TEMA 6

Programación avanzada

1. Introducción

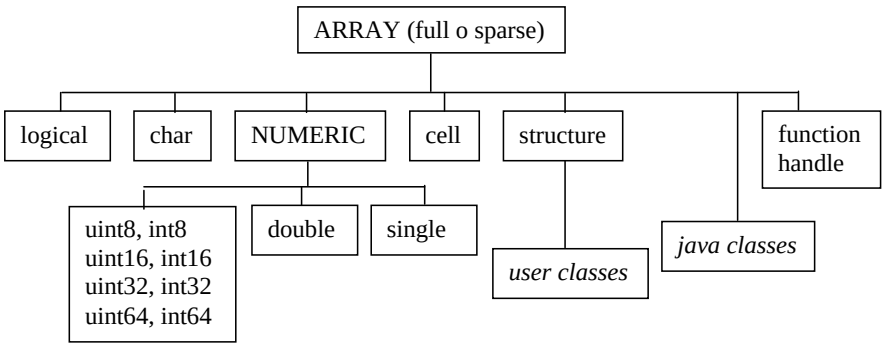
El objetivo del presente tema es mostrar otras posibilidades de MATLAB. No hay ninguna práctica asociada a este tema.

A continuación se presentan aspectos tales como la relación entre MATLAB y la programación orientada a objetos, la interficie de comunicación con otros lenguajes y programas de aplicación (API) y la comunicación con elementos de *hardware*.

2. Programación orientada a objetos

Clases en MATLAB: Los tipos de datos fundamentales (clases) en MATLAB son los siguientes:

- **`double`** (*double-precision floating-point number array*),
- **`single`** (*single-precision floating-point number array*),
- **`char`** (*character array*),
- **`logical`** (array de valores verdadero y falso),
- **`int8`** y **`uint8`** (*8-bit signed integer array*, *8-bit unsigned integer array*),
- **`int16`** y **`uint16`** (*16-bit signed integer array*, *16-bit unsigned integer array*),
- **`int32`** y **`uint32`** (*32-bit signed integer array*, *32-bit unsigned integer array*),
- **`int64`** y **`uint64`** (*64-bit signed integer array*, *64-bit unsigned integer array*),
- **`cell`** (*cell array*),
- **`struct`** (*struct array*),
- **`function_handle`** (array de valores para hacer llamadas a funciones)



La función **`class`** indica la clase de un objeto (esta información también es visible en la ventana *workspace*), por ejemplo:

```
>> x=@hola
x =
    @hola
>> class(x)
ans =
function_handle
```

Operaciones: Para cada una de las clases, MATLAB define unas operaciones concretas. Por ejemplo, se pueden sumar **`double`**'s pero no **`cell`**'s. Se pueden concatenar **`char`**'s pero no **`struct`**'s.

Objetos: A partir de la v5 MATLAB permite la creación de nuevas clases por parte del usuario y la posibilidad de definir nuevas operaciones para los tipos de datos básicos. Las variables de cada clase (o tipo de datos) se llaman *objetos*. La programación orientada a objetos (OOP, *Object Oriented Programming*) consiste en crear y usar dichos objetos.

Métodos: La colección de reglas o ficheros M que redefinen operadores y funciones reciben el nombre de *métodos*. Las operaciones sobre objetos se especifican por medio de métodos que encapsulan los datos y redefinen (*sobrecargan*) operadores y funciones. El encapsulado de los objetos impide que ciertas propiedades sean visibles desde la ventana de comandos con lo que para acceder a ellas hay que usar los *métodos* definidos para la clase.

Sobrecarga: Es posible redefinir las reglas internas de una operación o función. Ello recibe el nombre de sobrecargar (overload) y la operación o función resultante se dice que está sobrecargada.

Cuando el intérprete de MATLAB se encuentra con un operador como, por ejemplo, el producto, o una función con uno o más argumentos de entrada, primero mira el tipo de datos o clase de los operadores y, a continuación, actúa en consecuencia de acuerdo con las reglas definidas internamente. Por ejemplo, por defecto el producto está definido para variables de tipo numérico pero la operación producto se puede sobrecargar para que también sea posible realizar el producto de variables de tipo char (habrá que definir en qué consistirá disho producto).

Para facilitar la sobrecarga de los operadores **`+`**, **`-`**, **`.*`**, **`*`**,… MATLAB tiene definidas las funciones **`plus`**, **`minus`**, **`times`**, **`mtimes`**,… (ver Ejemplo 1). Para sobrecargar funciones se hace igual que para sobrecargar operadores.

Directorio de clase: Las reglas redefinidas para interpretar operadores y funciones son ficheros M de tipo función que se guardan en los directorios de clase (*class directories*) del MATLAB de nombre **`@class`** (donde **`class`** es el nombre de la variable). Por ejemplo los ficheros M del directorio **`@char`** redefinen funciones y operaciones sobre cadenas de caracteres.

Estos directorios no pueden estar directamente en el *search path* de MATLAB pero son subdirectorios de directorios que sí están en el *search path* de MATLAB. Es posible que haya múltiples directorios **`@class`** para el mismo tipo de datos (en ese caso, cuando MATLAB busque funciones en estos directorios seguirá el orden dado en el *search path*).

Ejemplo 1. Operador suma. Sobrecarga y directorio de clase [1]

En MATLAB el operador suma (o función plus) está definido para valores numéricos:

```
>> plus(3,8)
ans =
    11
>> 3+8
ans =
    11
```

Cuando se intenta usar con cadenas de caracteres, por ejemplo, `'asi'+'asa'`, lo que hace es encontrar el equivalente numérico ASCII de cada operando,

```
>> x=double('asi')
x =
    97   115   105
>> y=double('asa')
y =
    97   115    97
```

y sumarlos:

```
>> x+y
ans =
   194   230   202
>> plus('asi','asa')
ans =
   194   230   202
>> 'asi'+'asa'
ans =
   194   230   202
```

A continuación se va a sobrecargar este operador para que, cuando los argumentos de entrada sean dos cadenas de caracteres, en vez de pasar a ASCII y sumar, lo que haga sea concatenar las dos cadenas de caracteres.

Para ello se crea una nueva función **`plus.m`**

```
function s=plus(s1,s2)

if ischar(s1)&ischar(s2)
    s=[s1(:)',s2(:)'];
end
```

Para que esta nueva función pueda usarse cuando los argumentos de entrada son cadenas de caracteres, hay que guardarla en cualquier subdirectorio que lleve el nombre **`<@char>`**. Por ejemplo, dentro del directorio **`<work>`** creamos el directorio **`<@char>`** y guardamos esta función dentro de él. A continuación, verificamos su funcionamiento:

```
>> 'asi'+'asa'
ans =
asiasa
>> plus('asi','asa')
ans =
asiasa
```

Notar que si en vez de estar en **<work\@char>** está solo en **<work>** MATLAB no se entera de la sobrecarga.

Como MATLAB carga todos los subdirectorios de clase en el **startup**, si un directorio recién creado no se ve se puede, o bien reiniciar MATLAB, o bien ejecutar **rehash**.

Otras funciones útiles son: **methods**, **isa**, **class**, **loadobj** y **saveobj**.

Clases creadas por el usuario: Para crear una nueva clase, por ejemplo, la clase **polinomio** hay que crear el directorio de clase **@polinomio**. Este directorio debe contener como mínimo dos ficheros M de tipo *function*: **polinomio.m** y **display.m**. El primero de ellos es el *constructor* de la clase mientras que el segundo se usa para visualizar la nueva variable en la ventana de comandos. Aparte de estos dos ficheros habrá que definir ficheros M de *métodos* que permitan operar con la nueva clase creada.

Instancia: El fichero constructor **polinomio.m** lo que hace es crear una instancia de la clase. La instancia es un objeto que utiliza los métodos que sobrecargan el funcionamiento de los operadores y funciones cuando hay que aplicarlos al objeto considerado.

Fichero constructor de clase: Debe llamarse como la clase, es decir, **polinomio.m**. Debe manejar tres tipos de entradas: (1) si no se le pasan argumentos de entrada, debe generar una variable vacía; (2) si el argumento de entrada es de su misma clase debe pasarlo directamente a la salida; (3) si el argumento de entrada son datos para crear la nueva clase, debe crear una variable de dicha clase. Para ello hay que comprobar que los argumentos de entrada sean válidos y, a continuación, almacenarlos en los campos de una estructura. La nueva variable se crea cuando dichos campos son rellenados y se ejecuta la función **class**.

Ejemplo 2. Clases creadas por el usuario. Métodos [2]

En el directorio **<work>** hemos creado el subdirectorio **<@polinomio>**.

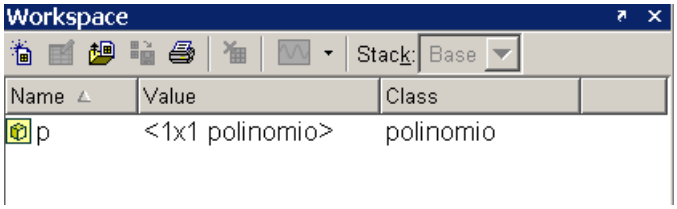
A continuación, en dicho subdirectorio, se crea el fichero constructor **polinomio.m**

```
function p=polinomio(vector_coefs)

if nargin == 0
   p.c = [];
   p = class(p,'polinomio');
elseif isa(vector_coefs,'polinomio')
   p = vector_coefs;
else
   p.c = vector_coefs(:).';
   p = class(p,'polinomio');
end
```

Lo probamos:

```
>> p = polinomio([1 0 -2 -5])
p =
 polinomio object: 1-by-1
```

Vemos que, efectivamente, aparece en el workspace.



Así pues, la clase polinomio representa polinomios por medio de estructuras con un solo campo .c que contiene los coeficientes. A este campo solo pueden acceder los métodos del directorio **@polinomio**.

```
>> p.c
??? Access to an object's fields is only permitted within its methods.
```

Aparte del método contructor **polinomio.m**, para que la clase sea útil debemos poder hacer cosas con sus objetos. Por ello, vamos a implementar los siguientes métodos: método para la conversión de polinomio a double, método para la conversión de polinomio a char, método de display, sobrecarga del operador *.

Método para la conversión de **polinomio** a **double**,

```
function c = double(p)
% POLINOMIO/DOUBLE
% Convierte el objeto polinomio en un vector de coeficientes
% Sintaxis:  c=double(p)
c = p.c;
```

Verificamos su funcionamiento:

```
>> p=polinomio([1 0 1]);
>> c=double(p)
c =
     1     0     1
```

Método para la conversión de **polinomio** a **char**,

```
function s = char(p)
% POLINOMIO/CHAR Convierte el objeto polinomio en char
% Sintaxis:  s=char(p)

if all(p.c==0),
    s='0';
else
   d=length(p.c)-1;%orden
   s=[];
   for a = p.c;
```

```
      if a ~= 0;
         if ~isempty(s)
            if a > 0,s = [s ' + '];
            else,s = [s ' - '];a = -a;
            end
         end
         if a ~= 1 | d == 0
            s = [s num2str(a)];
            if d > 0,s = [s '*'];end
         end
         if d >= 2,s = [s 'x^' int2str(d)];
         elseif d == 1, s = [s 'x'];
         end
      end
      d = d - 1;
   end
end
```

Verificamos su funcionamiento:

```
>> p=polinomio([1 0 3 0 -2]);
>> s=char(p)
s =
x^4 + 3*x^2 - 2
```

El método **`display`** hace uso de la función anterior:

```
function display(p)
% POLINOMIO/DISPLAY Muestra el objeto polinomio
% en la ventana de comandos
disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp(['    ' char(p)])
disp(' ');
```

Verificamos su funcionamiento:

```
>> p=polinomio([1 0 3 0 -2])
p =
   x^4 + 3*x^2 - 2
```

Sobrecarga del operador producto

```
function z = mtimes(x,y)
% POLINOMIO/MTIMES  Implementa x*y para objetos polinomio
x = polinomio(x);
y = polinomio(y);
z = polinomio(conv(x.c,y.c));
```

Verificamos su funcionamiento:

```
>> x=polinomio([1 1])
x =
   x + 1
>> y=polinomio([1 2])
y =
   x + 2
```

```
>> x*y
ans =
   x^2 + 3*x + 2
```

Se suguiere sobrecargar también el operador suma (función **`plus`**) a fin de que pueda sumar directamente objetos polinomio.

Otras funciones y operadores que se podrían sobrecargar son: **`minus`**, **`plot`**, **`roots`**, **`diff`**, **`polyval`**, **`subsref`** (*subscripted reference*),…

Para listar los métodos asociados a la clase polinomio, se puede usar la función **`methods`**:

```
>> methods('polinomio')
Methods for class polinomio:
char       display    double     mtimes     polinomio
```

Precedencia: Las clases definidas por el usuario tienen precedencia frente a las clases *built-in*. En aplicaciones simples no suele haber conflictos pero a medida que el número y complejidad de las clases crece, se recomienda usar las funciones **`inferiorto`** y **`superiorto`** (dentro del fichero constructor) para forzar la precedencia de las clases.

Herencia: Se puede crear una jerarquía de padres e hijos donde estos últimos heredan campos de datos y métodos de los padres. Una clase hija puede heredar de un solo padre o de varios.

Agregación: Se pueden crear clases mediante agregación, es decir, un tipo de objeto puede contener a otros objetos.

3. *Application Program Interface* (API)

Application Program Interface (API): En algunas ocasiones puede resultar útil que MATLAB interaccione e intercambie datos con programas externos. Para ello se ha definido su API (*Application Program Interface*) cuyas principales funciones intercambiar programas escritos en C y FORTRAN con MATLAB (ficheros MEX y Engine), importar/exportar datos hacia/desde MATLAB (ficheros MAT) y establecer relaciones cliente/servidor entre MATLAB y otros programas. Otras aplicaciones son la comunicación con dispositivos de *hardware* tales como puertos serie, tarjetas de adquisición de datos o DSPs.

3.1 El array de MATLAB

Para comunicar MATLAB con otros programas hay que usar el array de MATLAB. Cualquier fichero MEX, MAT o rutina ENGINE ha de acceder a arrays de MATLAB.

El array es el único objeto de MATLAB. Todas las variables se almacenan en forma de array de MATLAB. Todos los escalares y elementos compuestos tales como cadenas de caracteres, vectores, matrices, *cell arrays* y estructuras, son arrays de MATLAB. Los elementos del array pueden ser de cualquier tipo de datos fundamental u otros arrays.

El almacenamiento es por columnas, por ejemplo, $x = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$ se guarda en una columna cuyos elementos están ordenados de la siguiente manera: $\begin{bmatrix} a & c & e & b & d & f \end{bmatrix}^T$.

El objeto mxArray: Cuando se trabaja en C, el array de MATLAB se considera un nuevo tipo de dato en C y recibe el nombre **`mxArray`**.

El **`mxArray`** contiene información sobre:

- el tipo de array (*double*, *sparse*, *char*, *cell*, *object*, *single*, *uint8*…)
- las dimensiones del array
- las referencias a los datos
- información específica de cada tipo (p. ej.: si el array numérico es real o complejo; el numero de campos y nombres de una estructura o un objeto; los índices y número máximo de elementos no nulos en un *sparse array*)

Matrices numéricas: En las matrices numéricas, el objeto **`mxArray`** consta de dos columnas de *doubles* (u otro tipo de datos numérico): en la primera se guardan los números reales y en la segunda los números imaginarios. Se usan dos punteros para acceder a los datos: el puntero sobre el vector real (**`pr`**) y el puntero sobre el vector imaginario (**`pi`**). El puntero **`pi`** vale NULL si el array es real.

Funciones MX: Todas las funciones que acceden a los elementos del **`mxArray`** empiezan por el prefijo **`mx`**, por ejemplo,

```
x=mxGetScalar(prhs[0]);
```

El vector **`prhs`** (*pointer right hand side*) es un vector que contiene los punteros (*pointers*) de las variables de entrada (*right hand side*).

Aquí, **`mxGetScalar`** busca la posición apuntada por **`prhs[0]`** (que corresponde al primer elemento del vector puesto que aquí sí que los índices empiezan en 0 y no en 1), extrae el dato (que es de esperar que sea un escalar) y lo guarda en la variable **`x`**.

Para coger elementos de la columna de números reales o de la columna de números imaginarios se pueden usar las funciones **mxGetPr** y **mxGetPi**. Por ejemplo:

```
y=mxGetPi(prhs[1]);
```

Aquí **prhs[1]** es un puntero sobre la columna de imaginarios y establece que las componentes de la variable imaginaria **y** empezarán justo en **prhs[1]** (segunda posición de la columna de imaginarios).

mxGetPr hace lo mismo que **mxGetPi** pero para la columna de elementos reales del **mxArray**.

Algunas funciones MX:

- **mxCreateCellArray**, **mxCreateDoubleMatrix**,…: Para crear mxArrays de cells, matrices numéricas de doble precisión
- **mxGetM**: devuelve el número de filas en el mxArray
- **mxGetN**: devuelve el número de columnas en el mxArray
- **mxIsChar**, **mxIsComplex**,…: devuelve si la entrada es un char mxArray, complex mxArray,…
- **mxSetM**: establece el número de filas en el mxArray
- **mxSetN**: establece el número de columnas en el mxArray

3.2 Utilidad MEX

Para poder comunicar MATLAB con programas en lenguaje C y FORTRAN es necesario instalar los correspondientes compiladores y configurar la utilidad **mex**.

Compiladores:
Compiladores de FORTRAN:

- Plataforma Linux: G77 FORTRAN Compiler versión 3.3.3

Compiladores de C:

- Plataforma UNIX: ANSI C Compiler
- Plataforma Linux i386, con 2.6.6 kernel (arquitectura glnx86): GCC Compiler versión 3.3.3 (http://gcc.gnu.org/)
- Plataforma MacIntosh, OS X v10.3.4 (arquitectura mac): GCC Compiler
- Windows 2000 Pentium PC (arquitectura win32): LCC Compiler que viene con el propio MATLAB

También es posible configurar el MATLAB para usar otros compiladores (Microsoft Visual C Developer Studio 5, Watcom C, Borland C). En el subdrectorio **\bin\win32\mexopts** stán disponibles ficheros de opciones predefinidos para diversos compiladores.

Configuración de la utilidad MEX: Una vez instalado el compilador hay que indicarle a MATLAB cuál es. Para ello hay que ir a la ventana de comandos y configurar la utilidad **mex**. En las versiones más recientes el proceso es el siguiente:

Ejemplo 3. Mex setup en la v7

```
>> mex -setup
Please choose your compiler for building external interface (MEX) files:

Would you like mex to locate installed compilers [y]/n? y

Select a compiler:
[1] Lcc C version 2.4.1 in C:\ARCHIVOS DE PROGRAMA\MATLAB\R2006A\sys\lcc

[0] None

Compiler: 1

Please verify your choices:

Compiler: Lcc C 2.4.1
Location: C:\ARCHIVOS DE PROGRAMA\MATLAB\R2006A\sys\lcc

Are these correct?([y]/n): y

Trying to update options file: C:\Documents and Settings\Administrador\Application
Data\MathWorks\MATLAB\R2006a\mexopts.bat
From template:              C:\ARCHIVOS DE
PROGRAMA\MATLAB\R2006A\BIN\win32\mexopts\lccopts.bat

Done . . .

********************************************************************************
  Warning: The file extension of 32-bit Windows MEX-files was changed
           from ".dll" to ".mexw32" in MATLAB 7.1 (R14SP3). The generated
           MEX-file will not be found by MATLAB versions prior to 7.1.
           Use the -output option with the ".dll" file extension to
           generate a MEX-file that can be called in previous versions.
           For more information see:
           MATLAB 7.1 Release Notes, New File Extension for MEX-Files on Windows
********************************************************************************

>>
```

Ejemplo 4. Mex setup en la v5

En versiones más antiguas, la instrucción

```
» mex -setup
»
```

abre una ventana de MS-DOS de nombre PERL:

```
Welcome to the utility for setting up compilers
For building external interface file

C compilers
[1] Microsoft Visual C++
[2] Borland C/C++
[3] Watcom C/C++

Fortran compilers
[4] Microsoft PowerStation

[0] None

Compiler: 1
```

```
Which version
[1] Microsoft Visual C++ 4.x
[2] Microsoft Visual C++ 5.x
version: 2
Please enter the location of your C compiler: [c:\msdev] c:\devstudio

Please verify
Compiler: Microsoft Visual C++ 5.x
Location: c:\devstudio
Ok?([y]/n): y

'mexopts.bat' is being updated...
C:\MATLAB5\bin>_
```

Una vez hecho esto el fichero `mexopts.bat`, que guarda las opciones que MATLAB utilizará del compilador, queda de la siguiente manera:

```
@echo off
rem MSVC50OPTS.BAT
rem
rem    Compile and link options used for building MEX-files
rem    using the Microsoft Visual C++ compiler version 5.0
rem
rem    $Revision: 1.2 $  $Date: 1997/04/22 15:34:32 $
rem
rem ********************************************************************
rem General parameters
rem ********************************************************************
set MATLAB=%MATLAB%
set MSVC_ROOT=c:\devstudio
set MSVCDir=%MSVC_ROOT%\VC
set MSDevDir=%MSVC_ROOT%\sharedIDE
set PATH=%MSVCDir%\BIN;%MSDevDir%\bin;%PATH%
set
INCLUDE=%MSVCDir%\INCLUDE;%MSVCDir%\MFC\INCLUDE;%MSVCDir%\ATL\INCLUDE;%IN
CLUDE%
set LIB=%MSVCDir%\LIB;%MSVCDir%\MFC\LIB;%LIB%

rem ********************************************************************
rem Compiler parameters
rem ********************************************************************
set COMPILER=cl
set COMPFLAGS=-c -Zp8 -G5 -W3 -DMATLAB_MEX_FILE
set OPTIMFLAGS=-O2
set DEBUGFLAGS=-Zi

rem ********************************************************************
rem Library creation command
rem ********************************************************************
set        PRELINK_CMDS=lib        /def:%MATLAB%\extern\include\matlab.def
/machine:ix86 /OUT:%LIB_NAME%1.lib
set      PRELINK_DLLS=lib      /def:%MATLAB%\extern\include\%DLL_NAME%.def
/machine:ix86 /OUT:%DLL_NAME%.lib

rem ********************************************************************
rem Linker parameters
rem ********************************************************************
set LINKER=link
set       LINKFLAGS=/dll       /export:mexFunction       %LIB_NAME%1.lib
/implib:%LIB_NAME%l.lib
set LINKOPTIMFLAGS=
set LINKDEBUGFLAGS=/debug
set LINK_FILE=
set LINK_LIB=
set NAME_OUTPUT=/out:%MEX_NAME%.dll

rem ********************************************************************
```

```
rem Resource compiler parameters
rem ************************************************************
set RC_COMPILER=rc /fo mexversion.res
set RC_LINKER=
```

Notar que en el directorio `<bin>` ya existen diversos ficheros de opciones preconfigurados: `bccopts.bat` (para Borland C), `watcopts.bat` (para Watcom C) y `msvc50opts.bat` (para Microsoft Visual C 5.0).

Opciones de mex: Otras opciones son `-f` (para seleccionar otro fichero de inicialización cuando se usan programas Engine y MAT), `-v` (verbose, para ver la configuración del compilador y las etapas del complilación y lincado), `-g` (para usar debuggers tales como `dbx`, `ddd` o `gdb`), `-help` (para consultar la lista de las opciones disponibles).

3.3 Ficheros MEX

Ficheros MEX: Los ficheros MEX son programas (funciones compiladas) escritos en C y FORTRAN que pueden ser ejecutados desde MATLAB, como si fueran funciones M.

Se trata de rutinas dinámicamente lincadas (en Windows su extensión es `*.dll` o `*.mexw32`) que el intérprete de MATLAB carga y ejecuta automáticamente. A partir de la v7.1 la extensión `*.dll` ha sido sustituida por la extensión `*.mexw32`. Teclear `>>mexext` para ver la extensión en cada caso:

```
>> mexext
mexw32
```

Los ficheros MEX se utilizan principalmente por dos motivos:

- El usuario dispone de programas en C o FORTRAN que desearía ejecutar en MATLAB sin tener que volver a reescribir el código en lenguaje MATLAB. Para hacer esto posible basta con ampliar el código fuente original con unas pocas líneas más de código (cabecera y *gateway*) y compilar con la utilidad **`mex`**.

- Ciertos cálculos que en MATLAB son demasiado lentos (como, por ejemplo, los bucles **`for`**) se pueden programar en C o FORTRAN para reducir el tiempo de ejecución del programa global.

Se recomienda crear un fichero M para cada fichero MEX conteniendo los comentarios de ayuda.

Organización del código fuente del fichero MEX: En el código de los ficheros MEX se distinguen las siguientes partes:

- **Cabecera** (con los correspondientes **`include`**, **`define`**,...)

Como mínimo tiene que contener el comando **`#include "mex.h"`**).

El fichero de cabecera **`"mex.h"`** también incluye **`"matrix.h"`** para proporcionar funciones MX que soportan los tipos de datos de MATLAB, junto con los ficheros de cabecera estándar **`<stdio.h>`**, **`<stdlib.h>`** y **`<stddef.h>`**.

El prefijo **`mx`** indica que son funciones que operan sobre tipos de datos del MATLAB.

- **Rutina computacional** (instrucciones que ejecutan lo que deseamos que haga el programa)

- **Enlace con MATLAB o *gateway routine*** (hace de interficie con MATLAB).

  Se declara como:

```
void mexFunction( int nlhs,       mxArray *plhs[],
                  int nrhs, const mxArray *prhs[], )
{
}
```

donde

`nlhs` (*number of left hand side arguments*): número de argumentos de salida.

`nrhs` (*number of rigth hand side arguments*): número de argumentos de entrada.

`plhs` (*pointer to left hand side arguments*): vector que contiene los punteros hacia los valores de los argumentos de salida.

`prhs` (*pointer to left hand side arguments*): Análogo al anterior pero para los argumentos de entrada.

Notar que, a diferencia del lenguaje MATLAB, aquí sí que el primer índice es el 0, no el 1. Así,

- **`plhs[0]=6`** significa que el primer argumento de salida empieza en la posición **`6`** del **`mxArray`** de datos;
- **`prhs[1]=40`** significa que el segundo argumento de entrada empieza en la posición **`40`** del **`mxArray`** de datos;
- **`plhs[nlhs-1]`** es el puntero al último argumento de salida;
- **`plhs[0]=NULL`**, significa que no hay argumentos de salida.

Es en esta rutina de enlace donde:

1) Se verifica el número, tipo y coherencia de los argumentos de entrada y salida y se envían *warnings* y mensajes de error si no son los esperados.
2) Se obtienen los argumentos de entrada.
3) Se crean las matrices que han de contener los argumentos de salida.
4) Se llama a la rutina computacional.
5) Se guardan los argumentos de salida en el espacio asignado dentro del **`mxArray`**.

Algunas funciones MEX: Las funciones más usadas habitualmente son:

```
mexCallMATLAB
mexFunction
mexFunctionName
```

```
mexGetVariable
mexPutVariable
mexGetVariablePtr
mexEvalString
mexErrMsgTxt
mexErrMsgIdAndTxt
mexWarnMsgTxt
mexIsGlobal
mexPrintf
```

- `mexGetVariable`: copia una variable desde el *workspace* de MATLAB a un `mxArray`. (Nota: `mexGetVariable` en la v7 incluye y sustituye las funciones `mexGetArray`, `mexGetMatrix` y `mexGetFull` de versiones anteriores).
- `mexGetVariablePtr`: devuelve un puntero de solo lectura a una variable de MATLAB.
- `mexPutVariable`: copia un `mxArray` al *workspace* de MATLAB (mexPutVariable incluye y sustituye a `mexPutArray`, `mexPutMatrix` y `mexPutFull`).

Ejemplo 5. Fichero C MEX

A continuación vamos a crear un fichero C MEX que multiplica un escalar por 2. El fichero fuente **`mult2.c`** es el siguiente:

```
#include "mex.h"

/*Rutina computacional*/
void mult2(double y[], double x[])
{
y[0]=2.0*x[0];
}

/*Rutina de enlace con Matlab*/
void mexFunction( int nlhs,       mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    double *x, *y;
    int mfil,ncol;
    /* verificar que el argumento de entrada sea double */
    if(!mxIsDouble(prhs[0]))
    {mexErrMsgTxt("La entrada ha de ser 'double'");}
    /* crear la matriz que contendrá la salida */
    mfil=mxGetM(prhs[0]);
    ncol=mxGetN(prhs[0]);
    plhs[0]=mxCreateDoubleMatrix(mfil,ncol,mxREAL);
    /* asignar punteros a la entrada y a la salida */
    x=mxGetPr(prhs[0]);
    y=mxGetPr(plhs[0]);
    /* llamar a la rutina computacional */
    mult2(y,x);
}
```

Una vez escrito el código fuente, se vuelve a la ventana de comandos y, con ayuda de **`mex`**, se compila, linca y obtiene el ejecutable, **`mult2.dll`** o **`mult2.mexw32`**, según la versión:

```
» mex mult2.c
»
```

Finalmente se verifica su funcionamiento:

```
>> mult2(4.3)
ans =
    8.6000

>> mult2('bah')
??? La entrada ha de ser ‘double’
```

Se sugiere ampliar el código del fichero del ejemplo para que dé mensajes de error cuando el usuario introduzca:

1) Más de un argumento de entrada, por ejemplo, **`mult2(3,4)`** (Nota: usar **`nrhs`**)
2) Más de un argumento de salida, por ejemplo, **`[x,y]=mult2(5)`** (Nota: usar **`nlhs`**)
3) Un número complejo **`mult2(3*j)`** (Nota: usar **`mxIsComplex`**)
4) Una entrada no escalar **`mult2(ones(3))`** (Nota: usar **`mfil`**, **`ncol`**)

Librerías compartidas: Las librerías son colecciones de funciones disponibles para cualquier programa y están precompiladas en ficheros de librería. Las librerías pueden ser estáticas o compartidas. Las *estáticas* se lincan al programa cuando se realiza la compilación y así las funciones referenciadas quedan incluidas en el ejecutable creado. Las librerías *compartidas* se lincan al programa durante la ejecución (run time). Diferentes programas activos en un mismo ordenador en el mismo momento pueden compartir acceso a estas librerías.

MATLAB realiza la conversión entre datos de MATLAB y datos de C de manera automática pero también es posible controlar el proceso mediante funciones de librerías compartidas como, por ejemplo, `libisloaded` (para ver si cierta librería está cargada), `loadlibrary` (para cargar una librería externa), `unloadlibrary`, `libpointer` (crea un objeto puntero para usar con librerías externas), `libstruct` (crea una estructura como las de C para poderla pasar a librerías externas), etc.

3.4 Rutinas ENGINE

Las rutinas ENGINE sirven para llamar a MATLAB desde otros programas utilizando MATLAB como proceso computacional interno a ellos. La parte de MATLAB corre en el *background* como si fuera un proceso separado del programa que lo invoca. Una vez terminados los cálculos, MATLAB pasa los datos al programa en C o FORTRAN que lo ha llamado.

La ventaja es que no es preciso lincar todo el MATLAB con dicho programa (sería demasiado código) sino que basta con sólo una pequeña librería de comunicación.

El uso del ENGINE puede ser puntual (llamar a MATLAB para que invierta un array o bien para que calcule una FFT) o más elaborado, por ejemplo, montar un sistema entero con el *front-end* en otro programa y el *back-end* en MATLAB. Sería el caso de las aplicaciones con dispositivos físicos y en tiempo real.

En Windows las rutinas ENGINE hacen uso del Active X.

Organización: En el código de las rutinas ENGINE se distinguen dos partes: la **cabecera** (que debe contener al menos el comando `#include "engine.h"`) y el **cuerpo principal** (`main`) donde se abre la rutina ENGINE, se introducen los comandos que debe ejecutar el Matlab y se cierra la rutina ENGINE.

Algunas funciones ENG:

```
engOpen
engPutVariable
engGetVariable
engOutputBuffer
engEvalString
engClose
```

Ejemplo 6. Engine de MATLAB

Programa C (seno.c) que llama a las funciones del Matlab Engine

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "engine.h"
#define  BUFSIZE 256

int main()
{
    Engine *ep;
    mxArray *X = NULL;
    char buffer[BUFSIZE+1];
    double x[10] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
};

    /*Si no se abre el Engine, hay que configurarlo (help)*/
    if (!(ep = engOpen(NULL)))
    {
        fprintf(stderr, "\nNo se puede iniciar el MATLAB engine\n");
        return EXIT_FAILURE;
    }

    /*Se crea la variable X para guardar los valores de x */
    X = mxCreateDoubleMatrix(1, 10, mxREAL);
    memcpy((void *)mxGetPr(X), (void *)x, sizeof(x));

    /*Se pone X en el workspace*/
    engPutVariable(ep, "X", X);

    /*Se calcula el seno de X */
    engEvalString(ep, "Y = sin(X);");
```

```
    /*Se representa*/
    engEvalString(ep, "plot(X,Y);");
    engEvalString(ep, "title('seno');");
    engEvalString(ep, "xlabel('x');");
    engEvalString(ep, "ylabel('y');");

    /*fgetc() es para mantener el plot y que se pueda ver*/
    printf("Pulsa una tecla para terminar\n\n");
    fgetc(stdin);
    printf("Fin ejemplo\n");

    /*Se libera memoria y se cierra el engine*/
    mxDestroyArray(X);
    engEvalString(ep, "close;");
}
```

Al aplicar la utilidad mex,

```
>> mex -f lccengmatopts.bat seno.c
```

Aparece un fichero de nombre **seno.exe**. Este fichero puede ejecutarse desde MS-DOS. Se sugiere hacer doble clic sobre él cuando Matlab está cerrado. Se observará como desde DOS se abre el Matlab (una nueva command window), y se representa el seno.

Si aparece el siguiente error:

```
>> mex -f lccengmatopts.bat seno.c
>> !seno.exe

No se puede iniciar el MATLAB engine
>>
```

es que matlab no está registrado como un servidor COM. En general este registro se hace durante la instalación. Si por algún motivo no ha sido así hay que hacerlo manualmente:

```
>> matlabroot

ans =

C:\Archivos de programa\MATLAB\R2006a

>> !cd C:\Archivos de programa\MATLAB\R2006a\bin\win32
>> !matlab /regserver

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>>
```

3.5 Ficheros MAT

MATLAB también dispone de ficheros de cabecera y librerías para crear y acceder a ficheros MAT desde programas escritos en C y FORTRAN.

Algunas funciones MAT:

```
matOpen
matClose
matGetDir
matGetVariable
matGetVariableInfo
matPut Variable
matPutVariableAsGlobal
matDeleteVariable
```

Ejemplo 7. Ejemplo de fichero MAT [1]

El código fuente del fichero **`fichero_mat.c`** es el siguiente:

```
#include "mat.h"

int makemat(const char *nomf,
            double *datos, int m, int n,
            char *str)
{
    MATFile *mnomf;
    mxArray *mdatos, *mstr;

    /*abrir el fichero MAT para escribir en él*/
    mnomf=matOpen(nomf,"w");
    if (mnomf==NULL){
        printf("Error al escribir %s.\n",nomf);
        return(EXIT_FAILURE);
    }

    /*crear el array*/
    mdatos=mxCreateDoubleMatrix(n,m,mxREAL);

    /*poner los datos en el array*/
    memcpy((void *)(mxGetData(mdatos)),(void *)datos,
m*n*sizeof(double));

    /*crear el string del array*/
    mstr=mxCreateString(str);

    /*escribir los mxArrays en el fichero MAT*/
    matPutVariable(mnomf,"dades",mdatos);
    matPutVariable(mnomf,"nom",mstr);

    /*liberar la memoria del mxArray*/
    mxDestroyArray(mdatos);
    mxDestroyArray(mstr);

    /*cerrar el fichero MAT*/
    if (matClose(mnomf)!=0){
```

```
        printf("Error al cerrar %s.\n",nomf);
        return(EXIT_FAILURE);
        }
    return(EXIT_SUCCESS);
}

int main()
{
    int status;
    char *mstr="Ejemplo de fichero MAT";
    double datos[2][3]={{1.0, 2.0, 3.0},
                        {-1.0, -4.0, -9.0}};
    status=makemat("prova.mat", *datos, 2, 3, mstr);
    return(status);
}
```

Para probar su funcionamiento podemos hacr lo siguiente:

```
>> mex -f lccengmatopts.bat fichero_mat.c
>> !fichero_mat.exe
>> clear all
>> load prova.mat
>> who

Your variables are:

dades  nom

>> dades

dades =

     1    -1
     2    -4
     3    -9

>> nom

nom =

Ejemplo de fichero MAT
```

---

4. Relación con elementos de *hardware*

4.1 Comunicación con puerto serie

(Hanselman) Muchas veces los datos que vamos a utilizar vienen del exterior. El caso más sencillo es usar la interficie *built-in* del puerto serie para recoger y llevar datos directamente al MATLAB, aunque también existen *toolboxes* específicas que proporcionan funciones adicionales.

MATLAB soporta los estándares de comunicación serie RS-232, RS-422 y RS-485 en la plataformas Windows, Linux y Sun Solaris. Los dispositivos tienen diferentes nombres según sea la plataforma (`COM1` en Windows, `/dev/ttyS0` en i386 Linux)

Función serial: La interficie de MATLAB con el puerto serie está basada en objetos. La función **`serial`** crea el objeto de MATLAB conectado a un puerto serie específico y sus propiedades son accesibles con ayuda de **`set`** y **`get`**.

```
>> s=serial('COM1')

   Serial Port Object : Serial-COM1

   Communication Settings
      Port:               COM1
      BaudRate:           9600
      Terminator:         'LF'

   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
      BytesAvailable:     0
      ValuesReceived:     0
      ValuesSent:         0
```

Los objetos puerto serie soportan eventos y callbacks. Los callbacks son funciones que hay que ejecutar cuando ciertos eventos ocurren (fin del temporizador, interrupciones, buffer de salida vacío, datos disponibles en el buffer de entrada, cambios en el estado de un pin…)

Comunicación: Consiste en los siguientes pasos:

- Crear la interficie con el dispositivo serie con ayuda de la función `serial`
- Ajustar las propiedades con `set`
- Abrir el dispositivo con `fopen`
- Escribir y leer con `fprintf` y `fscanf`
- Cerrar el dispositivo con `fclose`
- Borrar el objeto serie con `delete`

4.2 Comunicación con tarjetas de adquisición de datos

Existen *toolboxes* y *blocksets* específicas para dispositivos tales como tarjetas de E/S y DSPs. Ver por ejemplo los *blocksets xPC Target*, diversos *Embedded Targets*, etc. Estas librerías ya contienen los bloques necesarios para comunicar con los dispositivos de *hardware* y, por tanto, si se dispone de ellas no es necesario crear y compilar los *drivers* tal y como se va a describir en el presente ejemplo.

Ejemplo 8. Conversión A/D y D/A en una tarjeta PCL-812-PG [3]

Vamos a ilustrar los pasos básicos en la comunicación de MATLAB con las entradas y salidas analógicas de una tarjeta de adquisición de datos Advantech PCL-812-PG mediante funciones S. En el Apéndice hay más datos sobre esta tarjeta.

La Función S contiene el programa en C que permite interaccionar en tiempo real con los puertos de entrada y salida de la tarjeta de adquisición de datos.

Función S para la conversión A/D:

```
/*              pcl812ad.c          Entradas analógicas             */

#define S_FUNCTION_NAME pcl812ad  /*igual nom que el fitxer*/
#include "simstruc.h"

#include <conio.h>
#include <stdio.h>
#include <iostream.h>

#define NO_CANAL  (ssGetSFcnParam(S,0))
#define CANAL     ((int) mxGetPr(NO_CANAL)[0])

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumContStates(     S, 0);
    ssSetNumDiscStates(     S, 0);
    ssSetNumInputs(         S, 0);
    ssSetNumOutputs(        S, DYNAMICALLY_SIZED);
    ssSetDirectFeedThrough( S, 0);
    ssSetNumSampleTimes(    S, 1);
    ssSetNumSFcnParams(     S, 1);
    ssSetNumRWork(          S, 0);
    ssSetNumIWork(          S, 0);
    ssSetNumPWork(          S, 0);
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
printf("\ncanal A/D no.:  ");printf("%d\n",CANAL);
}

static void mdlOutputs(double *y, const double *x, const double *u,
SimStruct *S, int tid)
{
unsigned short base=0x220;
int i,j,n;
int datoh,datol,dato;

for(n=0;n<2;n++){
_outp(base+10, (int) CANAL);  /*canal d'entrada*/
_outp(base+9,  (int) 0);      /*guany*/
```

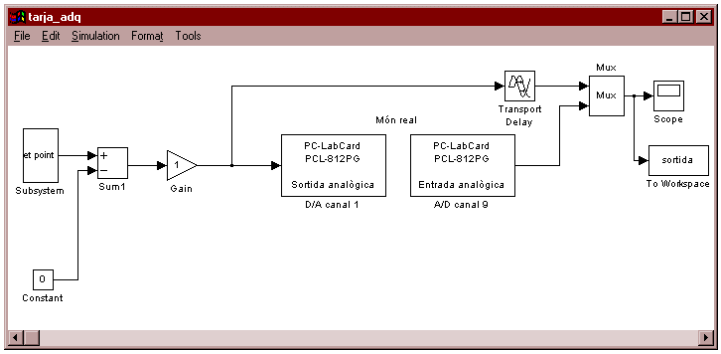
```
_outp(base+11, (int) 1);      /*mode: disparo i transferencia*/   }

outp(base+12,0);          /*disparo*/
j=0;do{i=_inp(base+5)&0x10;j++;}while(i==0 && j!=1000);  /*esperem que
DRDY=0*/
datoh=(_inp(base+5)<<8)&0x0f00;  /*llegim byte alt*/
datol=_inp(base+4)&0x00ff;       /*llegim byte baix*/
dato=datoh|datol;
*y=(dato*(5.0/2047.5)-5.0);
}

static void mdlUpdate(double *x, const double *u, SimStruct *S, int tid)
{}

static void mdlDerivatives(double *dx, const double *x, const double *u,
SimStruct *S, int tid)
{}

static void mdlTerminate(SimStruct *S)
{}

#ifdef  MATLAB_MEX_FILE    /* Compilarem com un MEX-file? */
#include "simulink.c"      /* interficie MEX-file */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

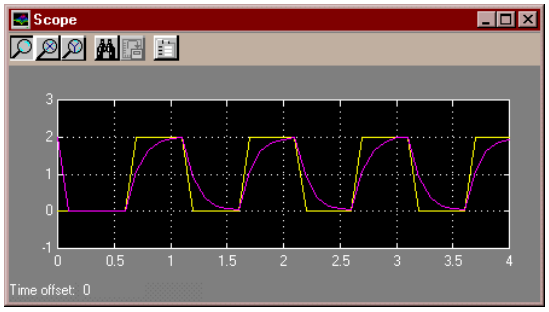
Función S para la conversión D/A:

```
/*            pcl812da.c           Sortides analògiques             */

#define S_FUNCTION_NAME pcl812da  /*igual nom que el fitxer*/
#include "simstruc.h"

#include <conio.h>
#include <stdio.h>
#include <iostream.h>

#define NO_CANAL  (ssGetSFcnParam(S,0))
#define CANAL     ((int) mxGetPr(NO_CANAL)[0])

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumContStates(     S, 0);
    ssSetNumDiscStates(     S, 0);
    ssSetNumInputs(         S, DYNAMICALLY_SIZED);
    ssSetNumOutputs(        S, 0);
    ssSetDirectFeedThrough( S, 0);
    ssSetNumSampleTimes(    S, 1);
    ssSetNumSFcnParams(     S, 1);
    ssSetNumRWork(          S, 0);
    ssSetNumIWork(          S, 0);
    ssSetNumPWork(          S, 0);
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
```

```
}

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
printf("\ncanal D/A no.:  ");printf("%d\n",CANAL);
}

static void mdlOutputs(double *y, const double *x, const double *u,
SimStruct *S, int tid)
{
unsigned short base=0x220;
int datoh,datol,dato;

dato=4095*(*u)/5.0;
datol= dato     & 0x00ff;
datoh=(dato>>8) & 0x000f;
if (CANAL==1) {
   _outp(base+4, datol);_outp(base+5, datoh);  }
if (CANAL==2) {
   _outp(base+6, datol);_outp(base+7, datoh);  }
}

static void mdlUpdate(double *x, const double *u, SimStruct *S, int tid)
{}

static void mdlDerivatives(double *dx, const double *x, const double *u,
SimStruct *S, int tid)
{}

static void mdlTerminate(SimStruct *S)
{}

#ifdef  MATLAB_MEX_FILE    /* Compilarem com un MEX-file? */
#include "simulink.c"      /* interficie MEX-file */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Una vez utilizada la función `mex`, se incluyen estas *S functions* en sus bloques correspondientes y se comprueba el funcionamiento (en este caso, se ha probado con un circuito RC)




Scope
3
2
1
0
-1
0
0.5
1
1.5
2
2.5
3
3.5
4
Time offset: 0

5. Relación con JAVA

(Hanselman) Java y MATLAB están muy relacionados. La máquina virtual de Java (JVM, *Java Virtual Machine*) es el intérprete que está detrás del GUI de MATLAB y MATLAB puede manipular las clases, objetos y métodos de Java.

Máquina virtual de Java: Java es un lenguaje de programación orientado a objetos, diseñado para entornos distribuidos, donde hay diferentes tipos de ordenadores y diferentes sistemas operativos. Los programas de Java se compilan en un *bytecode* independiente del tipo de ordenador o sistema operativo. La máquina virtual de Java es el programa que se encarga de interpretar este *bytecode* en el código máquina particular de cada ordenador, de manera que mientras el programa se ejecuta se ajustan las posibles diferencias en cuanto a la longitud de los comandos, formas de almacenamiento de datos, etc.

Librerías: Reciben el nombre de *class libraries*, *toolkits* o *packages*. Son colecciones de clases de Java relacionadas. La *Abstract Windowing Toolkit* (`java.awt.*`) es la que proporciona los servicios para crear y manejar ventanas y los servicios de GUI. La *Net Class Library* (`java.net.*`) proporciona servicios de comunicación e Internet.

Ejemplo 9. Java Net Package [1]

```
>> id=java.net.InetAddress.getLocalHost
id =
UPCNET-6D1E/127.0.0.1
>> id.getHostName
ans =
UPCNET-6D1E
>> id.getHostAddress
ans =
127.0.0.1
```

Ejemplo 10. Java Abstract Window Toolkit [1]

```
function vent_movil(varargin)

import java.awt.*
persistent ventana mq x y w h bl br

if isempty(ventana)
    ventana=Frame('Ejemplo de ventana');
    ventana.setLayout(FlowLayout);
    %dimensiones y color
    x=450;y=550;w=430;h=100;
    ventana.setLocation(x,y);ventana.setSize(w,h);
    set(ventana,'Background',[.5 .5 .5]);
```

```
    %Menu bar
    mf=Menu('Archivo');
    mq=MenuItem('Salir');
    set(mq,'ActionPerformedCallback','vent_movil(''salir'')');
    mf.add(mq);
    mb=MenuBar;
    mb.add(mf);
    ventana.setMenuBar(mb);
    %botones
    bl=Button('Izqda');
    br=Button('Dcha');
    set(bl,'MouseClickedCallback','vent_movil(''izqda'')');
    set(br,'MouseClickedCallback','vent_movil(''dcha'')');
    ventana.add(bl);
    ventana.add(br);
    %mostrar ventana
    ventana.setVisible(1);
elseif nargin==0
    x=450;y=550;w=430;h=100;
    ventana.setLocation(x,y);ventana.setSize(w,h);
    ventana.setVisible(1);
elseif nargin==1
    switch varargin{1}
        case 'izqda', x=x-20;
        case 'dcha',x=x+20;
        case 'salir', ventana.setVisible(0);return
    end
    ventana.setBounds(x,y,w,h);
    ventana.setVisible(1);
end
```

El resultado es la siguiente ventana:

que se mueve hacia la izquierda o hacia la derecha conforme se hace clic en el botón correspondiente. Se sugiere ampliar la función para que también se pueda mover hacia arriba y hacia abajo.

---

6. Relación con Windows

Con respecto a *Windows* (`win32`), MATLAB puede usar objetos COM (*Component Object Model*), controles ActiveX y DDE (*Dynamic Data Exchange*). MATLAB puede actuar como un servidor COM Automation a fin de comunicar con visual basic (VB) o con visual basic para aplicaciones (VBA) como es el caso de Microsoft Excel, PowerPoint y Word.

En UNIX y Linux para comunicar programas entre sí se usan *pipes*. En ellas, la salida estándar de un programa se redirecciona a la entrada estándar del otro programa.

En *Windows*, el DDE permite a las diversas aplicaciones comunicarse entre ellas para enviar comandos e intercambiar datos. El llamado OLE (*Object Linking and Embedding*) se construyó a partir de DDE y VBX (*Visual Basic Extensions*). Más tarde, algunas partes del OLE relacionadas con Internet y con las interficies gráficas de usuario GUI (*Graphical User Interface*) fueron renombradas como *ActiveX*. Y, finalmente, Microsoft renombró todo el entorno de componentes como COM (*Component Object Model*).

6.1 MATLAB como cliente o servidor COM (*Component Object Model*)

MATLAB puede funcionar como cliente COM. Para ello, la función **`actxserver`** crea un objeto `COM` y abre la conexión con un *COM Automation server* (éste puede ser Word, PowerPoint,...). Una vez creado el objeto se puede acceder a sus propiedades con **`set`** y **`get`**. Por ejemplo:

```
>> s=actxserver('powerpoint.application');
>> s=actxserver('excel.application');
>> s=actxserver('word.application');
>> get(s)
```

Para ver los métodos asociados a los objetos `COM`, basta con teclear

```
>> methods COM
Methods for class COM:
addproperty     events          invoke          propedit        set
delete          get             load            release
deleteproperty  interfaces      move            save
```

Para ver la ayuda de cada uno de estos métodos: `>> help COM/invoke`.

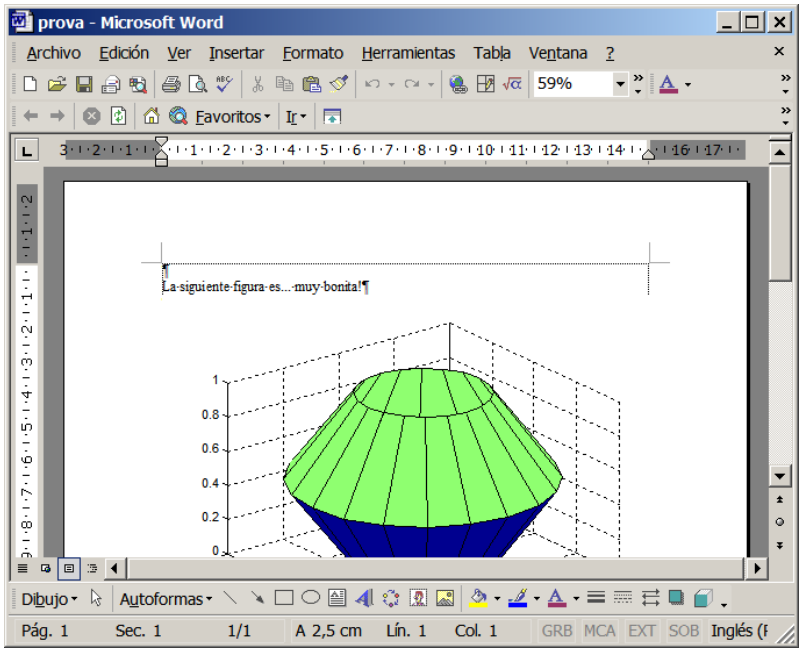
Ejemplo 11. MATLAB como cliente y WORD como servidor [1]

El siguiente fichero M, **`figu_word.m`**, crea una figura, abre el word, pega la figura, y cierra el word.

```
function figu_word(nomf)

%FIGU_WORD  abre un doc de word y pega la figura actual

%creamos la figura
close all,clc,cylinder([1 2 1]);

if nargin<1
    [fitxer,directori]=uiputfile('*.doc','Indicar o crear el fichero
word:');
    if isequal(fitxer,0),return,end
    nomf=fullfile(directori,fitxer);
end

%copiamos la figura como metafile
if nargin<2
    print('-dmeta');
```

```
end

%iniciar word
wrd=actxserver('word.application');
wrd.Visible=1;

%abrir el nomf.doc (o crearlo si no existe)
if exist(nomf,'file');
    docu=invoke(wrd.Documents,'Open',nomf);  %abrir nomf.doc
else
    docu=invoke(wrd.Documents,'Add');        %o crearlo
end

%añadir texto a nomf.doc
s=docu.Content;
s.InsertParagraphAfter;  %punto y aparte
invoke(s,'InsertAfter','La siguiente figura es... muy bonita!');
s.InsertParagraphAfter;
invoke(s,'Collapse',0);  %para que la fig se pegue después del texto

%pegar la figura:  (1): picture format, (3): float over text
invoke(s,'PasteSpecial',0,0,1,0,3);

%salvar y cerrar nomf.doc
if exist(nomf,'file')
    docu.Save;
else
    invoke(docu,'SaveAs',nomf,1);
end
docu.Close;

%salir de word y cerrar la conexión
wrd.Quit;
delete(wrd);
return
```

El resultado es:

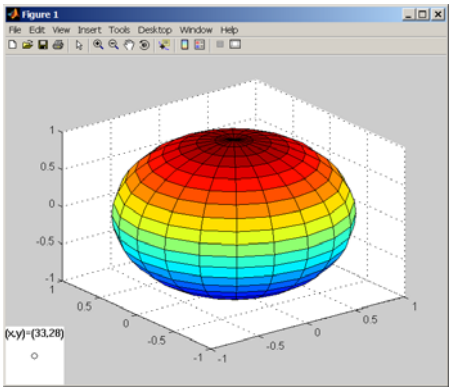


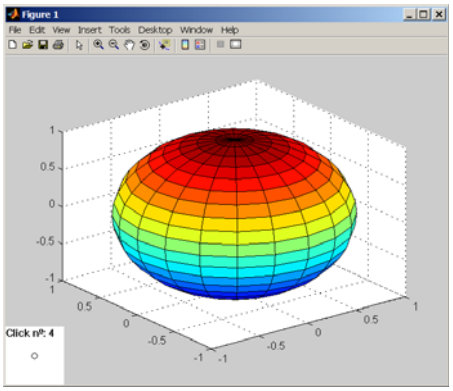
El MATLAB básico dispone de dos controles ActiveX muy simples: `mwsamp.ocx` y `mwsamp2.ocx` con sus librerías asociadas: `mwsamp.tlb` y `mwsamp2.tlb`. La clase `mwsamp` solo tiene un evento (`Click`) mientras que la clase `mwsamp2` es una ampliación que incluye los eventos `DblClick` y `MouseDown`.

Ejemplo 12. Controles ActiveX [1]

La función `cdemo.m` ilustra el funcionamiento de la clase `mwsamp2`. Con un clic simple sobre el control ActiveX lo que se muestra son las coordenadas del clic. Si se hace un doble clic lo que se muestra es el número de dobleclics realizados hasta el momento.

```
function cdemo(varargin)

persistent numclicks h

if nargin==0
    clc,close all,
    f=figure;sphere;numclicks=0;

    h=actxcontrol('MWSAMP.MwsampCtrl.2',[0 0 90 90],f,'cdemo');
    set(h,'Label','Clicar');
    h.Radius=5;
    invoke(h,'Redraw');

else

    if strcmp(varargin{end},'DblClick')
        numclicks=numclicks+2;
        h.Label=['Click nº: ',num2str(numclicks)];
    elseif strcmp(varargin{end},'MouseDown')
        h.Label=['(x,y)=(',num2str(varargin{5}),',',num2str(90-
varargin{6}),')'];
    end

    h.Redraw;

end
```





6.2 *Notebook* de MATLAB

El *notebook* de MATLAB permite incrustar comandos, resultados y gráficos de MATLAB en un documento de *Microsoft Word* (Nota: el *notebook* de MATLAB está pensado sólo para *word*).

Configuración: Teclear la siguiente instrucción en la ventana de comandos:

```
>> notebook -setup

Welcome to the utility for setting up the MATLAB Notebook
for interfacing MATLAB to Microsoft Word

Setup complete
```

Es posible que pida especificar la versión. En ese caso aparecerán las siguientes opciones:

```
>> notebook -setup

Welcome to the utility for setting up the MATLAB Notebook
for interfacing MATLAB to Microsoft Word

Choose your version of Microsoft Word:
[1] Microsoft Word 97 (will not be supported in future
releases)
[2] Microsoft Word 2000
[3] Microsoft Word 2002 (XP)
[4] Microsoft Word 2003 (XP)
[5] Exit, making no changes

Microsoft Word Version: 4
```

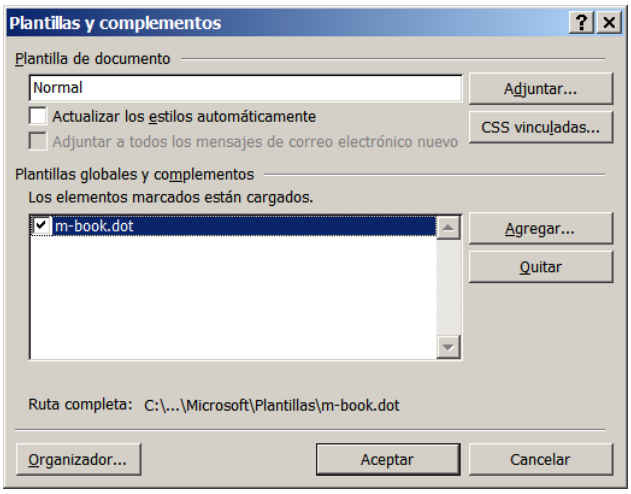
Es posible que MATLAB no pueda encontrar automáticamente el directorio donde se encuentra la plantilla **`Normal.dot`**. En ese caso, aparecerá un cuadro de diálogo a fin de seleccionar el directorio donde se encuentra la plantilla **`Normal.dot`** (por ejemplo, en `C:> \Documents and Settings \ Administrador \ Datos de Programa \ Microsoft \ Plantillas`).

Una vez localizada, la configuración se habrá completado,

```
Notebook setup is complete.
```

y un fichero de nombre **`m-book.dot`** habrá aparecido en el mismo directorio donde se encuentra la plantilla **`normal.dot`**

A continuación, hay que abrir el *Word* y seleccionar `Herramientas → Plantillas y complementos…` (`Tools → Templates and Add-ins…`) a fin de añadir la plantilla global **`m-book.dot`**. Notar que en la barra de menús de *Word* habrá aparecido la opción **`Notebook`**. Ver que las opciones son: `Define Input Cell`, `Evaluate Cell`, etc.



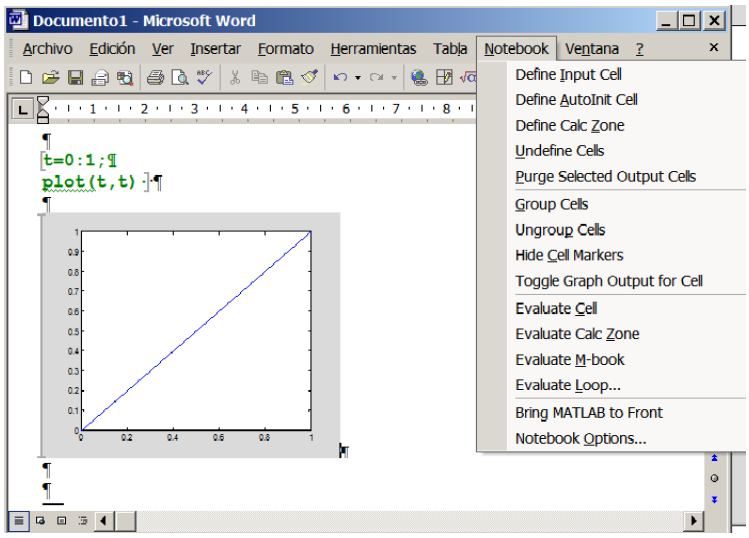
Iniciar el notebook: Si se teclea `>>notebook` en la ventana de comandos de MATLAB aparece el siguiente mensaje

```
>> notebook
Warning: MATLAB is now an automation server
>>
```

y se abre el Word. Si el MATLAB está cerrado, abrir el Word, añadir la plantilla m-book.dot y ver que aparece la opción de menú Notebook.

Uso del notebook: Escribir algunas instrucciones de MATLAB en el documento Word. Seleccionarlas y con ayuda de las opciones de Notebook, definirlas como Input Cell. A continuación evaluarlas (Evaluate Cell).

Aparecerán los mensajes "MATLAB Automation Server Startup" (se abrirá el MATLAB) y "MATLAB computing". Finalmente, se ejecutarán las instrucciones tecleadas:



Referencias

[1] D. Hanselman and B. Littlefield, *Mastering MATLAB 7*, Pearson Prentice Hall, International Edition, 2005.
[2] Documentación de Matlab, The MathWorks.
[3] Manual de instrucciones PCL-812-PG, Advantech.

Apéndice: TAD PCL-812-PG

Diagrama de bloques: La siguiente figura muestra el esquema de bloques de la tarjeta de adquisión de datos (TAD):

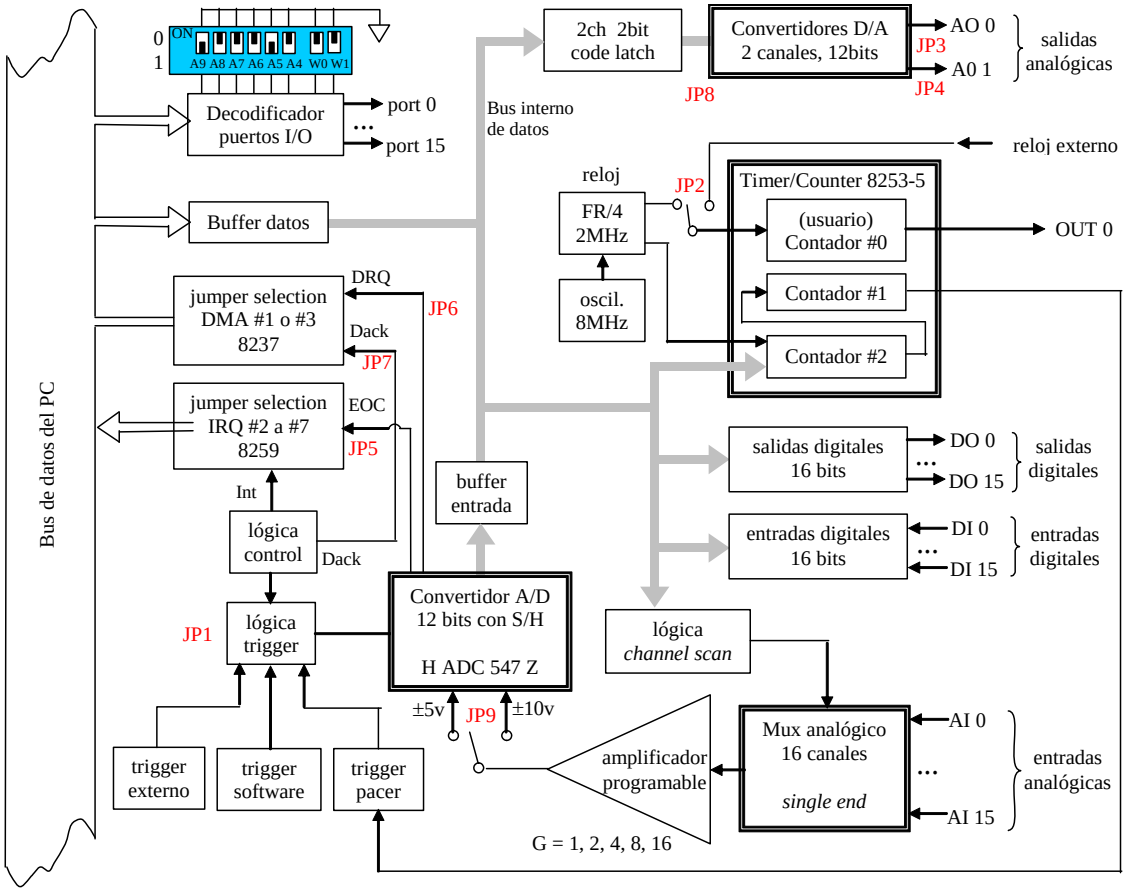


Fig. 1. Diagrama de bloques de la PCL-812-PG

La tarjeta se puede usar para realizar DDCs (*digital direct control*) e identificaciones *on-line* por ordenador sobre diversas plantas analógicas. Por ello, nos centraremos en las etapas de conversión D/A y conversión A/D y no tendremos en cuenta ni las entradas ni las salidas digitales. También es importante la sincronización del conjunto para permitir el procesado en tiempo real.

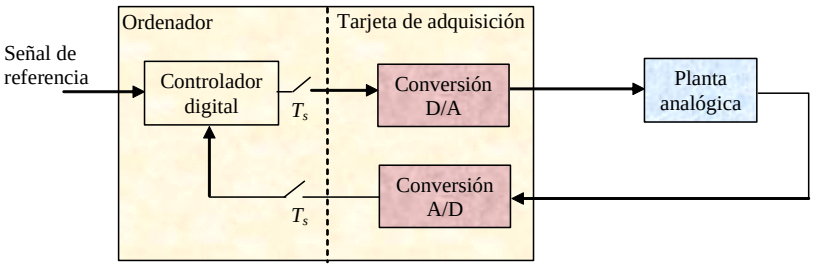


Fig. 2. Hardware on the loop

Mapa de direcciones de los puertos E/S:

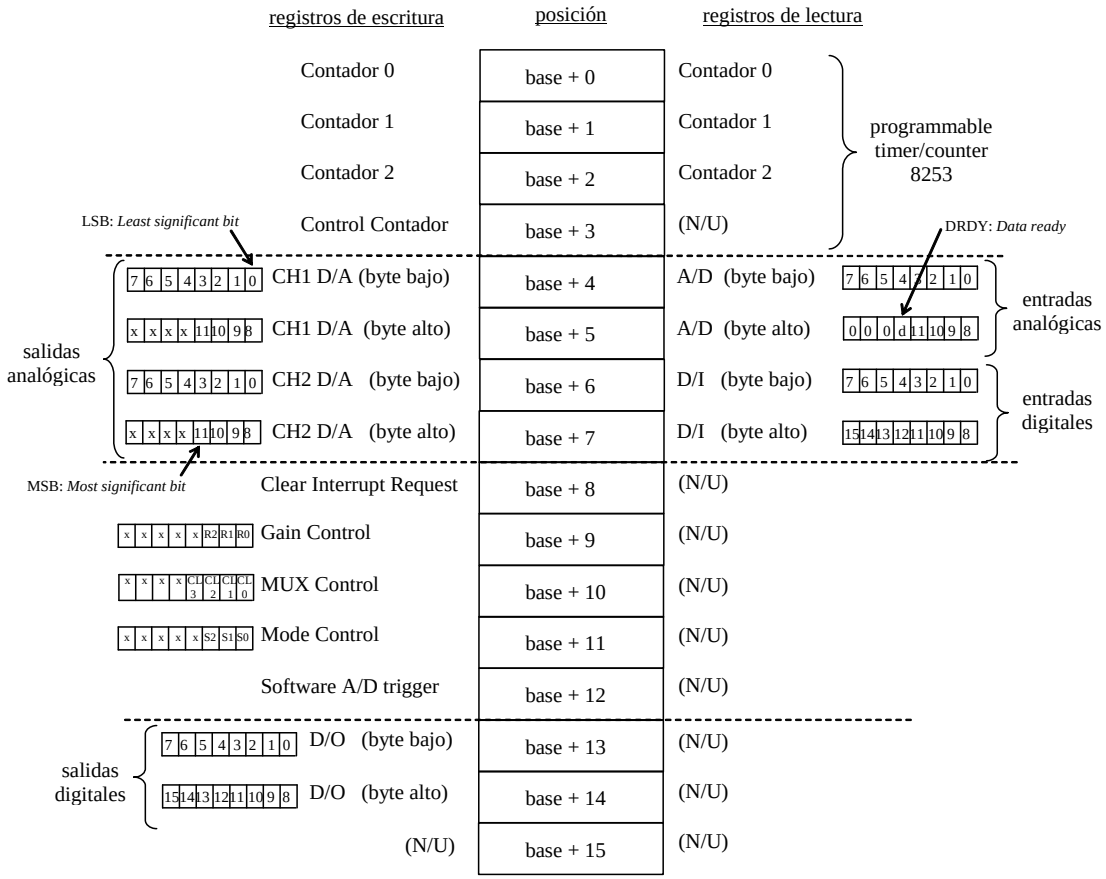
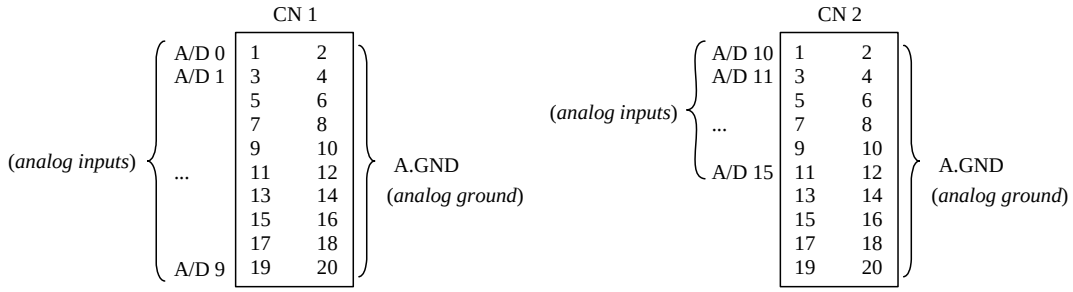


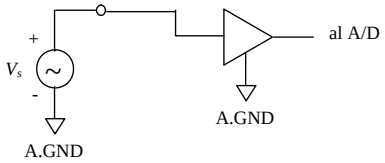
Fig. 3. Mapa de direcciones

El procedimiento de conversión A/D consta de los siguientes pasos:
1) la adquisición y eventual amplificación de la señal analógica,
2) la generación del disparo que activa la conversión A/D propiamente dicha y
3) la transferencia de los datos ya digitalizados al bus de datos del PC.

Adquisición de la señal analógica: Hay 16 canales de entrada de señales analógicas (A/D 0 a A/D 15), repartidos entre los conectores CN1 (canales 0 a 9) y CN2 (canales 10 a 15).



La configuración es *single-ended* (un único cable de señal por canal) y la fuente es flotante (no tiene masa local).



Todos los canales de entrada analógica van a parar a un multiplexor analógico *single-end*. La salida del multiplexor (es decir, el canal activo) la controla un bloque lógico *channel scan*. El número del canal activo se escribe en los bits CL3, CL2, CL1, CL0 del registro *MUX Control* (registro base+10). Este registro es de escritura y no se usa para lectura.

Amplificación de la señal analógica: La señal del canal analógico de entrada escogido pasa por un amplificador de ganancia programable. La ganancia puede ser 1, 2, 4, 8 o 16 y se selecciona escribiendo en los tres últimos bits (R2, R1, R0) del *gain control register* (registro base+9).

R0	R1	R0	ganancia
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	no válido
1	1	0	no válido
1	1	1	no válido

Una vez amplificada, la señal pasa al convertidor HADC 574 Z, que es el encargado de realizar la conversión A/D cada vez que se produce un disparo.

Generación del disparo: La conversión se pone en marcha en el momento que se produce un disparo (*trigger*). El disparo puede ser interno o externo según esté seleccionado el *jumper* JP1.

Conversión A/D: La conversión A/D realiza el circuito integrado HADC 574 Z. Este convertidor tiene dos posibles rangos de entrada seleccionables mediante el *jumper* JP9: ±5v y ±10v. En el primero se supone que las señales en los canales analógicos de entrada presentarán un rango de ±5v, ±5/2v, ±5/4v, ±5/8v o ±5/16v. En el segundo

caso, se presupone que los rangos de entrada serán ±10v, ±10/2v, ±10/4v, ±10/8v o ±10/16v. (No se recomienda usar el rango ±10V si la fuente de alimentación del PC no llega a V_{cc} = +12V ya que en ese caso la salida del amplificador no podrá alcanzar los 10V y la conversión no se hará sobre el valor correcto).

La conversión se realiza por aproximaciones sucesivas y la velocidad máxima de conversión (que se da en el modo DMA) es de 30kHz. El resultado de la conversión se guarda en los registros base+4 (byte bajo) y base+5 (byte alto) y el bit DRDY (*data ready*) del registro base+5 se pone a 0, para indicar que la conversión ha finalizado.

Transferencia de datos: Consiste en copiar los datos de los registros A/D (registros base+4 y base+5) a la memoria del ordenador. Existen tres modos posibles:

Por control del programa: El *jumper* JP4 debe estar en la posición X. Se efectua un *polling*. Después del disparo, el programa de aplicación lee el bit DRDY. Si éste vale 0 (conversión finalizada), el programa de aplicación mueve los datos convertidos de los registros A/D (registros base+4 y base+5) a la memoria del ordenador.

Por rutina de interrupción: Antes de nada hay que especificar el nivel de interrupción (JP5), el vector de interrupción, el controlador de interrupción (8259), el bit de control de interrupción (base+11) y el segmento de memoria a dónde irán los datos A/D. Cada vez que finaliza una conversión, el cambio de valor del bit DRDY a 0 genera una interrupción que hace que el gestor de interrupciones copie los datos A/D al segmento de memoria preespecificado. Si se escribe cualquier cosa en base+8 se resetea la petición de interrupción y se reactiva la interrupción

Por acceso directo a memoria (DMA, direct memory access): Antes de nada hay que especificar el nivel de DMA (JP6, JP7), el bit de activación del registro de control (base+11) y los registros del controlador DMA 8237. Este modo mueve los datos A/D del hardware a la memoria del PC sin que influya la CPU del sistema. Es útil para conseguir una alta velocidad de transferencia de datos pero su uso es complicado.

El registro mode control (base+11) selecciona tanto el modo de disparo como el modo de la transferencia de datos

Configuración del hardware de la conversión A/D:

- Con el *switch* SW1 se selecciona la baseAddress (0x220) y el wait state.
- Con el *jumper* JP1 se selecciona la fuente de disparo (interna).
- Con el *jumper* JP2 se selecciona el reloj para el contador de usuario (interno: 2MHz).
- Con el *jumper* JP5 se selecciona el nivel de interrupción.
- Con los *jumpers* JP6 y JP7 se selecciona el nivel de DMA (Nivel 3)

- Con el *jumper* JP9 se selecciona la tensión de entrada máxima al ADC (rango 10V)
- Se realiza la conexión física con la planta (conector CN1 para canales del 0 al 9 y conector CN2 para canales del 10 al 15)

Configuración del software para la conversión A/D:

- Escribir en los bits R2,R1,R0 del el *gain control register* (base+9) el factor de amplificación de la señal.
- Escribir en los bits CL3,CL2,CL1,CL0 del *mux control register* (base+10) el número del canal de entrada activo.
- Escribir en los bits S2,S1,S0 del *mode control register* (base+11) el modo de disparo y el modo de transferencia de datos.

Conversión D/A: Con el *jumper* JP8 se selecciona el rango de salida del DAC: De 0 a 10v (la referencia es –10V).

Nota: los bytes bajos son double-buffered (es decir, cuando se escribe en las direcciones 4 y 6, los datos se guardan en un buffer. Y mientras se escribe en las direcciones 5 y 7, los datos de 4 y 6 se mandan al DAC al mismo tiempo que los datos de 5 y 7.