

UNIDAD

6

DIPLOMATURA EN PROGRAMACION ABAP
MÓDULO 6: MODULARIZACIÓN

MODULARIZACIÓN

MODULARIZACIÓN

Este módulo introduce las tres primeras de las cinco unidades de modularización que posee ABAP/4: includes, subrutinas y macros. Quedan para la próxima unidad las otras dos, funciones y eventos

Unidades de modularización en ABAP / 4

Una unidad de modularización es como una *shell* en la que se puede colocar el código. Le permite segregar un grupo de líneas de código del resto y, a continuación, ejecutarlas en un momento específico. Las líneas de código dentro de una unidad de modularización actúan muy parecido a un mini-programa que se puede llamar desde otro programa.

ABAP / 4 ofrece cinco tipos de unidades de modularización:

- *Includes*
- Subrutinas
- Macros
- Eventos
- Módulos de función

Esta unidad explica los tres primeros. En la siguiente, se explicarán los módulos de función y los eventos.

Utilice unidades de modularización para eliminar código redundante dentro de su programa y para hacer que su programa sea más fácil de leer. Por ejemplo, suponga que tiene una serie de declaraciones que formatean una dirección de correo, y lo que necesita para dar formato a las direcciones de correo está en varios lugares diferentes en su programa. En lugar de duplicar el código dentro de su programa, es una buena idea poner ese código en una unidad de modularización y llamarla siempre que la necesite para dar formato a una dirección.

Comenzaremos entonces explicando los *includes*.

Inclusión de Código (Concepto de *Include*)

Un *programa include* es un tipo de programa en el que los contenidos están diseñados para ser utilizados por otro programa. Generalmente no se completan por sí mismos. En cambio, otros programas utilizan el código del programa *include* copiando las líneas de código del programa *include* a sí mismos. La copia se realiza en tiempo de ejecución a través de la sentencia *include*. Un programa *include* también se conoce como un *programa incluido*. Un programa que incluye a otro se conoce como un *programa incluyente*.

La siguiente es la sintaxis para el comando `include`.

```
incluir ipgm.
```

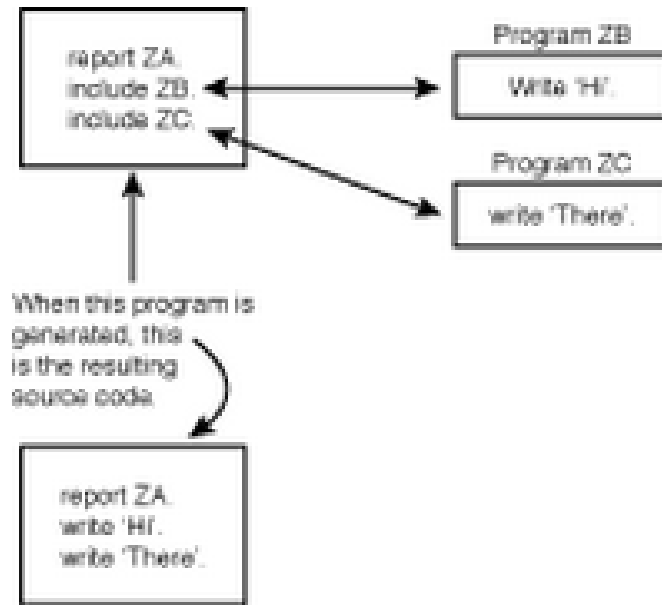
donde:

- `ipgm` es un programa de tipo `i` (recordar que el tipo de programa se define en la pantalla de Atributos del mismo).

Los siguientes puntos se aplican:

- Un programa *include* debe ser de tipo `i`. Se especifica el tipo de programa en el campo Tipo en la pantalla Atributos del Programa, cuando se crea el programa.
- Un programa *include* puede ser incluido en uno o más programas.
- Un programa tipo `i` no puede contener declaraciones parciales o incompletas.

La sentencia `include` copia los contenidos del programa incluido en el programa incluyente. El código del programa *include* se copia tal cual y sustituye a la declaración `include` en el momento de la generación del programa. La situación puede esquematizarse en la siguiente figura:



En la figura, ZA es el programa que incluye. Los programas include son ZB y ZC . En el momento que ZA se genera, el código fuente de ZB y ZC se inserta en ZA , y lo que resulta del programa se muestra en la parte inferior de la figura.

Un ejemplo de código podría ser éste:

```
1 report ztx01.
2 tables spfli.
3
4
5 include: ztx02.
6
7 write: / spfli-carrid.
```

Donde el código para el include sería éste:

```
1 ***INCLUDE ZTX02.
2 select single spfli-carrid from spfli.
```

El *include* se crea y se guarda aparte, como tipo *i*.

A la hora de la generación de ztx01 , la línea 5 copia el código del programa ztx02 en ztx01 . El código introducido sustituye a la línea 5.

Cuando el programa se ejecuta, ztx01 se comporta como un solo programa, como si esas líneas de código de los programas incluidos se hubiesen escrito directamente en ztx01.

Los `include` de SAP se usan para reducir la redundancia del código y para dividir programas muy grandes en unidades más pequeñas.

TIP

Mientras ve un programa que incluye, puede ver el contenido de un programa incluido simplemente haciendo doble clic sobre su nombre. Por ejemplo, durante la edición de `ztx1901` en el Editor ABAP / 4, pantalla de edición de programa, haga doble clic en el nombre `ztx1902`. El programa incluido se mostrará de inmediato.

Pasemos ahora a analizar las subrutinas.

Subrutinas

Una *subrutina* es una sección de código reutilizable. Es como un mini-programa que se puede llamar desde otro punto de su programa. Dentro de ella se puede definir variables, ejecutar instrucciones, calcular los resultados y escribir la salida. Para definir una subrutina, utilizar la declaración `form` para indicar el inicio de una subrutina, y utilizar `endform` para indicar el final de la subrutina. El nombre de una subrutina no puede superar los 30 caracteres.

Para llamar a una subrutina, utilice la declaración `perform` .

He aquí un programa de ejemplo que define y llama a una subrutina:

```
1  report ztx03.
2
3  write: / 'Antes de la llamada 1'.
4  perform sub1.
5  write: / ' Antes de la llamada 2'.
6  perform sub1.
7  write: / ' Después de las llamadas'.
8
9  form sub1.
10     write: / 'Dentro de sub1'.
11 endform.
```

La salida del listado sería la siguiente:

Antes de la llamada 1
Dentro de sub1
Antes de la llamada 2
Dentro de sub1
Después de las llamadas

Donde:

- La línea 3 se ejecuta.
- La línea 4 transfiere el control a la línea 9.
- La línea 10 se ejecuta.
- La línea 11 transfiere el control de nuevo a la línea 4.
- La línea 5 se ejecuta.
- La línea 6 transfiere el control a la línea 9.
- La línea 10 se ejecuta.
- La línea 11 devuelve el control a la línea 6.
- La línea 7 se ejecuta.

Hay dos tipos de subrutinas:

- Subrutinas internas
- Subrutinas externas

El listado anterior ilustra una llamada a una subrutina interna. En una subrutina interna, las definiciones de subrutina se colocan generalmente en el final del programa. La declaración `form` define el inicio de una subrutina. Las subrutinas no se pueden anidar.

La sintaxis para la declaración `form` es:

```
form s [tables t1 t2 ...]  
      [using u1 value(u2) ...]  
      [changing c1 value(c2) ...].  
---  
endform.
```

donde:

- `s` es el nombre de la subrutina.

- *T1* , *T2* , *U1* , *U2* , *C1* , y *C2* son parámetros (en seguida desarrollaremos el tema de parámetros).
- *tables* permite a las tablas internas que se pasen como parámetros (el tema de tablas internas se desarrollará más adelante, en próximas unidades).
- *value* no se puede utilizar después de *tables* .
- *value* se puede aplicar a cualquier variable pasada a través de *using* o *changing* .
- --- representa cualquier número de líneas de código.

Los siguientes puntos se aplican:

- Todas las adiciones son opcionales.
- Cuando se codifican adiciones deben aparecer en el orden que se muestra aquí. Si es codificada, *tables* deben venir primero, y luego *using* , y luego *changing* .
- Cada adición sólo se puede especificar una vez. Por ejemplo, *tables* sólo puede aparecer una vez. Sin embargo, varias tablas pueden aparecer después de ella.
- No utilice comas para separar los parámetros.
- *tables* sólo permite a las tablas internas pasar- no tablas de la base de datos física.
- Una subrutina puede llamar a otra subrutina.
- La recursividad es compatible. Una subrutina puede llamar a sí misma o a una subrutina que llama a sí misma.
- Las definiciones de subrutinas no se pueden anidar. (No se puede definir una subrutina dentro de otra subrutina.)

La sintaxis de la sentencia `perform` es:

```
perform a) s
      b) n of s1 s2 s3 ...

      [tables t1 t2 ...]
      [using u1 u2 ...]
      [changing c1 c2 ...].
```

donde:

- `s`, `s1`, `s2`, `s3`, son nombres de subrutinas.
- `n` es una variable numérica.
- a) y b) son mutuamente excluyentes.
- `tables`, `using` y `changing` pueden aparecer ya sea en a) o en b) .
- La adición `value()` no se puede utilizar con `perform` .

Utilizando la sintaxis b) se puede especificar que una de las subrutinas una lista de subrutinas se realice. La `n`-sima subrutina en la lista de nombres de subrutina siguientes a `of` es la que se realiza. Por ejemplo, si `n` es 2, se realizará la segunda subrutina en la lista.

Veamos un ejemplo:

```
1  report ztx04.
2  do 3 times.
3      perform sy-index of s1 s2 s3.
4  enddo.
5
6  form s1.
7      write: / 'Hola from s1'.
8  endform.
9
10 form s2.
11     write: / 'Hola from s2'.
12     endform.
13
14 form s3.
15     write: / 'Hola from s3'.
16 endform.
```

El código del listado genera el siguiente resultado:

```
Hola desde s1
Hola desde s2
Hola desde s3
```

Analicémoslo:

- La línea 2 comienza un bucle que se ejecuta tres veces.
- `sy-index` es el contador de bucles, y se incrementa en uno con cada pasada del bucle `do...enddo`.
- La primera vez que se ejecuta la línea 3, `sy-index` tiene un valor de 1. Por lo tanto, la primera subrutina (`s1`) se lleva a cabo.
- La segunda vez que la línea 3 se ejecuta, `sy-index` tiene un valor de 2. Por lo tanto, la segunda subrutina (`s2`) se lleva a cabo.
- La tercera vez que la línea 3 se ejecuta, `sy-index` tiene un valor de 3. Por lo tanto, la tercera subrutina (`s3`) se lleva a cabo.

Puede salir de una subrutina en cualquier momento con las siguientes declaraciones:

- `exit`
- `check`
- `stop`

Los siguientes párrafos describen el efecto de la `check` y de `exit` cuando se codifican dentro de una subrutina pero fuera de un bucle. El efecto de `stop` dentro de una subrutina es el mismo, independientemente de si se es codificado dentro de un bucle o no.

En subrutinas:

- `check` y `exit` dejará inmediatamente a la subrutina y el procesamiento continúa con la siguiente instrucción ejecutable tras `perform`.
- `stop` inmediatamente deja la subrutina y se va directamente al evento `end-of-selection` (que se verá en detalle en la próxima unidad).

`check`, `exit`, y `stop` no modifican ni establecen por sí solos el valor de `sy-SUBRC`.

Veamos un ejemplo del uso de estas 3 declaraciones:

```
1  report ztx05.
2  data f1 value 'X'.
3
4  clear sy-subrc.
5  perform s1. write: / 'sy-subrc =', sy-subrc.
6  perform s2. write: / 'sy-subrc =', sy-subrc.
7  perform s3. write: / 'sy-subrc =', sy-subrc.
8  perform s4. write: / 'sy-subrc =', sy-subrc.
9
10 end-of-selection.
11     write: 'Stop, sy-subrc =', sy-subrc.
12     if sy-subrc = 7.
13         stop.
14     endif.
15     write: / 'Después del Stop'.
16
17 form s1.
18     do 4 times.
19         exit.
20     enddo.
21     write / 'En s1'.
22     exit.
23     write / 'Después del Exit'.
24 endform.
25
26 form s2.
27     do 4 times.
28         check f1 = 'Y'.
29         write / sy-index.
30     enddo.
31     write / 'En s2'.
32     check f1 = 'Y'.
33     write / 'Después del Check'.
34 endform.
35
36 form s3.
37     do 4 times.
38         sy-subrc = 7.
39         stop.
40         write / sy-index.
41     enddo.
42 endform.
43 form s4.
44     write: / 'En s4'.
45 endform.
```

La salida que se producirá será la siguiente:

```
En s1
sy-subrc =      0
En s2
sy-subrc =      0
Stop, sy-subrc =      7
```

El análisis es el siguiente:

- La línea 5 transfiere el control a la línea 17.
- La línea 18 comienza un bucle. La línea 19 sale del bucle, no de la subrutina.
- La línea 22 sale de la subrutina. El control vuelve a la línea 6.
- La línea 6 transfiere el control a la línea 26.
- La línea 28 transfiere el control a la línea 27 cuatro veces seguidas.
- La línea 32 sale de la subrutina. El control retorna a la línea 7.
- La línea 7 transfiere el control a la línea 36.
- La línea 39 transfiere el control directamente a la línea 10.
- La salida de la línea 11 comienza en una nueva línea porque un nuevo evento ha comenzado.
- La línea 13 sale del evento y la lista se muestra.

Veremos ahora el tema del pasaje de parámetros entre subrutinas.

Además de la definición común y corriente de variables, a las subrutinas se les puede pasar parámetros al invocarlas con `perform`, lo que conlleva a la definición de variables en la misma sentencia `form`. Estas variables se conocen como parámetros *formales*, mientras que los que aparecen en `perform` son los parámetros *actuales*.

Por ejemplo, el siguiente código:

```
1  report ztx06.
2  data: f1 value 'A',
3         f2 value 'B',
4         f3 value 'C'.
5
6  perform: s1 using f1 f2 f3,
7           s2 using f1 f2 f3.
8
9  form s1 using p1 p2 p3.
10     write: / f1, f2, f3,
11            / p1, p2, p3.
12     endform.
13
14 form s2 using f1 f2 f3.
15
16     write: / f1, f2, f3.
17     endform.
```

Producirá la siguiente salida:

```
A B C
A B C
A B C
```

El análisis es el siguiente:

- La línea 6 transfiere el control a la línea 9.
- La línea 9 define tres variables: p1 , p2, y p3 . Asigna el valor de f1 a p1 , el valor de f2 a p2 , y el valor de f3 para p3 .
- La línea 10 escribe los valores de las variables p1 , p2 y p3 y las variables globales del programa f1 , f2 y f3 .
- La línea 12 devuelve el control a la línea 6.
- La línea 7 transfiere el control a la línea 14.
- La línea 14 define tres variables: f1 , f2 , y f3 . Asigna el valor de f1 a f1 , el valor de f2 a f2 , y el valor de f3 a f3 .
- La línea 16 escribe los valores de las variables definidas en la línea 14: f1, f2 y f3 .
- La línea 17 devuelve el control a la línea 8 y termina el programa.

Los parámetros pueden ser tipados o no.

Veamos un ejemplo con código para aclararlo:

```
1 report ztx07.
2 data: f1 value 'A',
3       f2 type i value 4,
4       f3 like sy-datum,
5       f4 like sy-uzeit.
6
7 f3 = sy-datum. * fecha actual
8 f4 = sy-uzeit. * hora actual
9
10 perform s1 using f1 f2 f3 f4.
11
12 form s1 using p1 type c
13           p2 type i
14           p3 type d
15           p4 type t.
16   write: / p1,
17         / p2,
18         / p3,
19         / p4.
20   endform.
```

La salida obtenida es:

```
AAA
      4
20140325
190000
```

El análisis es el siguiente:

- Las líneas 2 a 5 definen cuatro variables que tienen diversos tipos de datos. F3 es de tipo C longitud 8, con una longitud de salida de 10 (definido en el dominio para el sy-datum). F4 es de tipo C longitud de 6 con una longitud de salida de 8 (define en el dominio para el SY-uzeit).
- La línea 10 transfiere el control a la línea 12.
- En la línea 12, p1 sólo acepta parámetros reales de tipo c . f1 es de tipo c de longitud 3, por lo que una longitud de 3 se asigna a p1 . Si f1 no hubiera sido de tipo c un error de sintaxis se habría producido. p2 es un tipo de datos de longitud fija, tipo i que es siempre la longitud 4; f2 también, por lo que los parámetros coinciden plenamente, y p3 y p4 son de tipos de datos de longitud fija también , así como sus parámetros reales.

- Las líneas 16 a 19 escriben los valores de p1 a p4 . Observe que la longitud de salida no se pasa del parámetro actual al parámetro formal. La longitud de salida de los parámetros formales se establece en el valor predeterminado para cada tipo de datos. Esto hace que la fecha y la hora de salida se vean sin separadores, como están predefinidas.

Se pueden pasar parámetros a una subrutina por referencia, por valor o por valor y resultado.

Es la sintaxis de `form`, y no la de `perform`, la que me indica el tipo de pasaje, lo cual se verá muy bien en el siguiente ejemplo.

Antes veamos la siguiente tabla, referida a la sintaxis de `form`:

Adición	Método
<code>using v1</code>	Pasar por referencia
<code>changing v1</code>	Pasar por referencia
<code>Using value (v1)</code>	Pasar por valor
<code>Changing value (v1)</code>	Pasar por valor y resultado

Ahora sí, el ejemplo de código:

```

1  report ztx08.
2  data: f1 value 'A',
3         f2 value 'B',
4         f3 value 'C',
5         f4 value 'D',
6         f5 value 'E',
7         f6 value 'F'.
8
9  perform s1 using f1 f2
10             changing f3 f4.
11
12 perform s2 using f1 f2 f3 f4
13             changing f5 f6.
14
15 perform s3 using f1 f2 f3.
16
17 form s1 using p1 value(p2)
18             changing p3 value(p4).
19 write: / p1, p2, p3, p4.
20 endform.
21
22 form s2 using p1 value(p2) value(p3) p4
23             changing value(p5) p6.
24 write: / p1, p2, p3, p4, p5, p6.
25 endform.
```

```
26
27 form s3 using value(p1)
28     changing p2 value(p3).
29     write: / p1, p2, p3.
30     endform.
```

La salida obtenida será ésta:

A B C D
A B C D E F
A B C

El análisis que puede efectuarse es el siguiente:

- La línea 9 pasa cuatro parámetros a la subrutina s1 . La sintaxis de la línea 17 determina la forma en que se transmiten. f1 y f3 se pasan por referencia; f2 se pasa por valor; f4 se pasa por valor y resultado.
- La línea 12 pasa seis parámetros a la subrutina s2 . f1 , f4 y f6 se pasan por referencia, f2 y f3 se pasan por valor. f5 se pasa por valor y resultado.
- La línea 15 pasa tres parámetros a la subrutina s3, todos por referencia.
- Es la sintaxis de `form`, y no la de `perform`, la que prevalece en caso de dudas o inconsistencia entre ambas. Nunca preste atención a `perform` para saber cómo se pasan los parámetros a una subrutina, pues esto lo dicta `form`.

Recuerde, eso sí, que debe tener las siguientes cosas en mente:

- `perform` y `form` debe contener el mismo número de parámetros.
- La sintaxis entre `perform` y `form` puede diferir.
- La sintaxis de la declaración `form` por sí misma determina el método por el cual se pasa un parámetro.
- `value()` no se puede utilizar con la declaración `perform` .
- `using` debe venir antes de `changing` .
- La adición `using` sólo puede ocurrir una vez en una sentencia. La misma regla se aplica a `changing`.

Resumen de los métodos de pasaje de parámetros a subrutinas:

Método	Descripción	Ventajas
Por referencia	Pasa un puntero a la ubicación de memoria original.	Muy eficaz
Por valor	Asigna una nueva ubicación en la memoria para su uso dentro de la subrutina. La memoria se libera cuando finaliza la subrutina.	Evita cambios a variable pasada
En términos de valor y resultado	Similar a pasar por valor, pero el contenido de la nueva memoria se copia de nuevo en la memoria original antes de volver.	Permite cambios y permite una operación de deshacer

Veamos estos tres métodos un poco más en detalle.

Comencemos con el pasaje por referencia.

Cuando se pasa un parámetro por referencia, la nueva memoria no se reserva para el valor. En lugar de ello, se pasa un puntero a la ubicación de memoria original. Todas las referencias al parámetro son las referencias a la ubicación de memoria original. Los cambios en la variable dentro de la subrutina actualizan la ubicación de memoria original inmediatamente. La figura y el código a continuación ilustran cómo funciona esto:

```

report ztx1884.
data f1 value 'A'.

perform s1 using f1.
write / f1. output: X

form s1 using p1. p1 is a pointer to memory location 1000
  p1='X'. this changes memory location 1000 to 'X'
endform.
  
```

memory address 1000

before call to s1: A

after assignment in s1: X


```
1 report ztx09.
2 data f1 value 'A'.
3
4 perform s1 using f1.
5 write / f1.
6
7 form s1 using p1.
8     p1 = 'X'.
9     endform.
```

El código del listado genera el siguiente resultado:

x

El análisis del mismo nos dice lo siguiente:

- La línea 2 asigna memoria para la variable `f1` . Por el bien de este ejemplo, vamos a asumir que la posición de memoria es de 1000.
- La línea 4 transfiere el control a la línea 7.
- La línea 7 provoca que `f1` sea pasado por referencia. Por lo tanto, `p1` es un puntero a la ubicación de la memoria 1000.
- La línea 8 modifica la posición de memoria 1000, causando que el valor almacenado en la memoria para `f1` cambie a x .
- La línea 9 devuelve el control a la línea 5.
- La línea 5 escribe el valor x .

Cuando se pasa un parámetro por valor, la nueva memoria se asigna para el valor. Esta memoria se asigna cuando la subrutina es llamada y se libera cuando se vuelve de la subrutina. Por lo tanto, las referencias a los parámetros son referencias a un área de memoria única que se conoce sólo dentro de la subrutina; la ubicación de memoria original es separada. El original no se modifica si se cambia el valor del parámetro. La figura y el código a continuación ilustran cómo funciona esto:

```

report ztx1806.
data f1 value 'A'.
perform s1 using f1.
write / f1.  output: A

form s1 using value(p1).
  p1 = 'X'.
endform.

```

Diagram illustrating variable scope and value passing:

- A box labeled 'A' is shown next to the line `data f1 value 'A'.` with the text "before call to s1" and "after assignment in s1".
- A box labeled 'X' is shown next to the line `p1 = 'X'.` with the text "independent copy of f1".

```

1 report ztx10.
2 data: f1 value 'A'.
3
4 perform s1 using f1.
5 write / f1.
6
7 form s1 using value(p1).
8   p1 = 'X'.
9   write / p1.
10  endform.

```

La salida es:

X
A

El análisis es éste:

- La línea 2 asigna memoria para la variable f1 .
- La línea 4 transfiere el control a la línea 7.
- La línea 7 provoca que f1 se pase por valor. Por lo tanto, p1 se refiere a una nueva ubicación de memoria que es independiente de f1 . El valor de f1 se copia automáticamente en la memoria asignada para p1 .
- La línea 8 modifica la memoria para p1 . f1 no se modifica.
- La línea 9, escribe el valor X .
- La línea 10 devuelve el control a la línea 5.
- La línea 5 escribe A .

Utilice el paso por valor cuando se necesita una copia local de una variable que se pueda modificar sin afectar el original. Pasar por referencia es más eficiente que el paso por valor. Utilice paso por referencia a menos que necesite una copia local independiente de la variable.

Finalmente, analicemos el pasaje por valor y resultado.

Pasar por valor y resultado es muy similar al paso por valor. Al igual que el paso por valor, una nueva área de memoria se asigna y se mantiene una copia independiente de la variable. Se libera cuando termina la subrutina, y que es también cuando se produce la diferencia.

Cuando el `endform` se ejecuta, se copia el valor del área de memoria local de nuevo en el área de memoria original. Los cambios en el parámetro dentro de la subrutina se reflejan en el original, pero no hasta que la subrutina termina.

Esta puede parecer una diferencia pequeña, pero la diferencia es mayor. Usted puede cambiar si la copia se lleva a cabo o no.

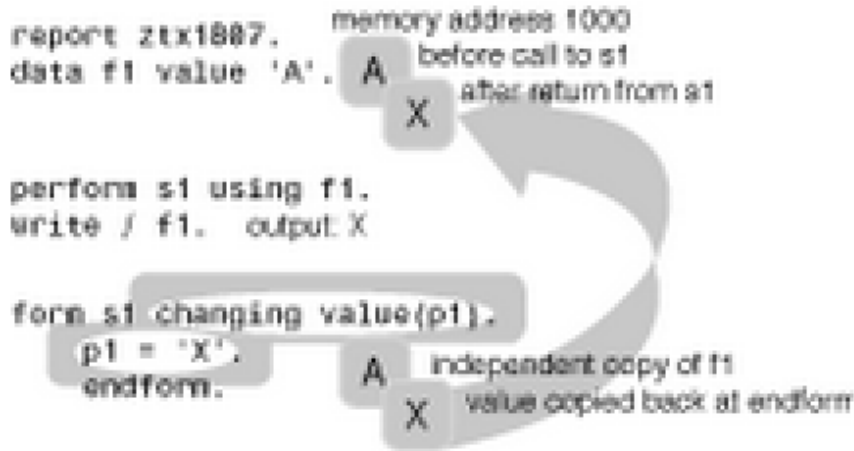
La copia se realiza siempre a no ser que salga de la subrutina antes de terminar, utilizando una de las dos declaraciones:

- `stop`
- `message nnn`

La declaración `stop` finaliza la subrutina y va directamente al evento `end-of-selection`. Si `p1` se pasa por valor y resultado, los cambios en `p1` se descartan antes de que se active `end-of-selection`. En cierto sentido, `stop` se comporta como un mini-retroceso para los parámetros de valor y de resultados. Cuando se utiliza dentro de una subrutina, la declaración `stop` suele ir precedida de una prueba para una condición anormal dentro del programa. Si surge la condición anormal, `stop` se ejecuta. Se descartan los cambios y se va directamente a `end-of-selection`, donde a continuación, se ejecutan los procedimientos de limpieza.

Utilice el paso por valor y el resultado para los parámetros que desea cambiar, pero donde puede haber una posibilidad de que usted vaya a querer descartar los cambios si una condición anormal se presenta en su subrutina. En cuanto a la declaración `message nnn`, que indica un mensaje de error con terminación anormal de la subrutina, la veremos más adelante.

La figura y el código siguientes ilustran el funcionamiento de los parámetros pasados por valor y resultado:



```

1  report ztx11.
2  data: f1 value 'A'.
3
4  perform: s1 changing f1,
5           s2 changing f1.
6
7  end-of-selection.
8      write: / 'Stop. f1 =', f1.
9
10 form s1 changing value(p1).
11     p1 = 'B'.
12     endform.
13
14 form s2 changing value(p1).
15     p1 = 'X'.
16     stop.
17     endform.

```

La salida del reporte es:

Stop. f1 = B

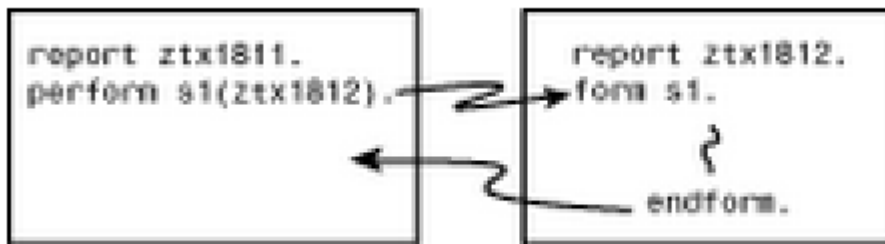
El análisis del código es éste:

- La línea 2 asigna memoria para la variable `f1` .
- La línea 4 transfiere el control a la línea 9.
- La línea 10 causa que `f1` sea pasado por valor y resultado. Por lo tanto, `p1` se refiere a una nueva ubicación de memoria que es independiente de `f1` . El valor de `f1` se copia automáticamente en la memoria asignada para `p1` .

- La línea 11 modifica la memoria para `p1`. `f1` sigue sin cambios.
- La línea 12 copia el valor nuevo de `p1` en `f1`, libera `p1`, y transfiere el control a la línea 5.
- La línea 5 pasa el control a la línea 14.
- La línea 14 causa que `f1` sea pasado por valor y resultado. Una nueva área de memoria se asigna para `p1` y el valor de `f1` se copia en ella.
- La línea 15 cambia el valor de `p1` a `B`.
- La línea 16 emite la sentencia `stop`. Se libera la memoria para `p1`, y el valor cambiado se pierde. Transfiere el control a la línea 7.
- La línea 8 escribe `B`.

El último tema que veremos con respecto a subrutinas son las llamadas subrutinas externas.

Una *subrutina externa* es la que reside en un programa diferente al que `perform` llama. La figura ilustra una subrutina externa:



Cuando `perform` realiza una llamada de subrutina externa:

- El programa externo que contiene la subrutina se carga.
- El programa externo entero se comprueba su sintaxis.
- Se transfiere el control a la sentencia `form` en el programa externo.
- Las declaraciones dentro de la subrutina externa se ejecutan.
- `endform` transfiere el control de nuevo a la instrucción que sigue a `perform`.

El hecho de que la comprobación de sintaxis se produce en tiempo de ejecución es importante, por las dos razones siguientes:

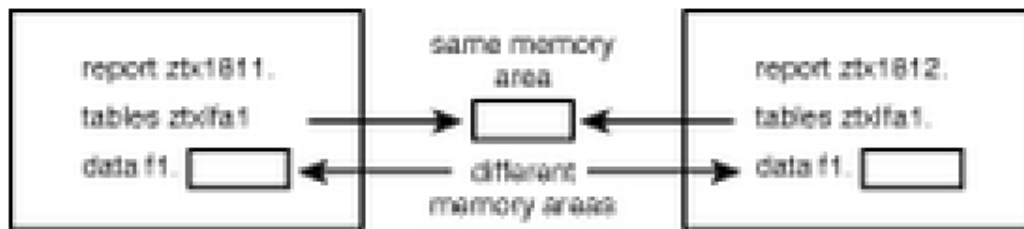
- Si un parámetro formal y un parámetro real no coinciden, se provoca un error en tiempo de ejecución en lugar de un error de sintaxis.

- Un error de sintaxis en cualquier parte del programa externo produce un error de tiempo de ejecución, ya sea dentro o fuera de la subrutina externa.

Las subrutinas externas son muy similares a las subrutinas internas:

- Ambos permiten que los parámetros se puedan pasar.
- Ambos permiten a los parámetros formales tipados.
- Ambos permiten que los parámetros se pasen por valor, por valor y resultado y por referencia.
- Ambos permiten la definición de variables locales.

La figura ilustra las diferencias entre subrutinas internas y externas:



Las siguientes son las diferencias entre las subrutinas externas e internas:

- Una variable global definida por el uso de `data` sólo se conoce dentro del programa que lo define. Por ejemplo, en la figura anterior la declaración `data f1` aparece en ambos programas. Esto define dos áreas de memoria con nombre `f1`. El `f1` definido dentro `ztx1811` sólo es accesible desde dentro `ztx1811`. El `f1` definido dentro `ztx1812` sólo es accesible desde dentro `ztx1812`. Todas las referencias a `f1` dentro `ztx1812` son las referencias a la `f1` definido dentro `ztx1812`.
- Una variable global con el mismo nombre en ambos programas y que se define utilizando la declaración `tables` en ambos programas es común a ambos programas. Un cambio a esta variable en un programa afecta a la otra. En la Figura anterior, el área de memoria llamada `ztxlfa1` se comparte entre ambos programas. Cualquier cambio en esa área de trabajo es visto inmediatamente por ambos programas.

Los dos siguientes listados ilustran una llamada a un subprograma externo.

```
1 report ztx12.
2 tables spfli.
3 data f1(3) value 'AAA'.
4
5 spfli-carrid = 'AZ'.
6 perform s1(ztx13).
7 write: / 'f1 =', f1,
8         / 'Compañía =', spfli-carrid.
```

```
1 report ztx1813.
2 tables spfli.
3 data f1(3).
4
5 form s1.
6     f1 = 'ZZZ'.
7     spfli-carrid = 'AA'.
8     endform.
```

El código del primero de los dos listados anteriores genera el siguiente resultado:

```
f1 = AAA
Compañía = AA
```

El análisis del código es éste:

- En ambos programas, la línea 2 define un área de trabajo común `spfli`. Esta área de trabajo se reparte entre los dos programas.
- En ambos programas, la línea 3 define una variable global `f1`. Dos áreas de memoria independientes se definen, uno para cada programa.
- En `ztx12`, la línea 5 asigna un valor de **AZ** a `spfli-carrid`.
- En `ztx12`, la línea 6 pasa el control a la línea 5 en `ztx13`.
- En `ztx13`, la línea 6 asigna **ZZZ** a la variable global `f1` para `ztx13`. `f1` en `ztx12` no se ve afectada.
- En `ztx13`, la línea 7 asigna **AA** a `carrid`. Esto afecta a los dos programas.
- En `ztx13`, la línea 8 devuelve el control a la línea 6 en `ztx12`.
- En `ztx12`, las líneas 7 y 8 escriben los valores de `f1` y `spfli-carrid`. `f1` no se ha modificado, y `carrid` ha cambiado.

Macros

La última unidad de modularización de ABAP/4 que veremos en la presente unidad, son las macros.

Muchas de las declaraciones ya vistas requieren un montón de codificación repetitiva. Se puede reducir la carga de trabajo utilizando macros. Una *macro* es un bloque de sentencias con un nombre que contiene varios comodines. Las macros son llamadas en los programas, y se le pasan parámetros actuales que reemplazan a los comodines, que funcionan como parámetros formales. Al igual que sucede con los *includes*, cuando el programa se ejecuta, la sentencia de definición de la macro en el programa es reemplazada por su código asociado. Es conveniente definir toda mi librería de macros en un determinado *include*, y luego incluir este en el programa deseado, de modo tal de tener disponible así todas las macros para usar en el mismo.

Veamos la sintaxis de definición de macro:

```
DEFINE macro.  
* sentencias  
END-OF-DEFINITION.
```

El nombre de la macro no necesariamente debe empezar con **y** o con **z**. Además, en las sentencias, se pueden usar los comodines.

Veamos un ejemplo:

```
DEFINE mi_macro.  
write &1.  
perform &2.  
END-OF-DEFINITION.
```

En el ejemplo anterior, yo he definido una macro que puedo guardar en mi *include* de macros, y lo que hace es imprimir algo y luego llamar a una subrutina (que suponemos preexistente). Este ejemplo es muy sencillo, pero podemos construir macros realmente complejas y muy útiles.

Para ver como funcionaría esta macro, supongamos que la guardamos en el *include* Zmacros (el nombre del *include* sí debe empezar con **Z** o con **Y**). Mi programa quedaría así:


```
report zmacros.  
include Zmacros.  
* la sentencia anterior me pone las macros disponibles en el programa.  
* si le hago doble clic, vería el código de definición de la macro  
* cuyo nombre es mi_macro, lo cual se mostró antes.  
* ahora la voy a invocar con distintos parámetros:  
mi_macro 'Hola' sub1.  
* 'Hola' reemplaza a &1 en mi_macro  
* sub1 reemplaza a &2 en mi_macro  
* esta sentencia llama a mi_macro e imprime el primer parámetro  
* e invoca la subrutina cuyo nombre es el segundo parámetro.  
mi_macro 'Chau' sub2.  
* vuelve a invocar la macro con otros parámetros  
* y así siguiendo, tantas veces como la quiera invocar  
* con distintos pares de parámetros.
```

Formateando un listado

ABAP/4 tiene una serie de instrucciones especialmente diseñadas para que la generación de reports sea más sencilla.

Formato de los datos de salida

Para visualizar un valor utilizaremos la sentencia WRITE:

WRITE /<offset>(<long>) '<datos a visualizar>'.
Universidad Tecnológica Nacional - Derechos Reservados

donde:

/ :Indica si queremos saltar una línea o no antes de empezar a imprimir en pantalla (opcional).

Offset: Indica la columna donde empezará la impresión (opcional).

Long: Indica la longitud de los valores a visualizar (opcional).

Ejemplo:

Código:

```
data: nombre(10) type c value 'Maribel',  
      edad(2) type n value '25',  
      telef (20) type c value '91-746-62-26',  
      prom type p decimals 2 value '8.75'.
```

write:/10 'NOMBRE:', nombre, 20 'EDAD', edad.
write:/10 'TELEFONO', (9)telf+3.
write:/10 'PROMEDIO', prom.

Salida:

NOMBRE: Maribel EDAD: 25
TELEFONO: 746-62-26
PROMEDIO: 8.75

Comandos Adicionales a WRITE:

WRITE v1 [opción(es)]

- (1) **under v2 | no-gap**
- (2) **using edit mask m | using no edit mask**
- (3) **mm/dd/yy | dd/mm/yy**
- (4) **mm/dd/yyyy | dd/mm/yyyy**
- (5) **mmddyy | ddmmyy | yymmdd**
- (6) **no-zero**
- (7) **no-sign**
- (8) **decimals n**
- (9) **round n**
- (10) **currency c | unit u**
- (11) **left-justified | centered | right-justified**

donde:

v1 es una literal, variable o el nombre de algún campo.

m es una máscara.

c es una moneda.

u es una unidad.

n es un literal numérico o una variable.

Las siguientes opciones se refieren a:

Para cualquier tipo de dato:

left-justified Salida justificada a la izquierda.

Centered Salida centrada.

right-justified Salida justificada a la derecha.

under v2 Salida que empieza directamente bajo el campo v2.

no-gap El blanco después del campo v1 es omitido.

no-zero Si el campo contiene únicamente ceros, son remplazados por blancos.

Para campo numérico:

no-sign El signo no aparece en la salida.

decimals n Donde n define el número de dígitos después del punto decimal.

round n En el tipo P el campo es multiplicado por $10^{*(-n)}$ y es redondeado.

exponent e En el campo F, el exponente es definido por e.

Currency m Para visualizar importes correctamente dependiendo del importe, donde m es la moneda

Para campo de fechas:

mm/dd/yy Fecha con separadores (mes/día/año) y año de 2 dígitos.

dd/mm/yy Fecha con separadores (día/mes/año) y año de 2 dígitos.

mm/dd/yyyy Fecha con separadores (mes/día/año) y año de 4 dígitos.

dd/mm/yyyy Fecha con separadores (día/mes/año) y año de 4 dígitos.

mmddy Fecha sin separadores (mes/día/año) y año de 2 dígitos.

ddmmyy Fecha sin separadores (día/mes/año) y año de 2 dígitos.

yyymmdd Fecha sin separadores (año/mes/día) y año de 2 dígitos.

using no edit mask Desactiva la opción del uso de máscaras para los datos de salida.

edit mask puede servir para:

Insertar caracteres hacia la salida.

Mover signos al principio de un campo numérico.

Insertando artificialmente o moviendo un punto decimal.

Desplegando un número punto-flotante sin usar notación científica.

En la sentencia edit mask el guión bajo (_) tiene un significado especial. Al principio de un edit mask, V, LL, RR y == también tienen un significado especial. Todos los demás caracteres son transferidos sin cambio a la salida.

El guión bajo en un edit mask es remplazado uno a uno con el carácter de un campo (variable). El número de caracteres se toma del campo que será igual al número de guiones bajos en el edit mask. Por ejemplo si son 3 caracteres en el edit mask, a lo más 3 caracteres del campo (variable) serán de salida.

Por ejemplo, la sentencia:

write (6) 'ABCD' using edit mask ' _: _: _'.

Produce como salida:

A:BC:D

Los caracteres son tomados del campo uno a la vez, y los caracteres desde el edit mask son insertados por la máscara (mask).

La sentencia write 'ABCD' using edit mask ' _: _: _' únicamente escribe A:BC porque por defecto la longitud de la salida es igual a la longitud del campo (4) debido a que va contando uno a uno los caracteres del campo y los de la máscara.

Si hay algunos guiones bajos más que en el campo, por defecto toma los n caracteres más a la izquierda del campo, donde n es igual al número de guiones bajos en el edit mask.

Esto se puede especificar explícitamente precediendo a edit mask con LL. Por ejemplo:

WRITE 'ABCD' using edit mask 'LL__:_'

toma los 3 caracteres más a la izquierda y escribe

AB:C.

Usando RR toma los 3 caracteres más a la derecha, por lo que:

WRITE 'ABCD' using edit mask 'RR__:_'

Escribirá:

BC:D.

Si son más los guiones que los caracteres del campo el efecto de LL es justificar a la izquierda el valor de la salida, RR justificará el valor a la derecha.

Cuando un edit mask comienza con V, cuando es aplicado a un campo numérico (tipo I,P y F) causa que el signo sea desplegado al comienzo si se aplica a un campo de tipo carácter.

Ejemplo:

Código:

```
data: f1(4) value 'ABCD',  
f2(5) value '1234-'.  
Write:/ '1. ', f2,  
'2. ', f2 using edit mask 'LLV____',  
'3. ', f2 using edit mask 'RRV____',  
'4. ', f2 using edit mask 'RRV_____',  
'5. ', f1 using edit mask 'V_____'.  
.
```

Salida:

1. 1234-
2. -1234
3. - 123
4. -123,4
5. VABC

Una conversión de salida es una llamada de subrutina que formatea la salida.
Un edit mask que comience con == seguido por cuatro caracteres ID llama

una función que formatea la salida. Esos 4 caracteres ID es conocido como una conversión de salida o conversión de rutina.

El nombre de la función puede ser `CONVERSION_EXIT_XXXX_OUTPUT`, donde XXXX son los 4 caracteres ID que siguen a `==`. Por ejemplo `WRITE '00001000' using edit mask '==ALPHA'` llama a la función `CONVERSION_EXIT_ALPHA_OUTPUT`. La sentencia `write` pasa el valor primero a la función, la cual cambiará algunas cosas y regresará el valor y este será escrito en la salida.

Cuando utilizamos la instrucción `WRITE` con números empaquetados, el sistema trunca por la izquierda en caso de ser necesario (deja un * como indicador de que ha truncado) y rellena con blancos si sobra espacio. Tenemos que tener en cuenta que si es negativo el signo ocupará una posición. Si se especifican los decimales con la cláusula `DECIMALS` del `DATA`, el punto o coma decimal también ocupará una posición. El signo decimal (punto o coma) estará determinado por los valores del registro de usuario.

Ejemplo:

```
DATA NUMERO TYPE P DECIMALS 2 VALUE -123456.  
WRITE NUMERO.
```

1.234,56-

y si no cabe el número:

```
WRITE (6) NUMERO.
```

*4,56-

Podemos imprimir una línea de horizontal con la sentencia `ULINE`. Tendrá las mismas propiedades que el `WRITE`. Nota: con algunos monitores, la línea no se ve.

ULINE /(<offset>)(<long>).
o **WRITE** /(<offset>)(<long>) **SY-ULINE**.

Para imprimir una línea vertical se necesita la sentencia `WRITE`.

WRITE /(<offset>) (<long>) **SY-VLINE**.

Para saltar una o varias líneas utilizaremos **SKIP n**, en donde *n* es el número de líneas en blanco.

Podemos modificar los atributos de pantalla para un campo.

FORMAT INTENSIFIED ON/OFF. Afecta el color del background.

FORMAT INVERSE OFF/ON. Afecta el color del background y foreground.

FORMAT INPUT OFF/ON. Indica cuando el usuario puede introducir datos.

FORMAT COLOR n. Color de la línea background, y *n* puede tomar los valores de la tabla siguiente:

<i>N</i>	<i>Código</i>	<i>Color</i>	<i>Intensidad para</i>
Off COLBACKGROUND	0	Depende del GUI	Background
1 ó COL_HEADING	1	Gris-Azul	Cabeceras
2 ó COL_NORMAL	2	Gris claro	Cuerpo de las listas
3 ó COL_TOTAL	3	Amarillo	Totales
4 ó COL_KEY	4	Azul-Verde	Columnas llaves
5 ó COL_POSITIVE	5	Verde	Valor positivo
6 ó COL_NEGATIVE	6	Rojo	Valor negativo
7 ó COL_GROUP	7	Violeta	Niveles de grupos

FORMAT RESET. Restablece todos los formatos anteriores.

Formato de página

También hay un grupo de instrucciones destinadas a dar formato a la salida del report, ya sea por pantalla o por impresora.

Podemos hacer tratamientos por inicio y fin de página con los eventos :

TOP-OF-PAGE y END-OF-PAGE.

Para cambiar la cabecera de una página individualmente se puede usar TOP-OF-PAGE.

Este evento ocurre tan pronto como el sistema empieza a procesar una nueva página. El sistema procesa la sentencia siguiente de TOP-OF-PAGE antes de la salida de la primera línea de la nueva página.

Estos eventos son independientes, es decir, se puede usar uno y el otro no, o ambos según sea necesario.

Hay que tener en cuenta (lo veremos en detalle en la próxima unidad) que para finalizar el bloque de procesos siguientes a TOP-OF-PAGE se puede usar START-OF-SELECTION.

END-OF-PAGE no se ejecutará si el salto de página se produce con un NEW-PAGE.

Si no queremos que la cabecera del report sea la estándar de SAP, ya que la queremos controlar nosotros directamente en el evento TOP-OF-PAGE, utilizaremos:

REPORT <Zxxxxxxx> NO STANDARD PAGE HEADING.

El formato de la página de report se define también desde la instrucción **REPORT**:

REPORT <Zxxxxxxx> **LINE-SIZE** <n> Ancho de línea.

LINE-COUNT <n(m)> Líneas por página (n). Si se desea se pueden reservar líneas para un pie de página (m):

PAGE-COUNT <n>. No. máximo de páginas.

Podemos impedir que con un salto de página se corten líneas que pertenezcan a una agrupación de líneas con significado lógico propio. Con la instrucción **RESERVE** reservamos un número de líneas:

RESERVE <n> **LINES**.

Esta instrucción se colocará justo antes del write que se quiere ‘reservar’, si no cabe se imprimirá en la siguiente página.

Hay varias formas de imprimir un report:

- Una vez ha salido el report por pantalla con la opción de ‘Imprimir’.
- Imprimir sin visualizar por pantalla con la opción ‘Imprimir’ desde la pantalla de selección o de parámetros.

Desde el programa ABAP/4 podemos controlar la impresión con la instrucción **NEW-PAGE**:

NEW-PAGE PRINT ON/OFF Impresora o pantalla.

NO DIALOG No visualiza la pantalla de opciones de impresión.

LINE-COUNT <n> Líneas por página.

LINE-SIZE <n> Tamaño de línea.

DESTINATION <des> Impresora destino.

IMMEDIATELY <’X’>. Impresión inmediata S/N.

Selección de parámetros

Si deseamos introducir una serie de delimitaciones en la ejecución de un report a nivel de parámetros, dispondremos de dos posibilidades:

1. PARAMETERS que permite utilizar parámetros de cualquier tipo en la pantalla de selección:

PARAMETERS: p <opción>

Opciones:

- (1) DEFAULT.- Le asigna un valor al parámetro.
- (2) TYPE typ.- Asigna el tipo typ a un campo interno.

Ejemplo:

parameters number(4) type p default '999'.

- (3) DECIMALS dec.- dec determina los número de lugares decimales en el campo. dec debe de ser numérico.

Ejemplo:

parameters number(4) type p decimals 2 default '123.45'.

- (4) LIKE g.- Crea un campo p con los mismos atributos que un campo g, el cual ya fue definido. g puede ser un campo de una base de datos o una campo interno existente.

Ejemplo:

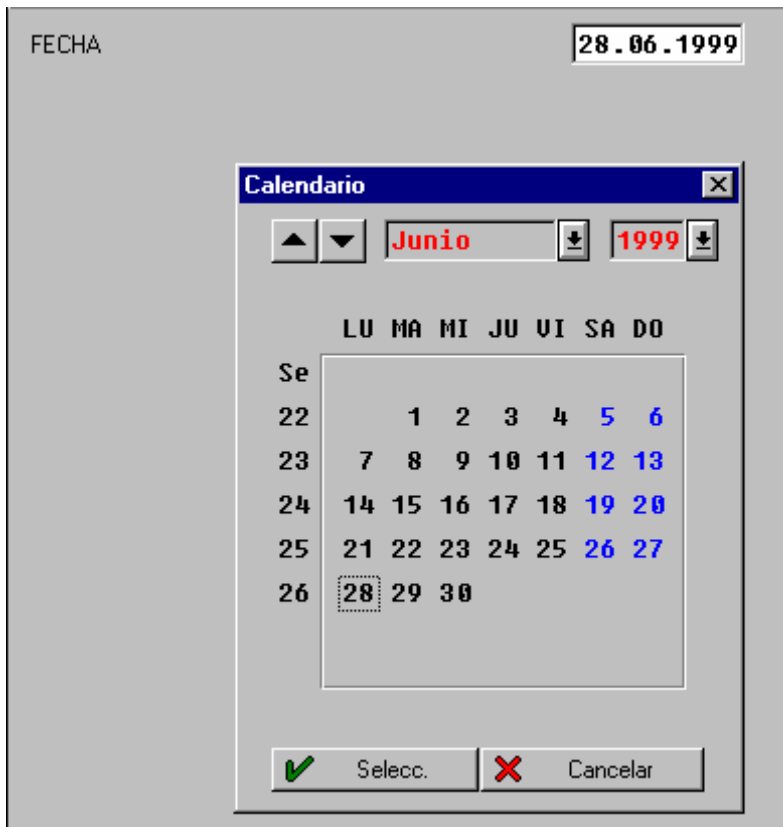
parameters fecha like sy-datum.

Salida:

The image shows a screenshot of an SAP selection screen. It features a light gray rectangular area. On the left side of this area, the word 'FECHA' is displayed in a dark, sans-serif font. To the right of the text, there is a small white rectangular input field. To the right of the input field is a small square button containing a downward-pointing arrow, indicating a dropdown menu.

En la pantalla de salida observaremos un campo como éste, el cual toma las características de la variable sy-datum de la estructura syst, por lo tanto aparece una flecha en el campo, lo cual nos indica que ese nuevo campo tiene asociado diferentes datos, si presionamos, la flecha obtendremos la siguiente

pantalla, en donde podemos escoger las fecha que deseemos con solo colocar el cursor en el día y hacer un doble clic y automáticamente aparecerá la fecha en el campo de FECHA:



- (5) LOWER CASE.- Permite introducir minúsculas.
- (6) OBLIGATORY.- Obliga a introducir un valor.
- (7) AS CHECKBOX.- Despliega un parámetro como un checkbox. Si no se especifica el tipo o la longitud cuando se define el parámetro, este será de tipo C y de longitud 1.
El checkbox es desplegado hacia la izquierda y hacia la derecha el texto. Para definir algún orden , se debe usar selection-screen.
- (8) RADIOBUTTON GROUP *radi* Despliega el parámetro en el selection-screen como un radio button. Todos los parámetros asignados en esta forma para el mismo grupo *radi* (el cual puede estar definido con 4 caracteres de longitud).

2. - SELECT-OPTIONS permite determinar un criterio de selección de los datos a utilizar en el report:

SELECT-OPTIONS <var> FOR <campo_tabla>.

<var> como mucho tendrá 8 caracteres.

La variable <var> tomará los posibles valores a seleccionar y <campo_tabla> nos indica para qué campo y de qué tabla será utilizado el parámetro (esto implícitamente nos está dando el tipo y la longitud de los posibles valores). Con esta sentencia, automáticamente en la pantalla de selección se podrán introducir rangos de valores posibles para el parámetro.

Ejemplo :

Para cada sentencia SELECT-OPTIONS, el sistema crea una tabla interna con el nombre de <var>. Cada registro de la tabla está formado por los campos : <var>-LOW, <var>-HIGH, <var>-SIGN, <var>-OPTION.

El contenido de cada registro será respectivamente: el valor inferior, el superior, el signo (Incluido/Excluido) y el operador.

En la pantalla de selección, si queremos realizar una selección compuesta de más de una condición (más de un registro en la tabla interna), tendremos que hacer Clic sobre la flecha situada a la derecha de cada campo.

Para seleccionar los datos de lectura en tiempo de ejecución mediante los valores de selección, utilizaremos la cláusula WHERE de la instrucción

SELECT y el operador IN, que buscará en la tabla de base de datos todos los registros que cumplan las condiciones incluidas en la tabla interna de la pantalla de selección.

SELECT-OPTIONS <var> FOR <campo>.

...

SELECT * FROM <tab> WHERE <campo> IN <var>.

En la pantalla de selección aparecerá el texto <var> como comentario a la selección de datos, si queremos que el texto sea distinto al nombre de la variable tendremos que ir a la opción *Textos de selección* del menú *Pasar a -> Elementos de Texto*.

Veamos ahora qué otras opciones existen en la utilización de la instrucción **SELECT-OPTIONS**.

Para asignar valores iniciales a un criterio de selección utilizamos la cláusula **DEFAULT**:

SELECT-OPTIONS <var> FOR <campo> DEFAULT '<valor>'.

Si queremos inicializar un rango de valores (inferior y superior) usaremos:

SELECT-OPTIONS <var> FOR <campo> DEFAULT '<ini>' TO '<fin>'.

Podemos hacer que se acepten valores en minúsculas:

SELECT-OPTIONS <var> FOR <campo> LOWER CASE.

Podemos obligar a que se introduzcan valores de selección inevitablemente:

SELECT-OPTIONS <var> FOR <campo> OBLIGATORY.

También es posible desactivar la posibilidad de introducir selecciones con condiciones compuestas. (Desaparecerá la flecha):

SELECT-OPTIONS <var> FOR <campo> NO-EXTENSION.

Otra opción es que no salga un intervalo, sino una sola caja. Es parecido a un parameter, con la diferencia que, si no se rellena un parameter, el valor es ' ', mientras, que, si no rellena un select-options, el valor es todo el rango de valores posibles:

SELECT-OPTIONS <var> FOR <campo> NO INTERVALS.

Pantalla de selección (selection-screen)

También es posible formatear a nuestro gusto la pantalla de selección con **SELECTION-SCREEN** utilizando las siguientes opciones:

SELECTION-SCREEN BEGIN OF LINE.

...

SELECTION-SCREEN END OF LINE.

Permite la combinación de varios parámetros y comentarios especificados entre las sentencias *selection-screen begin of line* y *selection-screen end of line* y sus salidas en una línea. Como resultado, no hay automáticamente línea nueva para cada parameters y los textos de selección se despliegan.

Ejemplo:

Código

```
selection-screen begin of line.  
selection-screen comment 1(10) text-001.  
parameters: p1(3), p2(5), p3(1).  
selection-screen end of line.  
Textos  
Text-001 'Comment'  
Salida  
Comment ____
```

Nota: No se puede colocar select-options entre selection-screen begin of line y selection-screen end of line porque se generan varios objetos en la selección de pantalla para un select-options (por ejemplo los campos de más bajo y más alto límites del rango).

SELECTION-SCREEN SKIP n

Genera n líneas en blanco. Se puede especificar el valor para n entre 1 y 9. Si se quiere que solo deje una línea en blanco se puede omitir n.

SELECTION-SCREEN ULINE <col> (<long>)

Genera una línea horizontal, se le puede especificar la posición en la que va a empezar con el parámetro col, y su longitud con el parámetro long.

SELECTION-SCREEN POSITION pos

La posición de la salida de los parámetros es proporcionada por la variable pos, en caso de que no se coloque esta instrucción, la posición de los parámetros es cero 0.

SELECTION-SCREEN COMMENT <col> TEXT-nnn.

Introduce comentarios para un parámetro, en donde TEXT-nnn, será el text symbol que contenga el comentario que se desea introducir, y con col se posiciona el comentario.

SELECTION-SCREEN BEGIN OF BLOCK block <WITH FRAME><TITLE>.

Comienza un bloque lógico en la pantalla de selección, en donde block es el nombre de dicho bloque. Si se le coloca la opción WITH FRAME, se genera un marco alrededor del bloque. La opción TITLE coloca el nombre de ese bloque, se puede utilizar solamente si está también la opción FRAME.

SELECTION-SCREEN END OF BLOCK block.

Cierra el bloque abierto por:

SELECTION-SCREEN BEGIN OF BLOCK *block*. Si el bloque tiene un frame o marco, el marco se cierra también con esta instrucción.

Ejemplo:

(Text-001 contiene 'BLOCK CHARLY')

Código:

tables saplane.

selection-screen begin of block charly with frame title text-001.

parameters parm(5).

select-option sel for saplane-planetype.

selection-screen end of block charly.

Salida:

BLOCK CHARLY

PARM _____

SEL _____ to _____

SELECTION-SCREEN NEW-PAGE.

Sirve para utilizar varias páginas de selección.

A continuación se muestra un ejemplo, en el cual se observa el uso de varias de las sentencias anteriores.

Código:

report zlb40.

tables: zclient.

selection-screen skip 2.

selection-screen begin of block caja1 with frame title text-001.

parameters: clientes as checkbox,

distrib as checkbox,

personal as checkbox, todos as checkbox.

selection-screen end of block caja1.

selection-screen uline.

selection-screen begin of block caja2 with frame title text-002.

parameters: hombres radiobutton group rad1,

mujeres radiobutton group rad1,

ambos radiobutton group rad1.

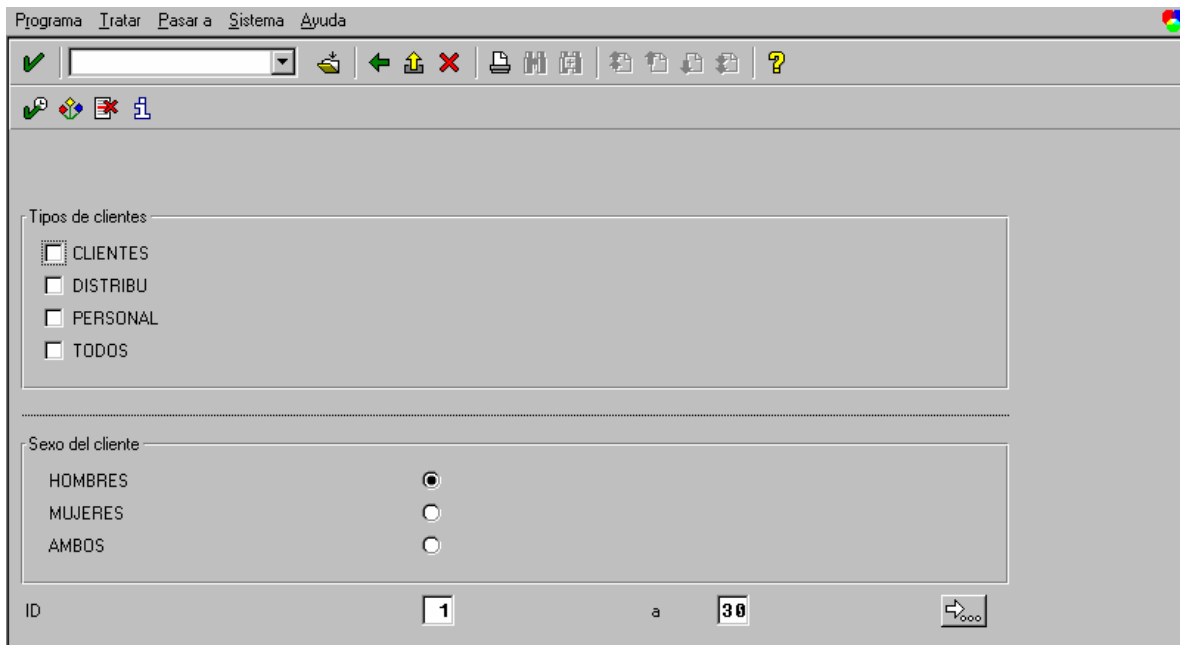
selection-screen end of block caja2.

selection-screen begin of block caja3.

select-options id for zclient-cla_client default '1' to '30'

selection-screen end of block caja3.

Salida:



Una instrucción que nos permite verificar los datos de entrada en la pantalla de selección es la siguiente:

AT SELECTION-SCREEN ON <campo>.

...

ENDAT.

Elementos de texto y mensajes

El entorno de desarrollo de programas en ABAP/4 nos permite manejar elementos de texto sin necesidad de codificarlos en el programa.

Los elementos de texto pueden ser títulos de reports, cabeceras de reports, textos de selección y textos numerados.

Podemos acceder a la pantalla de tratamiento de los elementos de textos desde el editor de programas: Pasar a -> Elementos de texto.

Con los Títulos y Cabeceras podemos tratar el título, cabeceras de report y cabeceras de columna que saldrán por pantalla e impresora.

Con los Textos de selección trataremos los comentarios que acompañan a los parámetros del tipo PARAMETERS o SELECT-OPTIONS.

Con los Textos numerados podemos utilizar constantes de tipo texto sin necesidad de declararlas en el código del programa. Los nombres de las constantes serán TEXT-xxx, donde xxx son tres caracteres cualquiera.

Además podemos mantener los textos numerados en varios idiomas.

Otras de las facilidades que nos ofrece ABAP/4 para el formateo y control de reports, es la de los mensajes de diálogo. Los mensajes de diálogo son aquellos mensajes que aparecen en la línea de mensajes y que son manejables desde un programa.

Los mensajes están agrupados en áreas de mensajes. Para indicar que área de mensajes vamos a utilizar en un report utilizamos MESSAGE-ID en la instrucción REPORT.

REPORT <report> MESSAGE-ID <área>.

Podemos ver, crear y modificar áreas de mensajes desde el editor : *Pasar a -> Mensajes.*

Para visualizar un mensaje utilizamos la sentencia MESSAGE.

MESSAGE Tnnn.

Donde nnn es el número de mensaje dentro de su respectiva área de mensajes y T es el tipo de mensaje:

A = Cancelación o 'Abend' del proceso.

E = Error. Es necesaria una corrección de los datos.

I = Información. Mensaje meramente informativo. El proceso continuará con un ENTER.

S = Confirmación. Información en la pantalla siguiente.

W = Warning. Nos da un aviso. Podemos cambiar los datos o pulsar 'intro' para continuar.

X = Exit. Transacción terminada con un short dump.

Ejemplo:

Código

```
report zlb401.  
parameter: num type i.  
if num > 10.  
message id 'Z1' type 'E' number '001'.  
endif.
```

Salida



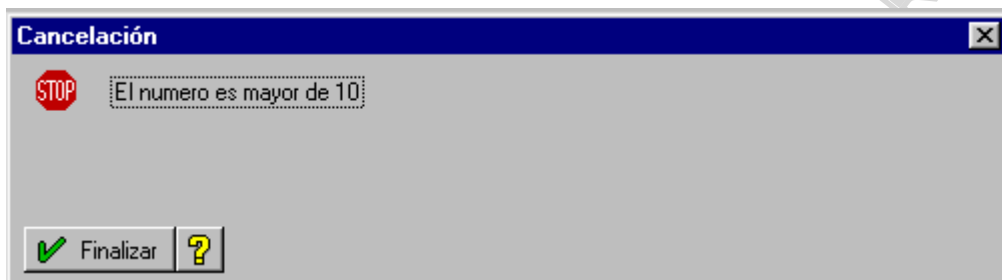
The screenshot shows a blue message box with the text 'E: El numero es mayor de 10' on the left. On the right, there are three fields: 'VMD (1) (002)', 'MSMADO', and 'IN'.

Este resultado se observa en la Barra de Estado, cuando se introduce un número mayor de 10.

Código:

```
report zlb401.  
parameter: num type i.  
if num > 10.  
message id 'Z1' type 'A' number '001'.  
endif.
```

Salida:

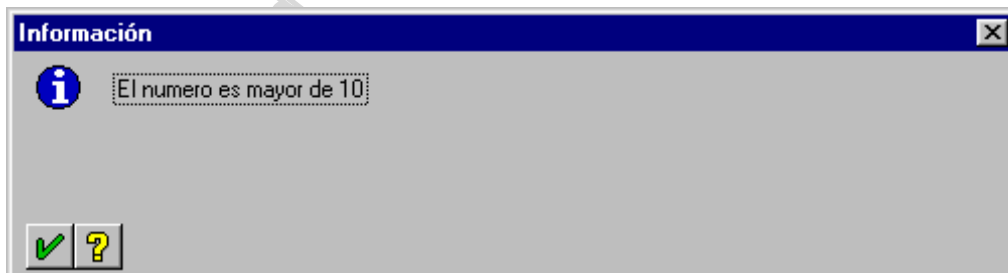


Este mensaje se observa en la pantalla, al presionar la tecla de EXIT, el sistema regresa a la pantalla principal del área de Desarrollo.

Código:

```
report zlb401.  
parameter: num type i.  
if num > 10.  
message id 'Z1' type 'I' number '001'.  
endif.
```

Salida:



Este mensaje nos permite intentar nuevamente a colocar un valor y ejecutar el programa.

Código:

```
report zlb401.
parameter: num type i.
if num > 10.
message id 'Z1' type 'S' number '001'.
endif.
```

Salida:

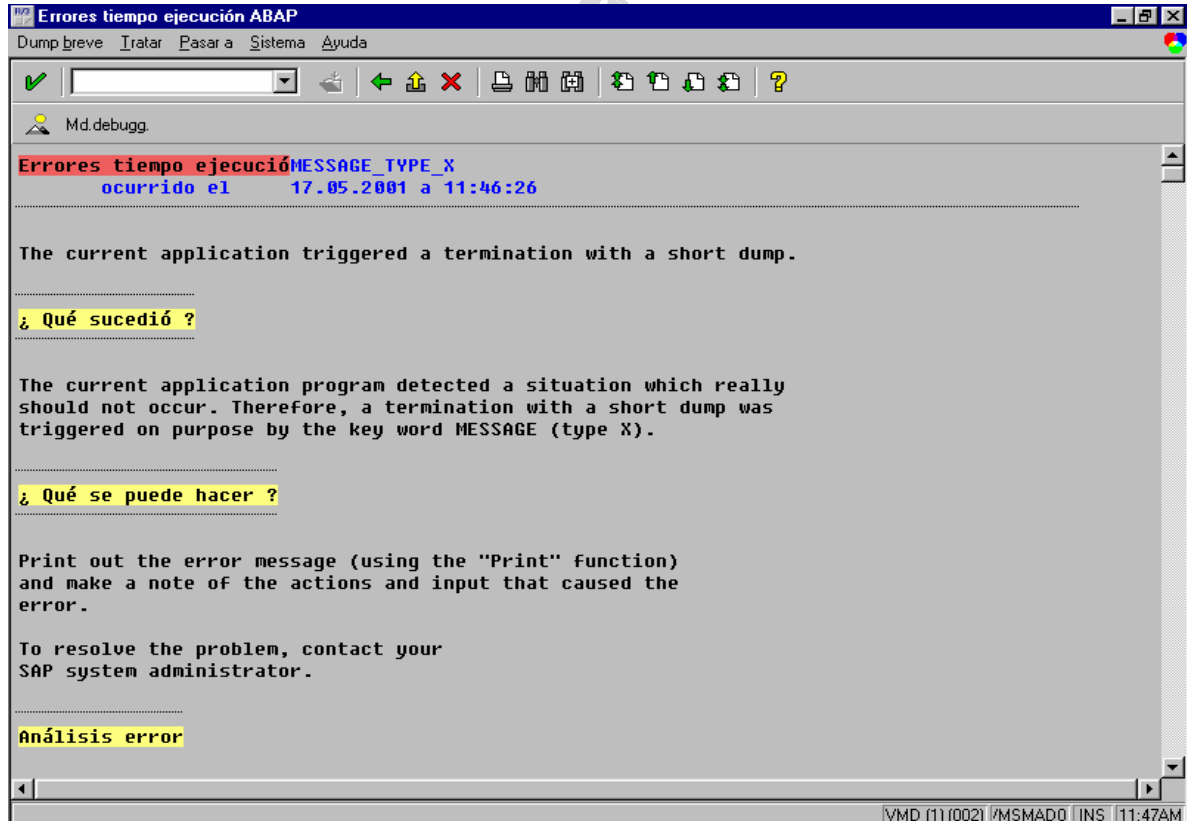


Este mensaje nos permite cambiar el valor del campo solicitado.

Código:

```
report zlb401.
parameter: num type i.
if num > 10.
message id 'Z1' type 'X' number '001'.
endif.
```

Salida:



Si se emiten mensajes del tipo W o E en eventos START-OF-SELECTION o END-OFSELECTION o GET se comportan como si fueran del tipo A. Podemos acompañar los mensajes de parámetros variables.

MESSAGE Tnnn WITH <var1> <var2> ...

En la posición del mensaje que se encuentre el símbolo & , podemos utilizar para visualizar el valor que le pasemos como parámetro a la instrucción MESSAGE.

No podemos utilizar más de 4 parámetros por mensaje.

Los datos sobre mensajes están en la tabla T100.

Ejemplo:

Área de mensajes ZZ.

Mensaje : 005 = Entrada &-& incorrecta.

```
REPORT ZPRUEBA MESSAGE-ID ZZ.  
IF ....  
MESSAGE A005 WITH SKA1 KTOPL.  
ENDIF.
```

El mensaje obtenido será :

A: Entrada SKA1-KTOPL Incorrecta

Para crear un área de mensaje, se tiene que realizar los siguientes pasos:

1. Dentro de una sesión de ABAP Workbench se debe seleccionar del menú la opción Desarrollo, posteriormente la opción Entorno de programación y finalmente Mensajes.
2. En la pantalla siguiente solo se debe colocar el nombre del área de mensaje y presionar el botón Crear.
3. La siguiente pantalla es en donde se introducirá el título del área de mensajes.
4. Finalmente se debe salvar:

Tablas internas

Si deseamos guardar una colección de registros de datos de la misma estructura en memoria sin necesidad de acceder a la base de datos y poder realizar operaciones diversas con este conjunto de información, utilizaremos las tablas internas.

Cómo declarar tablas internas

```
DATA: BEGIN OF <tabla> OCCURS <n>,  
<Def.Campo>,  
...  
END OF <tabla>.
```

Definiremos una tabla interna con n líneas en memoria, más una línea de cabecera o área de trabajo.

La cantidad de líneas que especifiquemos en el OCCURS no limita el tamaño de la tabla, sino la cantidad de registros que se guardan en memoria simultáneamente. Esto hace necesario

un especial cuidado al proponer el número de líneas ya que un OCCURS muy grande supone un gran gasto de recursos del sistema y un OCCURS pequeño un acceso muy lento, ya que necesita de un proceso de paginación.

El parámetro OCCURS n define cuántas entradas en la tabla son creadas inicialmente. El valor de n no es significativo excepto cuando se usa la instrucción “APPEND SORTED BY f”.

Esta instrucción inserta la nueva entrada en la tabla y la ordena en orden descendente de acuerdo al campo f. Cuando el número de entradas de la tabla alcanza el valor n que se ha declarado con el OCCURS, la última entrada será borrada si el valor de f de la nueva entrada es mayor. Sólo se puede ordenar por un campo.

Ejemplo:

```
DATA: BEGIN OF COMPANIES OCCURS 3,  
      NAME(10),  
      SALES TYPE I,  
      END OF COMPANIES.  
COMPANIES-NAME = 'big'.  
COMPANIES-SALES = 90.  
APPEND COMPANIES.  
COMPANIES-NAME = 'small'.  
COMPANIES-SALES = 10.  
APPEND COMPANIES.  
COMPANIES-NAME = 'too small'.  
COMPANIES-SALES = 5.  
APPEND COMPANIES.  
COMPANIES-NAME = 'middle'.  
COMPANIES-SALES = 50.  
APPEND COMPANIES SORTED BY SALES.  
CLEAR COMPANIES.
```

Ahora la tabla tiene tres entradas (=> OCCURS 3). La línea con la entrada ‘too small’ en el campo NAME será borrada ya que la entrada ‘middle’ tiene un valor más grande en el campo SALES. Esa entrada aparecerá en la tabla en segundo lugar (después de ‘big’ y antes de ‘small’).

NOTA: Hay que tener en cuenta que si se utiliza el APPEND con el parámetro SORT BY, el sistema cada vez recorrerá toda la tabla; así pues, a veces, será mejor llenar la tabla con APPEND y una vez se ha terminado de llenar, ordenarla en orden ascendente o descendente.

Llenado de una tabla interna

APPEND : Añade un registro a una tabla interna con los valores que tengamos en el área de trabajo, como se muestra en el ejemplo anterior:

APPEND <intab>.

NOTA: Después de realizar un APPEND debe ir un *CLEAR* <intab>.

COLLECT : Añade o suma la línea de cabecera. Sumará los campos de tipo P,F,I, si existe una línea en la tabla con campos idénticos (tipo C) a los del área de trabajo. Si no coincide con ninguno, lo que hará será añadir la línea de cabecera a la tabla:

COLLECT <intab>

El problema de esta instrucción es que es bastante lenta. Se puede sustituir por las instrucciones READ e INSERT o MODIFY.

Podemos llenar una tabla interna con el contenido de una tabla de base de datos. Siempre que la tabla interna tenga la misma estructura que la tabla de base de datos:

SELECT * FROM <tab> **INTO TABLE** <tabint>.

Ordenar una tabla interna

Para clasificar una tabla interna utilizamos SORT.

SORT <intab>.

Esta instrucción ordena la estructura de la tabla sin tener en cuenta los campos P,I,F. Para ordenar por el campo(s) que necesitemos (sea del tipo que sea) :

SORT <intab> **BY** <campo1><campo n>.

Si no se indica lo contrario la ordenación por defecto es ascendente, con las siguientes cláusulas se puede especificar:

SORT ... ASCENDING. o DESCENDING.

Ejemplo:

Si tenemos la siguiente tabla clientes que esta ordenada por clave:

Clave	Nombre	Número de Cuenta	Sucursal	Saldo
0001000	Ezequiel Nava	80194776-456	123	2,589.50
0001029	Ivonne Rodriguez	82325690-455	122	15,368.00
0001053	Sara Rosas	80000236-456	123	1,500.25
0001082	Alfredo Martínez	79268978-457	124	30,549.61
0001097	Rosalinda Trejo	85689782-457	124	21,987.30
0001256	Javier Solano	81569782-460	135	13,569.20
0001299	Pedro Gamboa	80000468-456	123	5,698.21
0001356	Marco Antonio Balderas	90265874-460	135	29,350.00
0001468	Jorge Villalon	78698014-457	124	62,165.50

Si la deseáramos por ordenar por sucursal, tendríamos que escribir la siguiente instrucción:

Sort clientes by sucursal.

Y obtendríamos la siguiente tabla:

Clave	Nombre	Número de Cuenta	Sucursal	Saldo
0001029	Ivonne Rodriguez	82325690-455	122	15,368.00
0001000	Ezequiel Nava	80194776-456	123	2,589.50
0001299	Pedro Gamboa	80000468-456	123	5,698.21
0001053	Sara Rosas	80000236-456	123	1,500.25
0001082	Alfredo Martínez	79268978-457	124	30,549.61
0001468	Jorge Villalon	78698014-457	124	62,165.50
0001097	Rosalinda Trejo	85689782-457	124	21,987.30
0001256	Javier Solano	81569782-460	135	13,569.20
0001356	Marco Antonio Balderas	90265874-460	135	29,350.00

Procesamiento de una tabla interna

Podemos recorrer una tabla interna con la instrucción LOOP ... ENDLOOP.

LOOP AT <intab> (WHERE <cond>).

**...
ENDLOOP.**

En cada iteración coloca la línea de la tabla que se está procesando en la línea de cabecera.

Podemos restringir el proceso de una tabla con una condición *WHERE*.

Si no existe ningún registro de la tabla que cumpla la condición especificada en la cláusula *WHERE*, la variable del sistema *SY-SUBRC* será distinta de 0. Dentro del *LOOP* la variable *SY-TABIX* contiene el índice de la entrada que está procesando en ese momento. También es posible hacer un :

```
LOOP AT <intab> FROM <inicio> TO <fin>.  
...  
ENDLOOP.
```

Donde <inicio> y <fin> son índices de la tabla interna.

Tratamiento de niveles de ruptura

En el tratamiento de un *LOOP* podemos utilizar sentencias de control de ruptura (conocidos como "cortes de control"):

AT FIRST... ENDAT Realiza las instrucciones que hay a continuación del *AT FIRST* para la . primera entrada de la tabla.

AT NEW <campo>... ENDAT Realiza las instrucciones que hay a continuación del *AT NEW* para cada inicio de nivel de ruptura.

AT LAST... ENDAT. Realiza las instrucciones que hay a continuación del *AT LAST* para la última entrada de la tabla.

AT END OF <campo>... ENDAT. Realiza las instrucciones que hay a continuación del *AT END* para cada final de nivel de ruptura.

Si utilizamos la instrucción *SUM* dentro de un *AT ... ENDAT* realizará la suma de todos los campos *P,I,F* de ese nivel de ruptura (para el cálculo de subtotales). El resultado lo encontraremos en el área de trabajo de la tabla. Será necesario que la tabla interna esté ordenada en el mismo orden que la utilización de los niveles de ruptura.

Ejemplo:

Considerando la tabla clientes, y queremos obtener la suma del saldo por sucursal, y saldo total de esos clientes no importando la sucursal, tendremos el siguiente código.

Código:

Loop at clientes_tab.

At new sucursal.

Write:/ 'Sucursal', clientes_tab-sucursal.

Endat.

Write:/ clientes_tab-clave, 15 clientes_tab-nombre, 25 clientes_tab-cuentas, 35
clientes_tab-saldo.

At end of sucursal.

Sum.

Write:/ 'Total de la sucursal:', 35 clientes_tab-saldo.

Uline.

Skip.

Endat.

At last.

Sum.

Write:/ 'Total de las sucursales:', 35 clientes_tab-saldo.

Endat.

Endloop.

Salida:

Sucursal: 122			
0001029	Ivonne Rodriguez	82325690-455	15,368.00
Total de la Sucursal:			15,368.00
Sucursal: 123			
0001000	Ezequiel Nava	80194776-456	2,589.50
0001299	Pedro Gamboa	80000468-456	5,698.21
0001053	Sara Rosas	80000236-456	1,500.25
Total de la Sucursal:			9,789.96
Sucursal: 124			
0001082	Alfredo Martinez	79268978-457	30,549.61
0001468	Jorge Villalon	78698014-457	62,165.50
0001097	Rosalinda Trejo	85689782-457	21,987.30
Total de la Sucursal:			114,702.41
Sucursal: 135			
0001256	Javier Solano	81569782-460	13,569.20
0001356	Marco Antonio Baldera	90265874-460	29,350.00
Total de la Sucursal:			42,919.20
Total de las Sucursales:			167,411.57

Lectura de entradas de una tabla

Podemos buscar un registro concreto en una tabla sin necesidad de recorrerla.

READ TABLE <intab>.

Para ello en primer lugar rellenaremos la línea de cabecera con la clave de búsqueda y luego haremos el READ.

El resultado de la búsqueda lo tendremos en SY-SUBRC:

- Si SY-SUBRC = 0 la búsqueda ha sido positiva.
- Si SY-SUBRC \neq 0 no ha encontrado el registro solicitado.

Existen otras extensiones a la instrucción READ que necesitarán que la tabla esté ordenada. Podemos buscar por clave con:

READ TABLE <intab> WITH KEY <clave>.

No necesita llenar la línea de cabecera. Buscará desde el inicio de la tabla que carácter a carácter coincida con la clave. Es posible una búsqueda aún más rápida con una búsqueda binaria:

READ TABLE <intab> WITH KEY <clave> BINARY SEARCH.

Una lectura directa de un registro de la tabla la podemos realizar con:

READ TABLE <intab> INDEX <num>.

Modificando tablas internas

Una vez llena la tabla interna tenemos la posibilidad de modificar los datos con una serie de sentencias ABAP/4.

MODIFY : Podemos sobrescribir el contenido de la entrada <i> con el contenido de la línea de cabecera.

MODIFY <intab> (INDEX <i>).

NOTA: No modifica solo un campo sino modifica todo el registro (renglón). Dentro de un LOOP, la cláusula INDEX es opcional. Por defecto será el contenido de la variable SY-TABIX.

INSERT : Añade una entrada delante de la entrada <i> con el contenido de la línea de cabecera.

INSERT <intab> (INDEX <i>).

DELETE : Para borrar una entrada de una tabla.

DELETE <intab> (INDEX <i>).

Otras instrucciones de manejo de tablas:

- Inicializar el área de trabajo o línea de cabecera:

CLEAR <intab>.

Borra los campos del renglón (limpia ese renglón).

- Inicializar (borrar) contenido de una tabla:

REFRESH <intab>.

Esta instrucción no borra la cabecera de la tabla, para ello es necesario utilizar la instrucción anterior (**CLEAR <intab>**).

- Liberar el espacio ocupado por una tabla en memoria:
FREE <intab>.
- Para obtener información sobre una tabla interna:
DESCRIBE TABLE <tab> LINES <contador_entradas> OCCURS <valor_occurs>.

Bases de datos lógicas

¿Qué es una base de datos lógica ?

Para obtener datos en un programa existen dos posibilidades:

- Programar la lectura de datos de la base de datos en el mismo programa con la instrucción **SELECT**.
- Dejar que otro programa de lectura (BD lógica) lea los datos y se los proporcione en la secuencia apropiada.

En un report se pueden simultanear los dos tipos de selección de datos.

Una base de datos lógica (LDB) proporciona una visión lógica de las tablas físicas, pudiendo relacionar tablas entre sí. Las LDB simplifican la programación de reports ofreciendo accesos de lectura, verificación de autorizaciones y selecciones estandarizadas.

La comunicación entre el programa de lectura y el report que utiliza la base de datos lógica se realiza mediante los eventos **PUT** y **GET**.

Por regla general utilizaremos bases de datos lógicas que ya existen en el sistema, aunque también es posible crear nuevas y modificarlas. (Transacción **ALDB**).

Si utilizamos LDB ya creadas en el sistema, únicamente tendremos que utilizar un evento para recoger la información que el programa de lectura (que ya existe) nos va dando.

Si por el contrario nos decidimos a crear una LDB con la transacción **ALDB**, el sistema generará todo lo necesario para utilizar la base de datos lógica, incluyendo el programa de lectura.

Utilización de las bases de datos lógicas

Las bases de datos lógicas tienen un nombre de tres caracteres, siendo el último carácter el módulo funcional al que va dirigido.

Ejemplo :

KDF : clientes FI

En el programa que va a utilizar bases de datos lógicas será necesario especificar en los atributos del programa la LDB que va a ser utilizada. Y en el código simplemente utilizaremos el evento GET.

GET <tablaBDD1>.

<sentencias evento>

.....

GET <tablaBDD2>.

<sentencias evento>

.....

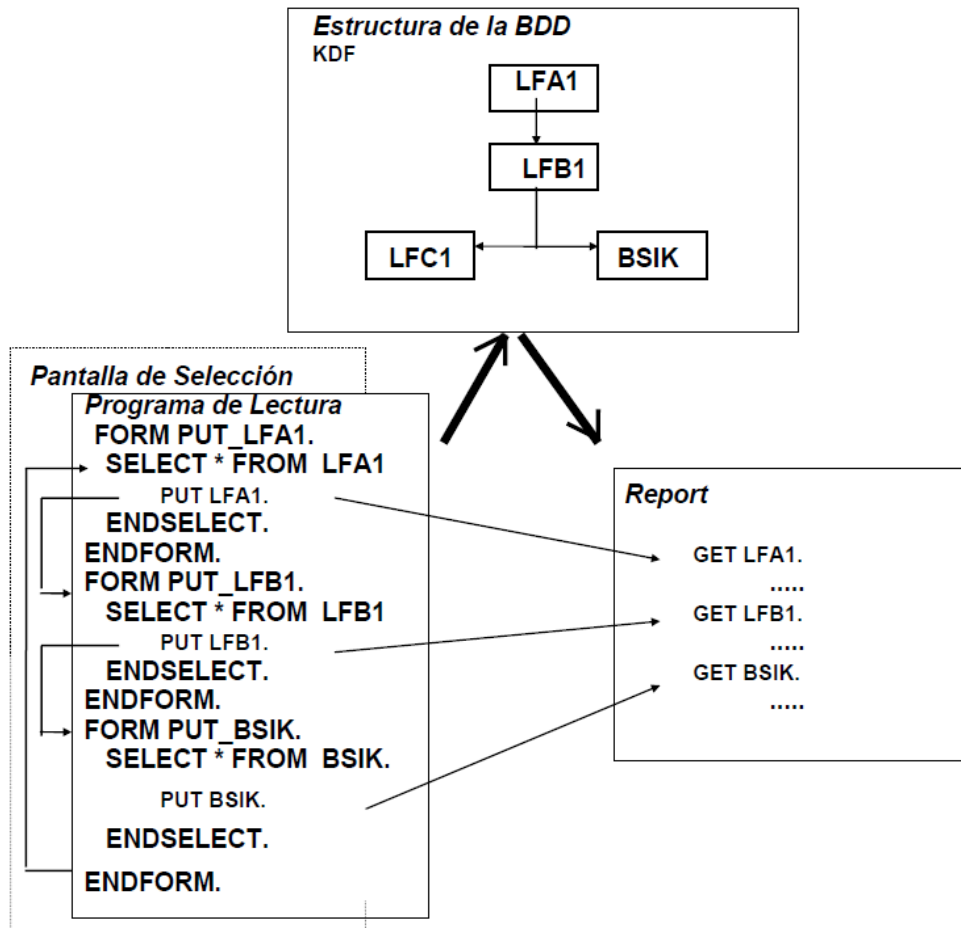
Mediante el GET dispondremos de un registro de la base de datos que especifiquemos, siempre y cuando esta tabla esté dentro de la estructura de la base de datos lógica.

Para comunicar el programa de lectura con nuestro report se utiliza el PUT, que suministra el registro de la BDD que especifiquemos, previamente habrá realizado el SELECT.

PUT <tablaBDD>.

Una base de datos lógica tiene tres componentes fundamentales:

- Una definición de la estructura de las tablas que utiliza.
- Una pantalla de selección de los datos a leer. (SELECT-OPTIONS)
- Un programa de lectura de datos de la BDD. (PUT).



También existe la posibilidad de utilizar el evento:
GET <tabBDD> LATE.

...

Este evento se produce cuando se han procesado todas las entradas de tablas subordinadas a un registro de datos de una tabla, y antes de que el sistema solicite la siguiente entrada de la misma tabla (mismo nivel jerárquico). Existe una instrucción de salto o finalización de lectura de una tabla, **REJECT**.

Esta instrucción sale del proceso del registro en curso y continúa con el proceso del siguiente registro dentro del mismo nivel de jerarquía.

Si indicamos un nombre de tabla, lo que hará será continuar con el siguiente registro de la tabla especificada. <tabla> no puede ser un nivel de jerarquía más profundo que el actual:

REJECT <tabla>.

No siempre se necesitan datos de todas las tablas de la estructura de la base de datos lógica. Por ejemplo podemos estar interesados solamente en datos de la tabla LFA1, con lo cual:

TABLES: LFA1.

.....

GET LFA1.

Pero si quisiéramos datos únicamente de la tabla BSIK, deberíamos declarar bajo **TABLES** todas las tablas que haya en el “path” hasta llegar a la que nos interesa. Así pues quedaría:

TABLES: LFA1, LFB1, BSIK.

.....

GET BSIK.

En principio, únicamente utilizaremos la sentencia GET , ya que utilizaremos LDBs que ya existen en el sistema.

Si necesitamos crear una nueva debido a que se han de desarrollar muchos reports con una estructura de lectura muy similar, y ésta no está en ninguna base de datos lógica, utilizaremos la transacción ALDB.

Field-groups

Ya vimos que cuando queremos ordenar y/o controlar las rupturas de campos en un report, es necesario utilizar las tablas internas. Sin embargo existe otra utilidad de ABAP/4 que nos facilita estos procesos de ordenación y rupturas, en el caso de que sean complejos.

Supongamos un listado en el que las líneas sean de muy distinto tipo, por ejemplo, un listado de proveedores con datos generales de éste, (dirección ...) y las ventas que nos han realizado cada uno de los proveedores, ordenados por distintos campos y con subtotales. En este caso no tendremos más remedio que utilizar diversas tablas internas, una para cada tipo de línea, ordenar estas tablas internas y procesarlas adecuadamente. Para casos como este, ABAP/4 nos ofrece la técnica especial de los FIELD GROUPS.

Esta técnica consiste en crear conjuntos de datos intermedios. (‘intermediate datasets’). Se definen los diferentes registros con idéntica estructura, dentro de un mismo tipo de registro (FIELD GROUP). Será necesario definir todos los FIELD GROUP al inicio del report con:

FIELD-GROUP : HEADER, <f_g_1>, <f_g_2>...

El FIELD GROUP HEADER es fijo. Contendrá los campos por los cuales queremos ordenar el conjunto de datos intermedio.

Para determinar qué campos pertenecen a cada FIELD GROUP, utilizamos la instrucción :

INSERT <campo1> <campo2><campo_n> INTO HEADER.

INSERT <campo1> <campo2><campo_n> INTO <f_g_1>.

....

Un campo podrá estar dentro de varios FIELD GROUPS.

Para llenar con datos los conjuntos de datos intermedios se utiliza la instrucción:

EXTRACT <f_g_1>.

Esta instrucción asigna los contenidos de los campos especificados en el INSERT al FIELD GROUP indicado.

En cada EXTRACT, el sistema realiza automáticamente una extracción de los datos del FIELD GROUP HEADER, estos precederán siempre a los datos del FIELD GROUP sobre el que realizamos el EXTRACT.

Si algún campo de la cabecera no se llena, tomará el valor 0, de forma que el proceso de ordenación funcione correctamente.

Veamos el funcionamiento de los FIELD GROUP con un ejemplo.

Ejemplo:

Para realizar un listado de partidas de proveedores, ordenado por código de proveedor y números de documentos de las diferentes partidas. (Utilizaremos la base de datos lógica KDF).

TABLES: LFA1, LFB1, BSIK.

FIELD-GROUPS : HEADER, DIRECCION, IMPORTES.

INSERT LFA1-LIFNR BSIK-BELNR INTO HEADER.

INSERT LFA1-NAME1 LFA1-STRAS LFA1-PSTLZ LFA1-ORT01
INTO DIRECCION.

INSERT BSIK-DMBTR INTO IMPORTES.

*-----

GET LFA1.

EXTRACT DIRECCION.

GET BSIK.

EXTRACT IMPORTES.

*-----

En cada EXTRACT se va llenando el conjunto de datos intermedios.

EXTRACT DIRECCION:

PROVEEDOR1 RIVERLAND DIAGONAL 618 BARCELONA

EXTRACT IMPORTES:

PROVEEDOR1 DOC1 100.000

Así el dataset se irá llenando :


```
PROVEEDOR1 RIVERLAND DIAGONAL 618 BARCELONA
PROVEEDOR1 DOC1 100.000
PROVEEDOR1 DOC2 200.000
PROVEEDOR2 SAP A.G. PABLO PICASSO 28020 MADRID
PROVEEDOR2 DOC1 250.000
PROVEEDOR2 DOC2 1.200.000
```

Una vez extraídos los datos, los podemos procesar de forma similar a como lo hacíamos en las tablas internas.

En primer lugar ordenaremos el ‘dataset’, con la instrucción SORT. La ordenación se realizará por los campos que indica el HEADER.

Posteriormente podemos procesar los datos en un **LOOP...ENDLOOP**, pudiendo utilizar las instrucciones de ruptura por campos **AT NEW** y **AT END OF**. También podemos utilizar estos eventos por inicio y final de registro (FIELD-GROUP).

Además podemos comprobar si para un registro, existen registros asociados de otro tipo, con el evento:

```
AT <f_g1> WITH <f_g2>.
```

```
...
```

```
ENDAT.
```

Por ejemplo, si existen registros de importes para un registro de dirección, imprimir en el report los datos de dirección:

```
AT DIRECCION WITH IMPORTES.
```

```
WRITE: LFA1-NAME1 .....
```

```
ENDAT.
```

También podemos contar o sumar por campos con las instrucciones:

```
CNT (<campo>).
```

```
SUM (<campo>).
```

Así podríamos completar nuestro listado de proveedores del ejemplo con:

```
END-OF-SELECTION.
```

```
SORT.
```

```
LOOP.
```

```
AT DIRECCION WITH IMPORTES.
```

```
WRITE: LFA1-NAME1, LFA1-STRAS,
```

```
LFA1-PSTLZ, LFA1-ORT01.
```

```
ENDAT.
```

```
AT IMPORTES.
```

```
WRITE: BSIK-BELNR, BSIK-DMBTR.
```

```
ENDAT.  
AT END OF LFA1-LIFNR.  
SKIP.  
WRITE: 'Suma proveedor', LFA1-LIFNR,  
SUM(BSIK-DMBTR).  
SKIP.  
ENDAT.  
ENDLOOP.
```

Field symbols

Cuando tenemos que procesar una variable, pero únicamente conocemos de qué variable se trata y cómo tenemos que procesarla, en tiempo de ejecución, lo haremos mediante los 'field symbols'. Por ejemplo, si estamos procesando cadenas, y queremos procesar una parte de la cadena cuya posición y longitud depende del contenido de la misma, utilizaremos 'field symbols'.

Los Field Symbol tienen cierta similitud con los punteros o apuntadores de otros lenguajes de programación.

Los Field Symbol permiten soluciones elegantes a problemas pero su utilización incorrecta puede implicar resultados impredecibles.

Los Field Symbol se declaran con:

FIELD-SYMBOLS : <«Field Symbol»>.

La declaración se realizará en la rutina o módulo de función donde se utilice.

Para asignar un campo a un 'Field Symbol' utilizaremos la instrucción ASSIGN. Una vez asignado, cualquier operación que realicemos sobre el field symbol afectará al campo real. No hay ninguna diferencia entre utilizar el campo o el field symbol:

ASSIGN <campo> TO <«Field Symbol»>.

Ejemplos :

1.-

```
FIELD-SYMBOLS <F>.  
ASSIGN TRDIR-NAME TO <F>. 'ZPRUEBA'  
MOVE 'ZPRUEBA' TO <F>.  
WRITE TRDIR-NAME.
```

2.-

```
FIELD-SYMBOLS <F>.  
TEXT0 = 'ABCDEFGH' .  
INICIO = 2 . 'CDEFG'  
LONGITUD = 5 .
```

```
ASSIGN TEXTO+INICIO(LONGITUD) TO <F>.
WRITE <F>.
```

3.-

* Rellena con ceros por la izquierda.

```
FORM PONER_CEROS USING NUMERO VALUE(LONGITUD).
FIELD-SYMBOLS: <PUNTERO>.
LONGITUD = LONGITUD - 1.
ASSIGN NUMERO+LONGITUD(1) TO <PUNTERO>.
WHILE <PUNTERO> EQ SPACE.
SHIFT NUMERO RIGHT.
WRITE '0' TO NUMERO(1).
ENDWHILE.
ENDFORM.
```

También es posible utilizar asignación dinámica. Esto permite asignar un campo que sólo conocemos en tiempo de ejecución a un field symbol. Será necesario encerrar el campo entre paréntesis en la asignación del field symbol.

ASSIGN (<campo>) TO <«Field Symbol»>.

Ejemplo:

```
PARAMETERS: CAMPO(10).
FIELD-SYMBOLS: <F>.
ASSIGN (CAMPO) TO <F>.
WRITE <F>.
```

Imprimirá 'ZPRUEBA' (siendo 'ZPRUEBA' el valor que se ha introducido en el parámetro CAMPO al ejecutar el programa).

También podríamos usar los Field Symbols con la instrucción SORT:

SORT <F>.

Tratamiento de ficheros desde un programa en Abap/4

ABAP/4 dispone de una serie de instrucciones para manejar ficheros binarios o de texto (**OPEN, CLOSE, READ, TRANSFER**).

Una utilidad típica de estas instrucciones, como ya veremos más adelante, será para las interfaces entre otros sistemas y SAP, vía Batch Input.

- Para abrir un fichero utilizaremos la sentencia OPEN:

OPEN DATASET <fichero>.
FOR APPENDING / OUTPUT / (INPUT) Escritura / Lectura (por defecto)
IN BINARY MODE / IN TEXT MODE Binario (por defecto) / Texto.

Si SY-SUBRC = 0 Fichero abierto correctamente.

SY-SUBRC = 8 Fichero no se ha podido abrir.

- Para cerrar un fichero utilizamos CLOSE:

CLOSE DATASET <fichero>.

- Si queremos leer de un fichero utilizamos READ:

READ DATASET <fichero> INTO <registro>
(LENGTH <long>) Guarda en <long> la longitud del registro leído.

Los datos deben de ser de tipo char, no importando que sean únicamente números.

Ejemplo:

```
DATA: BEGIN OF REC,  
LIFNR(16),  
BAHNS(16),  
END OF REC.  
OPEN DATASET '/usr/test'.  
DO.  
READ DATASET '/usr/test' INTO REC.  
IF SY-SUBRC NE 0.  
EXIT.  
ENDIF.  
WRITE: / REC-LIFNR, REC-BAHNS.  
ENDDO.  
CLOSE DATASET '/usr/test'.
```

Notas :

- Se pueden leer de/hasta 4 ficheros simultáneamente
 - Si SY-SUBRC = 0 Fichero leído correctamente.
- SY-SUBRC = 4 Fin de fichero encontrado.

- Para escribir sobre un fichero disponemos de la instrucción TRANSFER:

TRANSFER < registro> TO <fichero>
(LENGTH <long>) Transfiere la longitud especificada en la variable <long>.

Por defecto la transferencia se realiza sobre un fichero secuencial (texto) a no ser que se abra el fichero como binario.

En el caso de que el fichero no se encuentre todavía abierto, la instrucción TRANSFER lo intentará en modo binario y escritura.

Universidad Tecnológica Nacional - Derechos Reservados