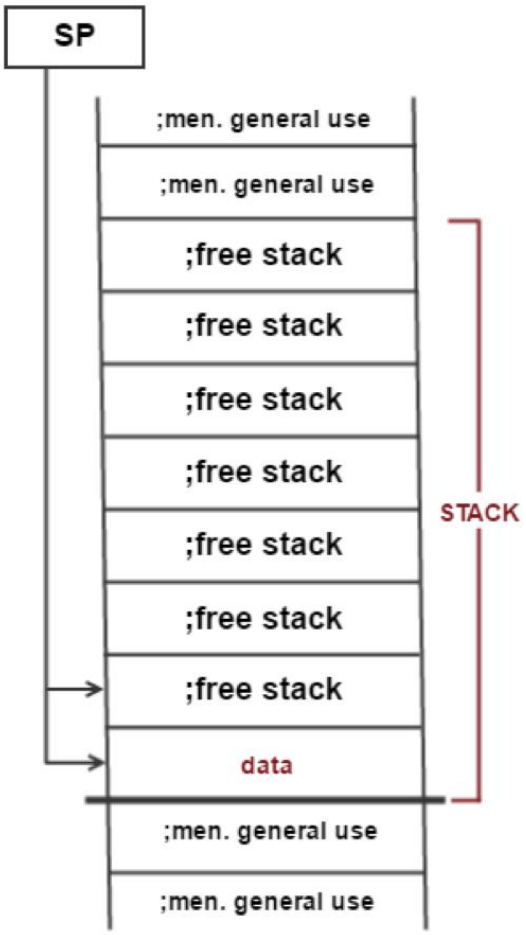
Ante todo, dejar claro que la pila es una **región normal de la memoria**, solo que se puede utilizar de una manera especial. Es la forma de usar esta porción de memoria la que le da sus propiedades, no la construcción física de la misma.

La principal característica de la pila, es que tiene **dos índices** (punteros) que sirven para llevar un control de cómo está siendo utilizada esta pila. Normalmente estos punteros son dos registros especiales:

SP
Puntero pila = Stack Pointer. Apunta a la **cima de la pila**, al último elemento almacenado.
BP
Puntero base = Base Pointer. Apunta al **inicio de la pila**, al primer elemento guardado.

Inicialmente, cuando la pila está vacía, los punteros SP y BP **apuntan al mismo sitio**, al inicio de la memoria donde comienza la pila.

En detalle - Código para la inicialización del registro de pila



Los microcontroladores Atmel(***), **no tiene registro BP**, por lo que no hay manera de saber ni de controlar cuando la pila está vacía.

A medida que se insertan elementos en la pila, se disminuye de valor SP. Sí, se disminuye su valor, ya que

Instrucciones para manejar la pila

Para mantener cierto orden y coherencia en la pila, esta no se puede (más bien no se debe) utilizar libremente, ya que, al ser memoria, sí se puede alterar el valor libremente, pero su comportamiento quedaría errático. Ensamblador incorpora dos instrucciones para manejar la pila:

PUSH: PUSH Rr ;	Pone el contenido del registro Rr en la pila, y luego **decrementa SP** en una unidad. Esta instrucción necesita dos ciclos de reloj.
POP: POP Rd ;	Saca un byte de la cima de la pila y lo almacena en **registro Rd**, a continuación, incrementa SP en una unidad.

Dado que Atmel, permite direccionamiento de SRAM de 16 bits, obliga a que el registro PC sea de 16 bits. Por tanto, el almacenamiento del registro PC en la pila, utilizado en las llamadas a las funciones, provoca realmente el almacenamiento de dos valores [SPH:SPL] de 8 bits cada uno.

El registro SP (y por consiguiente SPH y SPL), solo se puede acceder a él utilizando la instrucción OUT y no la LD como otros registros genéricos. Aun así, se desaconseja la modificación manual de este registro, las consecuencias son indeterminadas.

Ya que Atmel no utiliza un indicador de inicio de pila, (si no se especifica lo contrario), toma la dirección de memoria más alta disponible, como base de la pila, y empieza a llenar a partir de ahí. Esta dirección se define como la **RAMEND**, que es una constante definida en la especificación del microcontrolador utilizado (m328pdef.inc en nuestro caso) y tiene un **valor específico** para cada microprocesador.

En detalle - Código para la inicialización del registro de pila

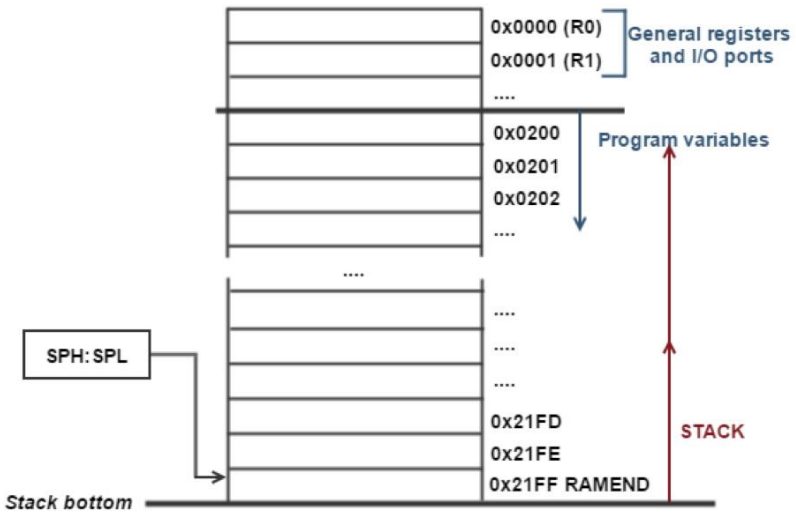
Actualmente, ya no es necesario inicializar la variable SP para que apunte a la posición correcta, pero en algunos compiladores antiguos, esto si lo es, por lo que el código necesario para la inicialización del

Así, la **distribución de memoria** de programa de Atmel, queda como sigue:

- Las primeras posiciones de la memoria de datos SRAM se asignan a los registrosR0...R31
- Luego los puertos I/O
- Después variables del programa
- Y, por último, las posiciones finales de SRAM son utilizado por la pila.

De esta manera, se **maximiza el uso de pila y de datos de programa**. A medida que la pila se llena ocupa SRAM, si el tamaño de la pila crece en exceso, podría solaparse con las variables del programa, provocando la sobreescritura con valores aleatorios.

Distribución de la memoria de programa de Atmel



Invocación: Instrucciones CALL Y RCALL

La instrucción que permite hacer llamadas a las funciones son:

CALL (llamada absoluta)

- Su formato es **CALL n**.

1001	010a_{21}	$a_{20}a_{19}a_{18}a_{17}$	111a_{16}
$a_{15}a_{14}a_{13}a_{12}$	$a_{11}a_{10}a_9a_8$	$a_7a_6a_5a_4$	$a_3a_2a_1a_0$

  - Almacena la dirección de retorno (PC + 2) en la pila (STACK ← PC + 2, SP ← SP-2).
  - carga la dirección absoluta de la rutina en el PC (PC ← n).
- Donde 0 < n < 32K. Sólo disponible en algunas CPUs (Central Processing Unit) Atmel (con gran memoria de programa).
- Es una instrucción **grande y lenta** (ocupa dos posiciones de memoria de programa, y toma cuatro ciclos de reloj).
- La dirección de la rutina puede indicarse mediante una **etiqueta en el código**.

RCALL (Relative CALL)

Su formato es RCALL r

1001	$r_{11}r_{10}r_9r_8$	$r_7r_6r_5r_4$	$r_3r_2r_1r_0$

  - Almacena la dirección de retorno (PC + 1) en la Pila (STACK ← PC + 1, SP ← SP-2).
  - Incrementa el valor de PC en r (PC ← PC + r + 1).
- Donde r puede ser la **distancia positiva o negativa** a la dirección de la rutina, -2K <r <2K.
- Aumenta o disminuye (si r es negativo) el valor de PC con respecto al valor actual. Es por eso que esta llamada es relativa (relativa a la posición actual del programa).
- La instrucción **es más corta y más rápida** que CALL, ocupa una posición de memoria y necesita tres ciclos de reloj para ejecutarse.
- La distancia a la dirección de la rutina puede indicarse mediante una **etiqueta** en el código.

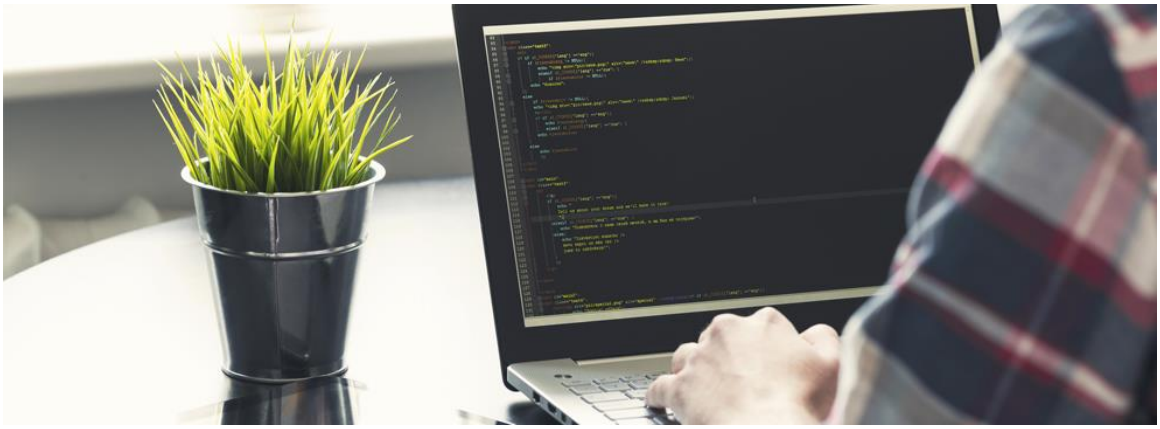
Retorno de funciones: instrucción RET (RETorno de la rutina)

Su formato es: **RET**

- Extrae los 2 bytes de la cima de la pila y los carga en el contador de programa, PC ← STACK (2 bytes).
- Incrementa el valor de SP en dos unidades, SP ← SP + 2.

Esta instrucción permite la **vuelta al punto de llamada** desde una función mediante la recuperación de la dirección de retorno de la pila, previamente acumulado por una llamada CALL o RCALL.

RET siempre utiliza los dos bytes de la cima de la pila como una dirección de retorno. Por lo general, esos 2 bytes, son realmente la dirección de retorno. Pero si la pila se ha utilizado por el programador para cualquier otro propósito (lo cual es muy frecuente, pues su uso está muy estandarizado para otras funcionalidades), los valores más altos de la pila podrían ser otros valores en lugar de la dirección de retorno correcta, lo que origina que RET vuelva a una dirección incorrecta, provocando un comportamiento aleatorio en el programa.



Uso de registros en funciones

Como ya se ha repetido varias veces, un registro debe ser **autónomo**, es decir, no debe necesitar nada de fuera del entorno de la propia función.

Si este necesita información del exterior, esta información debe ser suministrada en el momento de llamar a la función.

Este tipo de interacción incorrecta se realiza normalmente a través de **variables globales**, que como ya se conoce, su uso esta "(casi) prohibido" por los problemas que trae un uso indebido de ellas.

Pero como sabemos, la mayor parte de las instrucciones en ensamblador operan sobre **registros**. Los registros, los podemos ver como variables globales a todo el microprocesador. Y en este caso, su uso no está prohibido, sino que más bien es obligatorio. Lo que conlleva a que una modificación en un registro dentro de una función, puede provocar alteraciones en variables almacenadas ahí por el programa principal. Es decir, es un caso similar al uso de variables globales.

El uso de registros dentro en una función, es **practica necesaria** (es casi imposible programar sin usar los registros) por lo que los efectos colaterales están asegurados. Pero este problema tiene solución: **la pila**.

Si el código de una función necesita utilizar algún registro de propósito general, es probable que este registro ya se utilice en el programa principal, por lo que su valor actual sea importante, por lo que es necesario hacer una copia de seguridad del mismo, para poder restaurarlo cuando se termine de utilizar.

Copia de seguridad
Si una rutina utiliza un registro, antes de modificar su contenido (por lo general al principio de la función) se debe **salvar su contenido** almacenándolo en la pila (PUSH). Para luego, justo antes del RET, poder hacer un restaurar (POP) al valor original antes de la llamada a la función.

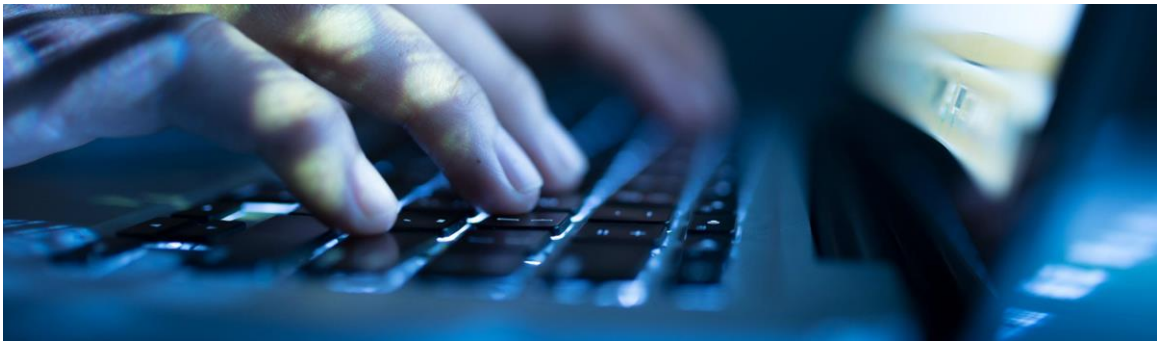
Ejemplo - Iniciar el puerto B como salida y ponerlo a unos. Función InitBPort

```
; blink_funciones.asm
; Author : Sergio B.

    ldi r19, 0x00       ;Inicializamos r19 a cero, ya ue se usara como registro de salida
    call InitBPort      ;Inicializamos el puerto B a salida

P_Principal:
    com r19             ;Hacer el Complemento a 1. Invertir bits el reg. 19
    out PORTB, r19      ;Sacamos por el puerto B en contenido de r19. Esto inclulle el led en el pin 5.
    jmp P_Principal     ;Bucle infinito del Programa Principal

InitBPort:
    push r19            ;Salbamos el contenido actual de r19
    ldi r19, 0xff       ;Inicializamos r19 a todo unos.: 11111111
                        ;Modificamos el registro r19, que tambien se usa en el P.Principopal
    out DDRB, r19       ;Ponemos el puerto B como todo salida.
    out PORTB, r19      ;Sacamos por el puerto B todos unos.
    pop r19             ;Restauramos el contenido de r19
    ret                 ;Terminamos la rutina InitBPort
```



Resumen

Las funciones son la manera de **agrupar funcionalidades** y ayudar a su reutilización, base del paradigma de programación estructurada.

Las funciones necesitan el concepto de **almacenamiento de información** (TAD = Tipo Abstracto de Datos) Pila. Que se usará para almacenar el punto de llamada y por consiguiente saber cuál es el punto de retorno una vez ejecutada la función.

La Pila, además se usará para hacer **copia de seguridad de los registros** modificados por una función, lo que la hará totalmente independiente del programa principal e inocua para él, al eliminar los efectos colaterales de la modificación de variables globales.

Las instrucciones vistas en este tema son:

- **CALL:** llama a una función de manera absoluta
- **RCALL:** llamada a una función de manera relativa
- **RET:** vuelve al punto de llamada a una función en el programa principal
- **PUSH:** guarda un byte en la cima de la pila.
- **POP:** extrae el byte de la cima de la pila.