Advanced
Micro
Tools

http://www.amctools.com

VMLAB Programming Interface Manual

Part I
User-defined Components

Part II
AVR™ Microcontrollers and Peripherals

A hands-on guide to develop your own VMLAB hardware components and micro-peripherals

---

Table of Contents

Part I: User-defined Components Programming Interface

Prerequisites

In order to develop user-defined components, the following tools and skills are necessary:

- A Win32 C++ compiler. If you do not have one installed in your computer, there are some of such compilers available as freeware in the Web. We recommend Borland C++ 5.5 command line toolkit (BCC55). Download it from: http://www.borland.com/products/downloads/download_cbuilder.html. VMLAB provides a utility, "usercomp.exe", which is a minimum Win32 IDE especially designed for this task. Its default settings are ready to work with BCC55, although it can be configured for any other command-line tool.
- A Win32 resources compiler, if your component has an associated control window. Available also as free download as above. BC55 does include it.
- Some basic understanding of the so-called "callback" programming technique. Callback programming implies to write functions that you never call. It is your partner software, VMLAB in this case, who will call them when needed.
- Basic C programming skills. Although the involved source file has to be compiled as C++, the source code aspect can be 100% C-style. There is no need to define classes, etc.

What components can be developed

Any analog/digital subsystem, with the following restrictions:

- Infinite impedance in inputs, either analog or digital. Current flow through pins is not allowed in this release.
- Analog outputs will be clamped to the VDD - VSS values.
- Analog outputs cannot be a function of analog inputs at a given time.

The above restrictions do not derive from the impossibility to simulate them in VMLAB, but from the fact that it would be needed a too complex user interface. Therefore, in a first release of this Programming Interface such features have not been implemented.

Components do not mean only true hardware models; a great variety of other tools and goodies can be also developed, like:

- Virtual instruments.
- Digital pattern generators based on special file formats.

- Ad-hoc tools for debugging purposes, like establishing complex signal-based breakpoints.
- File recorders.
- Ad-hoc interactive panels.

The basics of a user-defined component

A VMLAB user-defined component is a Win32 DLL, compiled and linked from a C++ source file in which you have to write certain **Callback Functions (C.F.)**. VMLAB will call such functions upon determined simulation events, establishing the component behavior.

The template C++ source file to fill up such functions it is automatically generated by the "usercomp.exe" utility. Callback Functions have a common naming style, **On_xxx(..)**, being "xxx" the event producing the function call. Examples: On_create(..), On_destroy(..), On_digital_in_edge(..)

Within Callback Functions you can invoke **Interface Functions (I.F)**, for example, to set logic values at pins, read voltages from pins, etc. Interface Functions are named uppercase and normally have the form **SET_XXX(..) / GET_XXX(..)**. Examples: SET_LOGIC(..), GET_VOLTAGE(..), etc.

Both the above C.F. and I.F., need to identify the component I/Os to address them in the proper way. There are some **macros** to do this and some other jobs. Example: DECLARE_PINS, ANALOG_IN(..), etc.

How to place a user-defined component in the VMLAB Project File? Just use the underscore '_' before the name, and place it as a macro-model (X card). For a DLL file called "abc.dll", the placement line in the .PRJ file will be:

```
XinstName _abc(10 20) PB1 PB2 PB3 PB4 ; My own component
```

This component has 4 I/O and 2 parameters. "instName" is an arbitrary instance name.

User-defined component DLLs must be placed either in the current working directory or in a general library directory: **<VMLAB install dir>\userlib>.** It is recommended to use this last directory, in order to have access from all the projects.

The basic structure of the C++ code

The C++ file must have the following structure:

- Header, comprising the Win32 selected include file, an own VMLAB header called "blackbox.h", and the DLL entry function; the equivalent to main( ), but for DLLs: DllEntryPoint(..).
- Pins declaration area, identified with macros DECLARE_PINS / END_PINS

- Global variables declaration area DECLARE_VAR / END_VAR. See at a later point why and when it is necessary to declare global variables in this way.
- Window specification macro USE_WINDOW(..). At this point is where you define whether or not the component will have an associated window. Values can be zero, for no window, or a Win32 Dialog resource ID (WINDOW_USER_x).
- Callback functions related code. Some of these functions must be present, even if contain no code; some others can be omitted

See some code examples at the end of the document.

The "blackbox.h" header file

Part of the VMLAB User-defined Components Interface functionality is implemented in this header. It provides a user-transparent way to declare pins, automatic instance handling, etc., that otherwise would result tedious.

Likewise, the "blackbox.h" file, provides some "typedef 's" and "#define's" especially oriented to hardware management. C.F. and I.F. use such types in its definitions.

- PIN: identifies a pin declared in DECLARE_PINS / END_PINS.
- LOGIC: a digital logic value. It can be 0, 1, UNKNOWN and TOGGLE.
- EDGE: identifies a signal edge in a digital pin. It can be RISE or FALL.

In addition, the C++ code uses also some standard Win32 types, like HWND (window handle). Therefore it is always necessary to include the relevant Win32 header file, usually <windows.h>. We refer to some Win32 API manual for information about this subject.

The "blackbox.h" file is located at "<VMLAB_install_dir>\bin", therefore you must either invoke the full path in the source code, or use the brackets directive, #include <blackbox.h>, provided that the mentioned path is present in the compiler “include” path list.

Declaring I/Os

The component I/O pins have to be declared as C++ identifiers in a special section enclosed by the macros DECLARE_PINS / END_PINS, as in this example:

```
DECLARE_PINS
   ANALOG_IN(AIN1, 1);
   ANALOG_IN(AIN2, 2);
   DIGITAL_OUT(DOUT1, 3);
   DIGITAL_OUT(DOUT2, 4);
END_PINS
```

This is a component with two analog inputs identified as AIN1, AIN2, and two digital outputs identified as DOUT1, DOUT2. The numbers identify the node placement sequence in the VMLAB project file.

If the component DLL is MyCell.dll, the placement in the Project File will be:

```
;          AIN1 AIN2 DOUT1 DOUT2  pin names
;
```

```
X1 _MyCell  PB1  PB0  PC1   PC2 ; node names
```

Therefore, the pin AIN1 is connected to the node PB1, pin AIN2, to the node PB0, etc.

Following are the available declaration macros for analog inputs/outputs, and digital inputs/output/bidirectional pins:

ANALOG_IN(<pin identifier>, <number>)
ANALOG_OUT(<pin identifier>, <number>)
DIGITAL_IN(<pin identifier>, <number>)
DIGITAL_OUT(<pin identifier>, <number>)
DIGITAL_BID(<pin identifier>, <number>)

Declaration rules:

- <pin identifier> must be a valid C++ identifier.
- <number> must be a positive number. Note that the maximum <number> will be the number of nodes needed in the cell placement.
- <number> can be zero. This has a special meaning. Pins declared as 0, mean unconnected. This can be useful for component debugging or maintenance. Like this, it is possible to void a pin without the need to re-arrange the source code or the cell placement in the .PRJ file.
- <number> cannot be repeated. Two pins cannot share the same sequence placement. Even if the compiler won't show any error, VMLAB will flag it at load time.
- There can be "missing numbers" in a sequence. For example, you can declare a two pins component, with the sequence #4 and #2. Thus, the placement will need 4 nodes, in which #1 and #3 are considered unconnected. This can be useful for debugging/maintenance (like the pin #0).

Declaring and using variables

A special macros set is provided to define C++ global/static variables: DECLARE_VAR /END_VAR. Why that? Is it not possible to define a global, static storage variable in the standard way, like this?

```
int My_voltage;
```

Yes, it is possible; the problem arises just if you place multiple instances of the component, since all those instances will share the same "My_voltage" variable. This can be unacceptable if such variable contains, for example, a voltage that is strictly related to the instance. A "My_voltage" per instance is needed.

To insure a copy of each global variable per instance, declare your variables in the DECLARE_VAR / END_VAR area:

```
// A set of variables by instance is automatically kept
DECLARE_VAR
   int My_voltage;
   int My_array[10]
   MyStruct *My_pointer;
END_VAR
```

C/C++ initializers are not allowed within DECLARE_VAR / END_VAR.
All variables are automatically initialized to zero at component creation, but not at simulation start, therefore this task is user's responsibility.

To refer to variables defined in DECLARE_VAR / END_VAR, use the **VAR(..) macro**.

Examples:

```
VAR(My_voltage) = 3.25;
J = VAR(My_array)[3];
VAR(My_pointer)->my_member = 10;
```

Caution !
C.F. (On_create(..), On_destroy(..) , etc.), are called for each instance. If you declare variables outside DECLARE_VAR / END_VAR, pay attention to initialize pointers, files etc, and NULL them after deletion, otherwise, a delete of previous deleted pointer, etc, could occur, causing a probable access violation fault. Example of correct code:

```
FILE *My_file = NULL;   // Initialize always

const char *On_create() {
   if(!My_file) { // No problem if called several times
      My_file = fopen("myfile.dat", "rt");
   }
   return NULL;
}

void On_destroy() {
   if(My_file) {  // No problem if called several times
     fclose(My_file);
     My_file = NULL;
   }
}
```

Callback Functions (C.F.)

The last part of the C++ source code must be the C.F. implementation. VMLAB will call automatically such functions during the simulation flow. They are the placeholders for the code that describes the component behavior.

Some functions must be defined, even if contain no code; some others are optional and can be omitted if not used. "usercomp.exe", provides placeholders for all functions, mandatory or not. Later on, non-mandatory ones can be optionally deleted.

List of C.F. prototypes in alphabetical order. Names in **bold** are mandatory functions: the compiler/linker will give an error if no present.

```
void On_break(BOOL pStatus);
const char *On_create();
void On_destroy();
void On_digital_in_edge(PIN pDigIn, EDGE pEdge, double pTime);
void On_gadget_notify(GADGET pGadgetId, int pCode);
void On_remind_me(double pTime, unsigned long pData);
void On_simulation_begin();
void On_simulation_end();
void On_time_step(double pTime);
double On_voltage_ask(PIN pAnalogOut, double pTime);
void On_window_init(HWND pWindow);
void On_update_tick(double);
```

Interface Functions (I.F.)

Within C.F., the way to define the hardware behavior, like setting pin logic values, reading/setting voltages, as well as some other miscellaneous tasks, and tracing/debugging goodies, is to use this set of functions.

To distinguish them from the C.F., they are named with FULL UPPERCASE.

List of I.F. prototypes, classified by type:

Data retrieving:

```
HWND GET_HANDLE(int pGadgetId);
const char *GET_INSTANCE();
LOGIC GET_LOGIC(PIN pPin);
double GET_PARAM(unsigned long pIndex);
double GET_VOLTAGE(PIN pPin);
double GET_CLOCK();
BOOL GET_DRIVE(PIN pPin);
double POWER();
double TEMP();
```

Data setting:

```
void SET_DRIVE(PIN pPin, BOOL pEnable);
void SET_LOGIC(PIN pPin, LOGIC pValue, double pDelay = 0);
void SET_VOLTAGE(PIN pPin, double pValue);
```

Debugging and miscellaneous:

```
void BREAK(const char *pMessage = NULL);
void PRINT(const char *pText);
void REMIND_ME(double pDelay, int pParam = 0);
void TRACE(bool pEnable);
```

See at Interface Functions Reference a detailed description of each function.

The Callback Functions sequence

Following is a typical sequence of how VMLAB calls these functions. Let’s assume a time step of 1us, although this is determined automatically.

On Project File parsing, (command "Build"):

- On_create()
- On_window_init(..) If window defined

On simulation start, (command "Go"):

- On_simulation_begin()
- On_time_step(..)   For Time = 0

... For Time = 1.0e-6

- On_voltage_ask(..) If applies
- On_time_step(..)   Always
- On_digital_in_edge(..) If applies
- On_remid_me(..)   If applies
- On_gadget_notify(..) If applies

... For Time = 2.0e-6

- On_voltage_ask(..) If applies
- On_time_step(..)   Always
- On_digital_in_edge   If applies
- On_remid_me(..)   If applies
- On_gadget_notify(..) If applies

...

On command "Restart Simulation":

- On_simulation_end(..)

On “Project File | Rebuild or Close”:

- If present, the window interface is destroyed (no special notification)
- On_destroy(..)

The time step is automatically calculated by VMLAB, mainly as function of the micro-controller clock rate. If a particular time step needs to be forced, use REMIND_ME(..) function.

The C.F. On_updated_tick( ) is called at a regular time intervals, oriented to update visual controls (see more in the C.F. Reference)

Callback Functions Reference

On_break( )

```
void On_break(BOOL pStatus);
```

This C.F. is received when the micro instruction flow is stopped by some reason: breakpoint, warning, user stop, etc (then, the ‘pStatus’ pararameter will be TRUE) or either, when the instruction flow is resumed, due to the user clicked the ‘Go/Continue’ button (‘pStatus’ parameter will be FALSE).

Note that once On_break(true) has been received, there can be other notifications while the instruction flow is stopped; for example, if the user performs some actions that involve a re-calculation of the electrical circuit conditions. Of course, during a break, all parameters containing the simulation time will show that the value does not change.

Example:

```
void On_break(BOOL pValue) {
   if(pValue)
      PRINT(“Let’s take a break”);
   else
      PRINT(“Back to work”);
}
```

On_create( )

```
const char * On_create();
```

VMLAB notifies that component needs to be created. This function is called when the command "Project | Build" is invoked.. The function return must be either NULL, for a successful creation, or an error message, if something wrong prevents the component behavior to be OK. Such message will be displayed as an error in the Messages Window.

Typical uses:

- Open and read a data files.
- Allocate memory, objects, etc. (malloc( ), new, etc).
- Component initialization, provided it is a forever initialization. Simulation dependant initialization must be performed at On_simulation_begin.(..).
- Parameters consistency check.

Do not use On_create(..) to initialize pins logic/voltage values. Use instead On_simulaton_begin(..).

Example:

```
char Data_table[256];  // All instances share these
FILE *My_file = NULL;  // variables!

const char *On_create()
//**********************
{
   for(int j = 0; j < 256; j++)
      Data_table[j] = (char)j;    // Fill data table

   if(!My_file) {
      My_file = fopen("datafile.dat", "rt");
      if(!My_file)
         return "Unable to open data file"; // Error
   }

   if(GET_PARAM(1) < 1)
      return "Invalid parameter"); // Wrong component
   else
      return NULL;  // Successful creation

}
```

On_window_init( )

```
void On_window_init(HWND pHandle)
```

VMLAB will notify with this function just in case the component has an associated window (USE_WINDOW(..) not zero). The parameter "pHandle" is the main component window handle; use this handle if you need to hook an own Windows structure, (VCL, MFC,…), setting your own windows as children.

Typical use: initialize window controls/gadgets. Usually, coding tasks related to this function require certain knowledge of the Win32 API.

Example:

```
void On_window_init(HWND pHandle)
//********************************
// Fill a ListBox control with 4 Strings. ListBox
// default style is to be sorted.
{
   static const char *myText[] = {"A", "C", "B", "D"};
   HWND lBox = GET_HANDLE(GADGET8);
   for(int j = 0; j < 4; j++) {
      SendMessage(lBox, LB_ADDSTRING, 0,(LPARAM)myText[j]);
   }
}
```

Caution !

Do not perform any window related operation within On_create(..), since at that time the window has not yet been created by VMLAB, therefore any attempt to use GET_HANDLE(..) will cause an error message to be displayed and a NULL return .

On_destroy( )

```
void On_destroy();
```

VMLAB calls it after closing the project, or if by any reason the Project File needs to be rebuilt, due to a modification. On_destroy( ), gives a chance to undo the operations performed in On_create(..), like closing files, free memory, delete objects, etc. Like in On_create( ), the window handle, if created, is no longer available, therefore, a GET_HANDLE(..) would return NULL.

Example:

```
void On_destroy()
//***************
{
   if(My_file) {
      fclose(My_file);
      My_file = NULL;
   }
}
```

On_simulation_begin( )

```
void On_simulation_begin();
```

Called when the simulation is starting, at Time = 0. Typical uses:

- Component initialisation: I/Os logic levels, internal variables, etc.
- Open data files that can change with the simulation, due to parameters, etc. If the file is fixed regarding the simulation, this job could be done at On_create(..).
- Launch simulation time ticks, by using the function REMIND_ME(..).
- Allocate memory (‘new’ objects, malloc(..), etc) that are simulation dependant.

Example:

```
void On_simulation_begin()
//************************
{
   VAR(Average_voltage) = POWER() / 2;
   VAR(Data_array) = calloc(1024, sizeof(char));
   SET_LOGIC(OUT1, 0);
   REMIND_ME(10.0e-3);
}
```

On_simulation_end( )

```
void On_simulation_end();
```

Called after a "Restart" command. It provides a chance to undo the operations performed at On_simuation_begin(..), like free memory, delete objects, close files, etc.

Example:

```
void On_simulation_end()
//*********************
{
   free(VAR(Data_array));
}
```

On_digital_in_edge( )

```
void On_digital_in_edge(PIN pDigIn, EDGE pEdge, double pTime);
```

Reports an edge occurrence at a digital input pin at time = "pTime". The pin "pDigIn" must have been declared as DIGITAL_IN(..). "pEdge" will come with values either RISE or FALL.

Example:

```
void On_digital_in_edge(PIN pDigIn, EDGE pEdge, double pTime)
//***********************************************************
// CLOCK_PIN needs to be delared as DIGITAL_IN
{
   LOGIC dataIn;
   if(pDigIn == CLOCK_PIN && pEdge == RISE) {
      dataIn = GET_LOGIC(DATA_PIN);
   }
}
```

On_voltage_ask( )

```
double On_voltage_ask(PIN pAnalogOut, double pTime)
```

This function applies to analog outputs. VMLAB is asking to return the analog outputs voltage that must behave in a continuous-time way. Unlike SET_VOLTAGE(..) that would produce sharp voltage step-like variations, returning a value in On_voltage_ask(..) will cause continuous time waveforms, as a function of time.

The parameter "pAnalogOut" will provide a pin identifier that must have been declared as ANALOG_OUT(..). There is a special return value: KEEP_VOLTAGE, that is interpreted by VMLAB as to keep the previous voltage.

SET_VOLTAGE(..) and SET_LOGIC(..) are not allowed within this function; a runtime error will be flagged.

Example:

```
DECLARE_PINS
   ANALOG_OUT(SINE_OUT, 1);
   ANALOG_OUT(RAMP_OUT, 2);
   ...
END_DECLARE
...
double On_voltage_ask(PIN pAnalogOut, double pTime)
//*************************************************
//   SINE_OUT: 1KHz, 1Vp-p centered sinewave
//   RAMP_OUT: 1ms ramp; stay after
//
{
   const double K = 2 * 3.1416 * 1.0e3;
   switch(pAnalogOut) {
      case SINE_OUT:
        return POWER() / 2 + sin(K * pTime);

      case RAMP_OUT:
         if(pTime < 1.0e-3)
            return POWER() / 1.0e-3 * pTime;
   }
   return KEEP_VOLTAGE;
}
```

On_time_step( )

```
void On_time_step(double pTime);
```

VMLAB reports that the simulation time up to parameter "pTime" has been reached. This is the place where to read voltages, logic values, etc.

VMLAB calculates automatically the simulation step in function of the microcontroller clock speed, etc. A way to force a time-step by the user is to use the REMIND_ME(..) function.

Tips
The response to this function at "pTime = 0" can be useful for certain kind of initializations, since it is the first time that input pins voltages/logic are known.

Example:

```
void On_time_step(double pTime)
// ***************************
// IN_PIN must be declared as DIGITAL_IN
{
   LOGIC myLogic;
   if(pTime == 0) {
      // First time that IN_PIN value is known
      myLogic = GET_LOGIC(IN_PIN);
   } else {
      ...
   }
}
```

On_remind_me( )

```
void On_remind_me(double pTime, int pData);
```

Response to a previous I.F. REMIND_ME(..) call. On_remind_me(..) can hold as well REMMID_ME(..) calls. Such technique allows producing simulation time ticks at a fixed or variable interval for whatever purpose. The parameter "pData" will hold the value passed in the previous REMIN_ME(..).

Example:

```
void On_remind_me(double pTime, int pData)
// ***************************************
// A tick every 10 ms
{
   REMIND_ME(10.0e-3); // A 10 ms tick
}
```

On_gadget_notify( )

```
void On_gadget_notify(GADGET pGadgetId, int pCode);
```

This C.F. only applies to components with an associated window. The parameter pCode will hold the Win32 notification code from the corresponding control; pGadgetId will hold the control ID (GADGET0 to GADGET31), as defined in the resources file (.RC). The use of this function presumes certain knowledge of the Windows API.

Example:

```
void On_gadget_notify(GADGET pGadgetId, int pCode)
// ***********************************************
// Example of button click based action. The button
// is identified as GADGET1 in the RC file
{
   if(pGadgetId == GADGET1 && pCode == BN_CLICKED) {
      SET_VOLTAGE(OUT_PIN, 0); // Set 0V on button click
   }
}
```

On_update_tick( )

```
void On_update_tick(double);
```

If defined, this C.F. is invoked regularly every ~80 msec (approx), and it is intended to refresh windows controls, etc, providing a half-standard animation rate (~12.5 times per second). The actual simulation time is passed as a double parameter.

The major use of this function is to optimize the simulation, by refreshing controls at a reasonable timing rate, in such way that the user perceives an animation effect, but the simulation performance is not affected too much.

Notes:

- This notification is received only if there is some window associated with the component, and even during breaks.
- The elapsed time between two calls is approximate. This function should not be used for accurate time measurement, nor to produce high quality animations.

Example:

```
void On_update_tick(double pSimTime)
{
   if(Value_changed_since_last_refresh) {   // If no need, no refresh
      char myBuffer[32];
      sprintf(myBuffer, “Value = %d”, Value);
      SetWindowText(GET_HANDLE(GADGET1), myBuffer); // Refresh control
      Value_changed_since_last_refresh = false;
   }
}
```

Interface Functions Reference

BREAK( )

```
void BREAK(const char *pMessage = NULL);
```

Forces the simulation to stop. The stop point will be the next microcontroller instruction boundary. An optional message, "pMessage" parameter, can be displayed in the Messages Window.

As BREAK( ) will be effective in the next micro instruction, bear in mind that this can take some time if it is issued during the RESET phase.

Example:

```
void On_time_step(double pTime)
//****************************
{
   ...

   BREAK("Stop here!");  // Request to stop the simulation
   . . .
}
```

GET_CLOCK( )

```
double GET_CLOCK();
```

Returns the actual microcontroller clock value in Hertz.

GET_DRIVE( )

```
BOOL GET_DRIVE(PIN pPin)
```

This function, an inverse of SET_DRIVE( ), returns the current driving state for a pin or port: ‘true’ if it is an output, and ‘false’ in case of input.

Note that its use is unnecessary in normal user components, since the DLL code must know at any time, when a bidirectional pin is input or output.

However, in micro ports is different, since the input/output control may come from different sources. See Part II.

GET_HANDLE( )

```
HWND GET_HANDLE(int pGadgetId);
```

In a component with an associated window, gets the window handle corresponding to the gadget ID "pId". "pId" must be one of the GADGETx values (GADGET0 to GADGET31), which has been assigned to the standard Windows controls placed in the resource file.

Use this to execute Win32 API commands in standard controls: ListBox, ComboBox, etc. present in the component window. The use of this function presumes certain knowledge of the Windows API. Example:

Even if the component has an associated window, GET_HANDLE( ) will return NULL if called within the On_create( ) or On_destroy( ) C.F, in the first case because the window has not yet been created, and in the second, because it has already been destroyed. A window handle is available for the first time after On_window_init( ) notification.

```
// Add a string to a ListBox type control, identified
// in the resources file (.RC) as GADGET5
//
const char *myString = "Hello World";
HWND lBoxHandle = GET_HANDLE(GADGET8);
SendMessage(lBoxHandle, LB_ADDSTRING, 0, (LPARAM)myString);
```

GET_INSTANCE( )

```
const char *GET_INSTANCE();
```

It retrieves the instance name as specified in the Project File (X<inst_name> ....).

Tip:
Use the instance name to pass a string type parameter to the component, to open a given filename, etc. Example:

```
const char *On_create()
//*********************
// Opens a file passed by instance name
{
   if(!My_file) {
      char fullName[32];
      sprintf(fullName, "%s.dat", GET_INSTANCE());
      My_file = fopen(fullName, "rt");
      if(!My_file) return "File does not exist";
   }
   return NULL; // Success
}
```

GET_LOGIC( )

```
LOGIC GET_LOGIC(PIN pPin);
```

It retrieves the logic value at a pin. "pPin" will contain a pin identifier previously declared as DIGITAL_IN(..), DIGITAL_OUT(..) or DIGITAL_BID(..). The return value will be a LOGIC type one, which can take values 0, 1 or UNKNOWN. Example:

```
DECLARE_VAR
   unsigned char RAM_data[4096];
   int Write_address;
   ...
END_VAR

void On_digital_in_edge(PIN pDigIn, EDGE pEdge, double pTime)
//*************************************************************
// A RAM with falling edge write-enable and write address
// self-increase. Pins WEN, DIN0...DIN7 must be declared!
// Unknown logic values (X) not handled, nor other checking,
// like max address, etc,
{
   if(pDigIn == WEN && pEdge == FALL) {
      LOGIC bit[8];
      bit[0] = GET_LOGIC(DIN0);
      ...
      bit[7] = GET_LOGIC(DIN7);

      unsigned char byteValue = 0;
      for(int j = 0; j < 8; j++)
         byteValue = byteValue * 2 + bit[j];

      VAR(RAM_data)[VAR(Write_address)] = byteValue;
      VAR(Write_address)++;
   }
}
```

GET_PARAM( )

```
double GET_PARAM(unsigned long pIndex);
```

It gets the user placement parameter with the given "pIndex", starting at 1. For example, if this component is placed in the Project FILE (.PRJ) file like this:

```
Xinst1 _MyComponent(3 4 5) node1 node2 node3
```

A call to GET_PARAM(2) will return the value 4.0 (type is double).

Any "pIndex" for which the corresponding parameter does not exist, will return zero.

GET_VOLTAGE( )

```
double GET_VOLTAGE(PIN pPin);
```

It retrieves the voltage at the given pin (parameter "pPin") referred to VSS. Example

```
void On_time_step(double pTime)
// ****************************
// Example of simulation recording in a file
// Log_file must have been open previously
// MY_PIN must be declared ANALOG_IN
//
{
   double v = GET_VOLTAGE(MY_PIN);
   fprintf(Log_file, "T = %f; V = %f\n", pTime, v);
}
```

POWER( )

```
double POWER();
```

Get the power level value, as defined in the Project File (VDD – VSS).

PRINT( )

```
void PRINT(const char *pText);
```

It displays a message in the Messages Window. Use this function for debugging or monitoring purposes.

Example:

```
void On_digital_in_edge(PIN pDigIn, EDGE pEdge, double pTime)
//***********************************************************
{
   if(pDigIn == CLK_IN && pEdge == RISE) {
      PRINT("Rising edge detected at pin CLK_IN");
   }
}
```

Caution!
Pay attention: the Messages Window could be flooded with messages, if PRINT(..) is not properly controlled, mainly in high traffic callback functions, like On_time_step(..).

REMIND_ME( )

```
void REMIND_ME(double pDelay, int pParam = 0);
```

It causes a On_remind_me(..) callback function to be invoked after "pDelay" time in the simulation with the optional auxiliary parameter "pParam". This parameter can be used for whatever purposes are required. Cast into "int" if you wish to pass another parameter type, and cast back to the original type at On_remind_me(..).

Use this function to cause simulation time ticks defined by the component.
A On_remind_me(..) callback function can contain inside REMIND_ME(..) calls. This can be used to define a regular simulation time tick.

Example:

```
void On_simulation_begin()
//************************
{
   REMIND_ME(100.0e-6, 1);
}

void On_remind_me(double pTime, int pData)
//***************************************
{
   PRINT("100 simulation us have passed!");
   REMIND_ME(100.0e-3, pData++); // pData = nr. of ticks
}
```

SET_DRIVE( )

```
void SET_DRIVE(PIN pPin, BOOL pIsOutput);
```

It applies to a bidirectional pin "pPin". SET_DRIVE forces the given pin to behave like an output when the parameter "pIsOutput" is "true".

Example:

```
void On_gadget_notify(GADGET pGadgetId, int pCode)
//************************************************
// Changes pin input <-> output with a button
{
   switch(pGadgetId) {
      case GADGET1:
         SET_DRIVE(BID1, true);
         break;
      case GADGET2:
         SET_DRIVE(BID1, false);
         break;
   }
}
```

SET_LOGIC( )

```
void SET_LOGIC(PIN pPin, LOGIC pValue, double pDelay = 0);
```

It applies a logic value in an output or bidirectional pin (if defined as output). The "pPin" value must have been defined inside DECLARE_PINS as DIGITAL_OUT(..), or DIGITAL_BID(..). The optional parameter "pDelay", specifies the delay at which the logic assignment must be performed. A zero value is assumed if not specified.

Example:

```
void On_simulation_begin()
//************************
// Launches a serial signal over pin SER_OUT.
// SER_OUT must have been declared as DIGITAL_OUT.
{
   const double BIT_TIME = 100.0e-6;
   char txByte = 0xA1;  // Byte to transmit
   for(int j = 0; j < 8; j++) {
      SET_LOGIC(SER_OUT, txByte & 1, j * BIT_TIME);
      txByte >>= 1; // shift
   }
}
```

SET_VOLTAGE( )

```
void SET_VOLTAGE(PIN pPin, double pValue);
```

It causes a voltage to be set in an analog output pin. "pPin" will contain a pin identifier previously declared as ANALOG_OUT. SET_VOLTAGE(..) causes sharp changes in the pin over which it is applied. For a continuous-time analog signal use the callback function On_voltage_ask(..).

Example:

```
void On_time_step(double pTime)
//*****************************
// A digital to analog conversion. Possible undetermined
// bits (X) read not handled!
{
   LOGIC bit[8];
   bit[0] = GET_LOGIC(IN0);
   ...
   bit[7] = GET_LOGIC(IN7);

   int binValue = 0;
   for(int j = 0; j < 8; j++) {
      binValue = binValue * 2 + bit[j];
   }
   SET_VOLTAGE(AOUT, POWER() / 255 * binValue);
}
```

TEMP( )

```
double TEMP();
```

Retrieves the simulation temperature in Kelvin degrees, as specified in the VMLAB Control Panel. Use this function to model temperature dependant features.

Example:

```
void On_simulation_begin()
//***********************
// Some delay value that changes with the temperature
// and power level. TEMP() gives Kelvin deg. (not Celsius!)
//
{
   VAR(Some_delay) = (K0 + K1 * TEMP()) / (K2 * POWER());
}
```

TRACE( )

```
void TRACE(BOOL pEnable);
```

This function is a debug oriented feature. After it has been called as TRACE(true), VMLAB will display in the Messages Window all the Callback Functions and Interface Functions, displaying the relevant parameters. To stop tracing, call the function again with "pEnable" = false.

Use this function with care, since it can flood the Messages Window.

Example:

```
void On_simulation_begin()
//**********************
// Trace the first 10ms of simulation.
{
   TRACE(true);
   ...
}

void On_time_step(double pTime)
//*****************************
//
{
   if(pTime > 10.0e-3) TRACE(false);
   ...
}
```

The Utility "usercomp.exe"

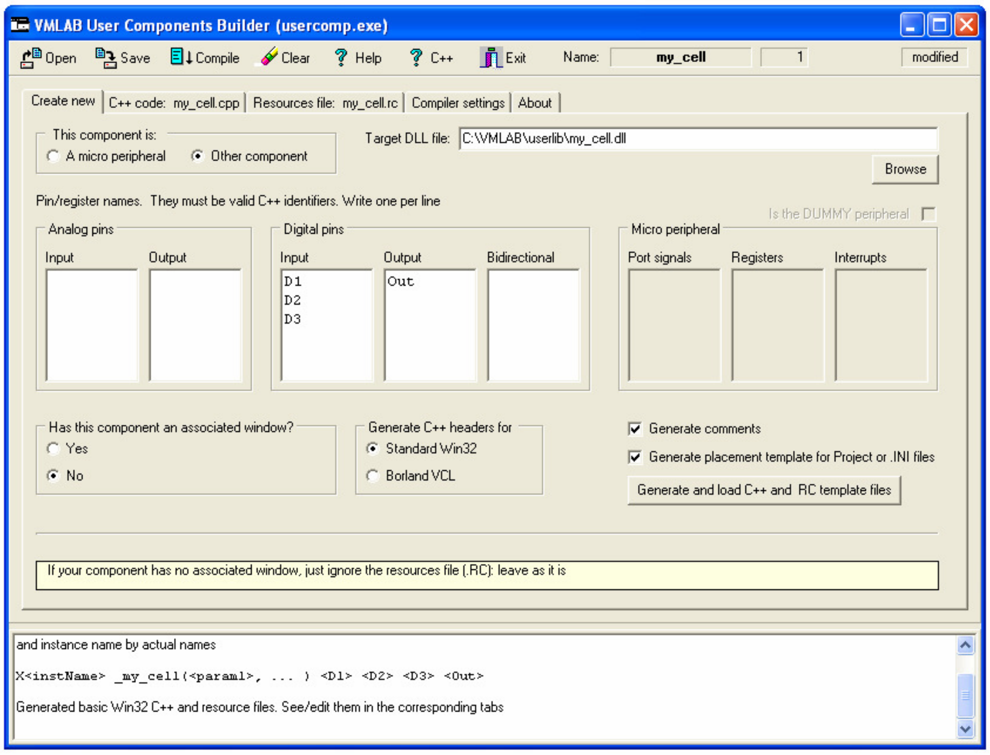
General overview

This is an stand-alone utility that can be run either invoking it directly, or from VMLAB: "Components | Create New" command. It location is "<VMLAB_install_dir>\bin\usercomp.exe".

It provides the following functionality:

- Automatic creation of basic files: C++ source and RC resources file, based on a graphical interface.
- Two tab-based standard editors for the C++ the RC files.
- Automatic Win32 command-line tool spawning, with errors reporting in an included Messages Window.
- User-configurable Win32 toolkit and command-line. Default behavior is to use BCC55.

This utility if designed to develop VMLAB user-defined components, specially for non-Windows programmers, but it use is optional: **you can use any market standard Win32 IDE**: (MS .NET, VC++, Borland C++ Builder, etc.). Just compile and link your component source code as a DLL not as EXE.


VMLAB User Components Builder (usercomp.exe)
Open
Save
Compile
Clear
Help
C++
Exit
Create new
C++ code: my_cell.cpp
Resources file: my_cell.rc
Compiler settings
About
Browse
Generate and load C++ and RC template files

Name: **my_cell** 1 modified

This component is: ( ) A micro peripheral [✓ Other component]

Target DLL file: C:\VMLAB\userlib\my_cell.dll

Pin/register names. They must be valid C++ identifiers. Write one per line

Is the DUMMY peripheral ☐

Analog pins: Input	Analog pins: Output	Digital pins: Input	Digital pins: Output	Digital pins: Bidirectional	Micro peripheral: Port signals	Micro peripheral: Registers	Micro peripheral: Interrupts
		D1 D2 D3	Out				

Has this component an associated window? ( ) Yes [✓ No]

Generate C++ headers for [✓ Standard Win32] ( ) Borland VCL

☑ Generate comments

☑ Generate placement template for Project or .INI files

If your component has no associated window, just ignore the resources file (.RC): leave as it is

```
and instance name by actual names

X<instName> _my_cell(<paraml>, ... ) <D1> <D2> <D3> <Out>

Generated basic Win32 C++ and resource files. See/edit them in the corresponding tabs
```

To create new basic template files for a new component, the "Create new" tab asks the user:

- The I/O pin names.
- If this component has an associated control window or not.
- The kind of Windows headers to be produced: Win32 SDK, or VCL. Use VCL only if you will compile the code with Borland C++ Builder IDE.

Once all these data have been filled in, click the button **"Generate and load C++ and RC basic files"**, and automatically, the application will generate two files, and will load them in two text editor windows.

- A C++ code file.
- A .RC file (Win32 resources file).

The resources file (RC)

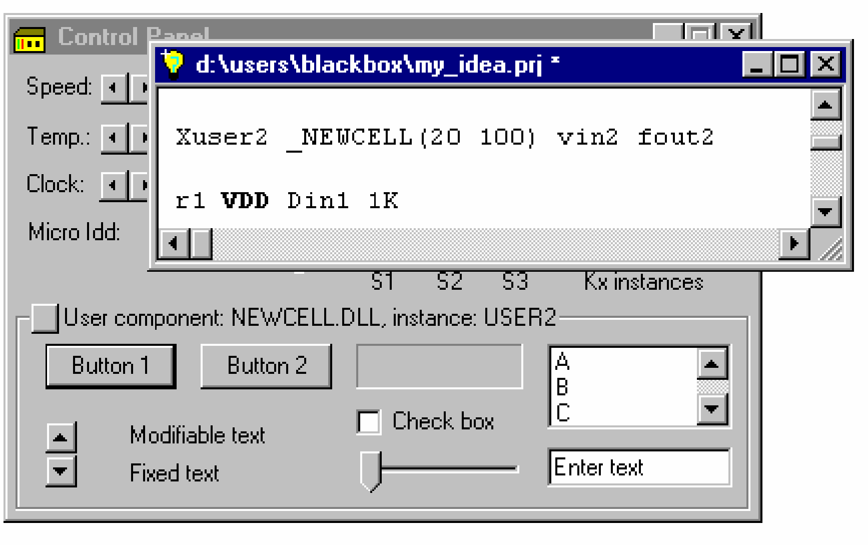
If your component has no associated control window, just ignore the RC file and leave it like is generated; the binary space overhead for this feature is negligible. If you intend to design a control window to be loaded in VMLAB, arrange the RC file, placing the controls according to your needs. Do not modify the main window width; it must fit into VMLAB Control Panel fixed dimensions. It is highly recommended to use some graphical Windows resources editor for this task, although you can do it with the text editor, modifying the control coordinates.

In Windows, every control is identified by a ID number. The "blackbox.h" file provides a set of values to be used: GADGETx (GADGET0 to GADGET31). If another naming is necessary for your code readability, do it like this (example):

```
#define GO_BUTTON GADGET1  // I prefer name GO_BUTTON !
```

Use then the "Compile" command to run the selected Win32 toolkit. Errors will be reported in a special Messages Window. Once all is error-free, your DLL is ready to load into a VMLAB project.

The following picture represent the default window corresponding to the default RC file generated by "userelem.exe". The C++ code must specify the main window dialog ID: WINDOW_USER_1:



```
Xuser2 _NEWCELL(20 100) vin2 fout2

r1 VDD Din1 1K
```

```
...
USE_WINDOW(WINDOW_USER_1);  // Window specification in C++ code
```

Debugging components

The "userelem.exe" does not provide source debugging features. To debug your DLL at source level it is needed some Win32 IDE/debugger.

However, most of the times the included debug oriented functions and features can result sufficient to achieve a successful development. Following is a summary:

- PRINT(..), to display variable values or other kind of messages in VMLAB Messages Window.
- BREAK(..) To stop the simulation.
- TRACE(..), To get a detailed trace of every Callback and Interface functions.
- Exception handling for Callback Functions. If by mistake, you cause an access violation, division by zero, etc. in your DLL code, VMLAB won't crash, but will display a message indicating the function responsible for that fault.

The VMLAB command "Project | Unload components"

Win32 linkers prevent the generation of a given DLL, is such DLL is currently in use by some application, therefore **it is necessary to insure that no user-component DLLs are loaded by VMLAB before attempting to re-build new ones**. The DLL load/unload process it is automatically performed by VMLAB upon Project File parsing / modifying, but however, it may be useful to force the unload of user-defined components. Use then the command "Project | Unload components" (Ctrl + U).

Optimizing performance

Here are some advices that will help to optimize the simulation speed:

- Do not define C.F. if you do not use them, especially those ones who are called in every time step, like On_voltage_ask( ), or On_time_step( ). If such functions are present in your DLL, VMLAB will call them, with the corresponding execution overhead.
- Use the PRINT( ) message for debugging or special purposes. This will slow down the simulation and will also overload the system memory, if used at every simulation step: messages are stored in the Messages Window.
- Use of On_update_tick( ) for Windows Controls update. Note that updating a visual control faster than the human eye can follow, makes no sense, causing a simulation time waste.

A Crash course on Win32 controls

Just if you don't know anything about Windows programming, but you want to implement some simple window-based component. See below some examples of common basic tasks. For a complete reference, see: http://msdn.microsoft.com/library/default.asp

The basics. Windows controls handling is based in a set of constants defined in <windows.h> or <commctrl.h>. Such constants are classified in 3 types:

- **Styles**, to be placed in the RC file to determine some variations/styles in the control.
- **Messages**, to request actions to controls, sending a Windows standard message.
- **Notifications**, when the control has something to say: a status change, etc.

Detecting a Button click or CheckBox change. At On_gadget_notify(..) :
```
if(pGadget == GADGETx && pCode == BN_CLICKED) { ...
```

Setting a text. This is valid for Static and Button controls:
```
SetWindowText(GET_HANDLE(GADGETx), "This is my text");
```

Reading Edit controls. Also valid for Static and Button:
```
char buffer[LEN]; // Preview a maximum length
GetWindowText(GET_HANDLE(GADGETx), buffer, LEN - 1);
```

Detecting changes in Edit controls. At On_gadget_notify(..):
```
if(pGadget == GADGETx && pCode == EN_CHANGE) { ...
```

Setting TrackBar range limits. Variables myMin, myMax: 16 bits integers.
```
HWND handle = GET_HANDLE(GADGETx);
SendMessage(handle, TBM_SETRANGE, true, MAKELONG(myMin, myMax));
```

Detecting movement in a TrackBar. At On_gadget_notify(..) :
```
if(pGadget == GADGETx && pCode == TB_THUMBTRACK) { ...
```

Reading the TrackBar slider position. Within the specified range.
```
int pos;
pos = SendMessage(GET_HANDLE(GADGETx), TBM_GETPOS, 0, 0);
```

Filling a ListBox. This will add a string. Consider if the list style is sorted or not.
```
int pos; // Will return the index, zero based.
pos = SendMessage(GET_HANDLE(GADGETx), LB_ADDSTRING, 0, "My text");
```

Detecting a double click selection in a ListBox. At On_gadget_notify(..):
```
if(pGadget == GADGETx && pCode == LBN_DBLCLK) { ...
```

Setting range limits in a ProgressBar. Variables myMin, myMax: 16 bits int.
```
HWND handle = GET_HANDLE(GADGETx);
SendMessage(handle, PBM_SETRANGE, 0, MAKELPARAM(myMin, myMax));
```

Setting the value in a ProgressBar. Variable myPos is in the specified range.
```
SendMessage(GET_HANDLE(GADGETx), PBM_SETPOS, myPos, 0);
```

Code sample #1: a voltage-to-frequency converter.

This code, V2F.CPP, correspond to the component V2F.DLL, located in <VMLAB_install_dir>\userlib.

```
#include <windows.h>
#include <commctrl.h>
#pragma hdrstop
#include "..\bin\blackbox.h"
int WINAPI DllEntryPoint(HINSTANCE, unsigned long, void*) {return 1;}

DECLARE_PINS
   ANALOG_IN(VIN, 1);    // Voltage input
   DIGITAL_OUT(FOUT, 2); // Frequency output
END_PINS

DECLARE_VAR
   double Average_v;  // To keep average read voltage
   double K1;         //
   double K2;         // Out freq(KhZ) = K1 + K2 * Average_V
END_VAR

USE_WINDOW(0);   // If window USE_WINDOW(WINDOW_USER_1) (for example)

const char *On_create()
{
   const char *errorMessage = NULL;
   if(GET_PARAM(1) < 0.001)
      errorMessage = "Minimum frequency is 1Hz";
   else
      VAR(K1) = GET_PARAM(1) * 1000.0; // Pass to Hz
   if(GET_PARAM(2) <= 0 || GET_PARAM(2) > 1000)
      errorMessage = "Invalid frequency coefficient";
   else
      VAR(K2) = GET_PARAM(2) * 1000.0;
   return errorMessage;
}

void On_destroy() { /* No action */ }

void On_simulation_begin()
{
   SET_LOGIC(FOUT, 0); // Initializes output
}

void On_time_step(double pTime)
{
   VAR(Average_v) = (GET_VOLTAGE(VIN) + VAR(Average_v))/2;
   if(pTime == 0)
      REMIND_ME( 0.5/(VAR(K1) + VAR(K2) * VAR(Average_v)));
}

void On_remind_me(double pTime, int pData)
{
   SET_LOGIC(FOUT, TOGGLE);
   REMIND_ME(0.5/(VAR(K1) + VAR(K2)*VAR(Average_v)));
}
```

Code sample #2: a interactive event counter

This code, "counter.cpp", correspond to the component "counter.dll", located in <VMLAB_install_dir>\userlib.

```
#include <windows.h>
#include <commctrl.h>
#pragma hdrstop
#include "..\bin\blackbox.h"
int WINAPI DllEntryPoint(HINSTANCE, unsigned long, void*) {return 1;}

DECLARE_PINS
   DIGITAL_IN(SIG_IN, 1);  // Signal input
END_PINS

DECLARE_VAR  // Initialized to zero at creation.
   unsigned long Count;
   unsigned long Break_count;
   bool Count_on_rise;
   bool Count_on_fall;
   bool Break_if_equal;
END_VAR

USE_WINDOW(WINDOW_USER_1);  // Dialog identifier from RC file

const char *On_create()
//********************
{
   VAR(Break_count) = 64; // To match initial GADGET5 (EditBox) value
(arbitrary)
   return NULL; // OK
}

void On_destroy() { /* No action */ }

void On_simulation_begin() { /* No action */ }

void On_digital_in_edge(PIN pDigitalIn, EDGE pEdge, double pTime)
//****************************************************************
// Process edges on pin SIG_IN
{
   bool mustCount = false;
   if(pDigitalIn == SIG_IN) {
      if(VAR(Count_on_rise) && pEdge == RISE)
         mustCount = true;
      else if(VAR(Count_on_fall) && pEdge == FALL)
         mustCount = true;
   }
   if(mustCount) {  // Increase and refresh 'Static' type control GADGET2
      char myBuffer[32];
      itoa(++VAR(Count), myBuffer, 10);
      SetWindowText(GET_HANDLE(GADGET2), myBuffer); // See Win32 API
      if(VAR(Break_if_equal)) {
         if(VAR(Count) == VAR(Break_count))
            BREAK("Counter reached break value");
      }
   }
}
void On_gadget_notify(GADGET pGadgetId, int pCode)
//**************************************************
```

```
{
   switch(pGadgetId) {  // Button click notification: BN_CLICKED
      case GADGET0:     // ---- "Count on rise" select ----
         if(pCode == 0)
            VAR(Count_on_rise) = !VAR(Count_on_rise);
         break;

      case GADGET1:     // ---- "Count on fall" select ----
         if(pCode == BN_CLICKED)
            VAR(Count_on_fall) = !VAR(Count_on_fall);
         break;

      case GADGET4:     // ---- "Break if equal" ----------
         if(pCode == BN_CLICKED)
            VAR(Break_if_equal) = !VAR(Break_if_equal);
         break;

      case GADGET3:    // ----- "Clear" button ------------
         if(pCode == BN_CLICKED) {
            VAR(Count) = 0;
            SetWindowText(GET_HANDLE(GADGET2), "0");
         }
         break;

      case GADGET5:  // Holds the break value. Must detect changes
         if(pCode == EN_CHANGE) {
            char myBuffer[32];
            GetWindowText(GET_HANDLE(GADGET5), myBuffer, 32 - 1);
            VAR(Break_count) = atoi(myBuffer);
         }
         break;
   }
}
```

Part II : AVR™ Microcontrollers and Peripherals Programming Interface

Scope & prerequisites

This pat of the manual is oriented to VMLAB users who wish to code any AVR microcontroller not supported internally by VMLAB.

VMLAB, up to release 3.14, includes internal models of several AVR micros, however, this new feature supported from release 3.15 on, allows users to code new family members, even if no internal VMLAB model is available.

Prerequisites

- A deep knowledge of the involved micro and peripherals to be developed.
- A basic understanding of the VMLAB user-defined components coding mechanism, described in Part I. The procedure for micro-peripherals development is the same, with some new Callback and Interface functions (C.F. and I.F.).
- C language and basic C++.
- A basic understanding of the Win32 controls and resources.

User-defined AVRs

A user-defined AVR model consists of:

- **A Windows .INI file**, for example, "ATMega777.INI". The Project File will use the name "ATmega777" in the .MICRO directive. INI files are a Windows standard type of text files, used to configure applications, etc. In this case such file will determine:

    - The micro attributes: RAM, Flash, EEPROM, etc
    - The supported instructions
    - Description of the interrupt vectors, flags, etc.
    - Description of the built-in peripheral, making reference to internal VMLAB models or an external DLL based user-mode.
    - Description of the special registers implementing additional features.
    - Description of the supported fuses, etc.

- **The necessary DLLs** referred in the above mentioned .INI file, in a DLL-per-peripheral basis. The code for such DLLs follows a similar structure as the user-defined components (see **Part I**) In addition, the utility "usercomp.exe" can also create these DLLs.

All the files (.INI and DLLs) must be placed in a the following directory:
<VMLAB install dir>\mculib>

To load a user-defined micro, there is no special action: just write the model in the Project File .MICRO directive:

```
.MICRO “ATMEGA777”
```

VMLAB will seek first for the ATMEGA777.INI file in the <mculib> directory; if not found, it will try to search for an internal model with this name. Therefore, **models included in the <mculib> directory override internal VMLAB models with the same name**.

The .INI file

Overview

Windows INI files are standard text files used by Windows for parameters settings, applications configuration, etc. A INI file consist of several **sections** named with square brackets, and within each section, there can be defined parameters in the form **Name=Value**, with either numeric or alphanumeric fields, like:

```
[MY_SECTION]
My_parameter_1 = 256         ; A semicolon is interpreted
My_parameter_2 = a string    ; as a comment up to the line end

[OTHER_SECTION]
My_parameter_1 = 254
My_parameter_2 = other string
Other_parameter = 7
```

The AVR micro definition uses these two sections/parameters concepts. **Note that a full description of all the options is not provided here**, since the given examples are self-explanatory, including many comments; therefore, this document outlines just the basic structure of a AVR INI file.

The **[AVRCORE]** section is the base for the rest of the .INI file. Parameters as the flash size, supported instructions, stack size, etc. must be defined in this section.

```
[AVRCORE]
Flash_size_k = 8          ; Flash size, etc
RAM_size = 1024           ; See example for complete details
EEPROM_size = 512         ;
Vector_size = 1
...
Support_IJMP = yes        ; Some instructions are not supported
...                       ; in all micros
```

Within the [AVRCORE] section there are some special parameters with the naming style ‘Xxxx_list’, referring to a list of interrupts, peripherals, etc. Such parameters are a comma separated text list:

```
Interrupt_list = “INT0, INT1, ... “
Peripheral_list = “TIMER0, TIMER1, UART, ...”
Port_8_list = “PA, PB, ...”
Other_pin_list = “AREF, ADC6, ADC7”
Debug_pin_list = “TOVF0, TIM1VIEW”
Flag_register_list = “GIFR, TIFR, ...”
Mask_register_list = “GICR, TIMSK, ...”
Fuse_list = "CKSEL, SUT, ...”
```

For each of the above lists, there must be its corresponding section with the additional details, named with the involved list prefix. Example, [Mask_register:TIMSK] .

Ports

The generic port list must be defined in **‘Port_8_list’**. For each of the ports included in such list, a special section has to be defined; for example:

```
[Port_8:PC]
Exist_mask = $7F  ; PC7 does not exist for this port
Register_map = "DR=$35, DDR=$34, PIN=$33"
```

- The **‘Exists_mask’** specifies which bits exist for the port.
- The **‘Register_map’**, specifies the DR, DDR and PIN addresses.

Additional pins

The parameters **‘Other_pin_list’** and **‘Debug_pin_list’** identify the additional stand-alone pins, which are not standard micro ports. Such pins can be real non-port pins (Other_pin_list) or auxiliary/virtual ones that can serve a variety of functions: monitoring, debugging, etc (Debug_pin_list)

The corresponding electrical nodes with the names given in the list will be created; both kind of extra-pins have exactly the same functionality; the only difference is the icon which identify them in the ‘Messages Window | Electrical Nodes’ section.

Interrupt flag and mask registers

The interrupt list, defined in **‘Interrupt_list’** parameter, must follow the order established in the AVR manual, **excluding the RESET**. Interrupts with some special treatment need a special section; in such section, it is possible to define the initial reset status, etc.

```
[Interrupt:UART_UDRE]          ; UART Data Register Empty
Reset_status = 1
```

Peripherals definition

Peripherals defined in **‘Peripheral_list’** must be detailed in a separated section, for example:

```
[Peripheral:TIMER1]
;Internal_model = TIMER16   ; internal VMLAB 16 bits timer model
DLL_model = my_timer1_dll   ; In this case is a DLL model
Version = 1
Register_map = "TCCRA=$4F, TCCRB=$4E, TCNTH=$4D, ... "
```

```
Port_map = "OCA=PB1, OCB=PB2, ICP=PB0, T1=PD5"
Interrupt_map = "oc1=COMPA1, ovflow=OV1"
```

Peripheral definition sections need the following parameters:

- A **'Internal_model'** or **'DLL_model'** specification. This says what model to use in the simulation. For internal VMLAB models/versions see the list in the annex of the provided INI example.
- A **'Register_map'**, indicating the associated registers and their addresses. Registers defined here must correspond with the ones declared in the the DLL code DECLARE_REGISTERS, and **must follow the same declaration sequence.**
- A **'Port_map'** (optional) if the peripheral has some associated ports, indicating the function name and associated port. Ports defined here must have its counterpart in the DLL DECLARE_PINS (MICRO_PORT( ))
- **'Interrupt _map'**, indicating the interrupts (defined in 'Interrupt_list') that are handled by the peripheral, and must be referred as well in the DLL code by the DECLARE_INTERRUPTS macro.
- A 'Version' number (optional) indicating the version for a given model. This parameter can be retrieved in the DLL with the VERSION( ) I.F.

Fuses

Fuses definition involve three parameters: **'Bit_position'**, **'N_of_bits'** and **'Default_value'** (1 means unprogrammed). It is recommended to use binary values (beginning with b) to describe fuses. Example:

```
[Fuse:CKSEL]
Bit_position = 0          ; Start position for CKSEL0
N_of_bits =  4            ; CKSEL3, CKSEL2, CKSEL1, CKSEL0
Default_value = b0001     ; CKSEL0 = 1 (unprogrammed)
```

Note that the 'Default_value' can be overridden later in the Project File, by assigning the desired value, like:

```
.MICRO "Atmega8" "CKSEL=0000"  ; All bits programmed
```

Special purpose register handling. The DUMMY Peripheral

The AVR family uses a common architecture and basic instruction set, with some variants depending on the family member. These features are supported in the [AVRCORE] section.

However, some more complex features, like external interrupts selection, etc. require a special process depending on the AVR model, and therefore to code them in a special DLL, as an additional "pseudo-peripheral", the so-called DUMMY peripheral.

There is a special section **[Peripheral:DUMMY]** which does this job, with similar parameters as an standard peripheral:

```
[Peripheral:DUMMY]
Register_map = "SFIOR=$50, OSCCAL=$51"
Interrupt_map = "External_0=INT0, External_1=IN1"
DLL_model = my_dummy ; to be processed in “my_dummy.dll”
```

If no DLL is specified in ‘DLL_model’, such registers will be considered just as normal r/w registers (plain RAM), with no special add-in process associated.

> This DUMMY Peripheral DLL follows the same structure as the peripheral one, with the only difference that it responds to some special callback function.

Coding micro-peripheral DLLs

Overview

Coding DLLs for AVR peripherals modeling or special purpose register handling follows the same methodology as coding user components: it is based in callback functions (C.F.) and interface functions (I.F.**).** See **Part I** for a complete description.

The "blackbox.h" header file

The source code (.cpp file) for coding peripherals or special register handling, must include the same “blackbox.h” file as in the user-defined component (see Part I); just a **special #define** must be included before. Three variants are possible now in the source code header:

1) Normal user-defined component:

```
#include <blackbox.h>   // A user-defined component
```

2) Micro peripheral:

```
#define IS_PERIPHERAL   // This will indicate that the DLL is not
#include <blackbox.h>   // a user-component, but a micro peripheral
```

3) DUMMY peripheral:

```
#define IS_DUMMY_PERIPHERAL   // This will indicate that the DLL is
#include <blackbox.h>         // the DUMMY peripheral
```

Compatibility: only release >= 3.15. A DLL so compiled, used with some earlier VMLAB release, will give an error message. However, VMLAB 3.15 accepts DLLs compiled for previous releases (only for user-defined components)

New C.F. and I.F.

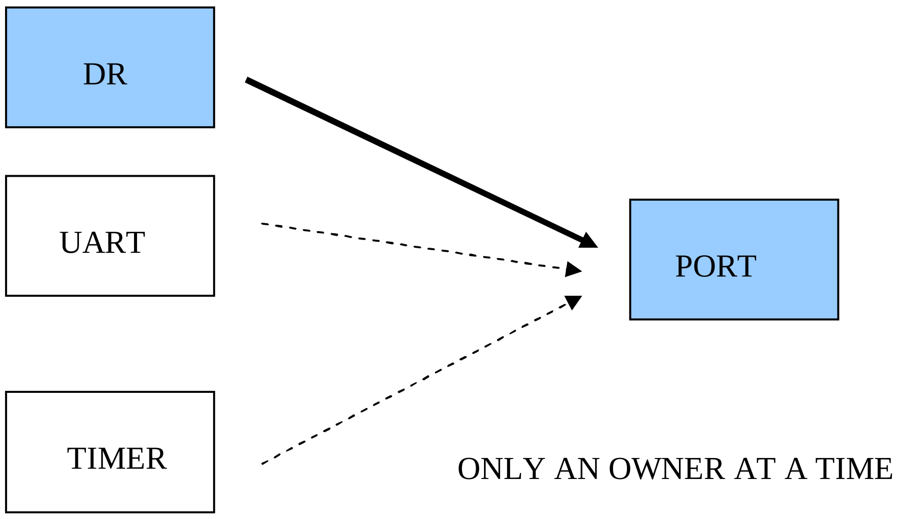
All the Callback and Interface Functions available for user components are also available for peripherals definition, with the following exception:

```
GET_PARAMETER() // Not available for peripherals, use VERSION() instead
```

In addition, the peripheral definition involves some new C.F / I.F. described later in this document.

Handling I/O ports. The AVR VMLAB port model

AVR ports share several functions each (I/Os, UART signals, ADC channels, etc). An accurate simulation behavior for such features is quite challenging, since reproducing at 100% such functionality would require a full port hardware model, leading to a very time consuming simulation.



The **TAKEOVER_PORT( )** function is the one implementing such capability. If a DLL model wishes to assume the port ownership as output, it must invoke this function, which returns a success/fail code for the operation.

The following code could be used in response to a peripheral register bit change:

```
int portStatus = TAKEOVER_PORT(PA0, true); // Want to be the owner
if(portStatus == PORT_OK) {.
   // I am the owner
} else {
  // Analyze return code, give warning message if necessary
}
...

TAKEOVER_PORT(PA0, false); // Release port
```

See a complete description of TAKEOVER_PORT( ) in the I.F. reference section.

Handling I/O ports. Port declarations

The DECLARE_PINS / END_PINS section allows now a new macro to identify micro ports. In the current release, peripherals cannot contain additional analog/digital pins other than ports. Ports must be defined in the same sequence as used in the .INI file "Port_map" section for a given peripheral:

```
Port_map = "OC0A=PA0, OC0B=PB2"  ; In .INI file

DECLARE_PINS
   MICRO_PORT(OC0A, 1);  // Output compare A
   MICRO_PORT(OC0B, 2);  //                B
END_PINS
```

The DLL does not care about actual port assignments (PAx, PBx, etc). This information is only supplied in the .INI file 'Port_map', allowing the use of several instances of the same DLL, associated to different port sets.

Handling I/O ports. Additional pins

Additional pins (analog, digital, etc) can be defined in the DECLARE_PINS in the form explained in Part I. Such non-port pins have to be declared as well in the INI file as previously indicated (Other_pin_list / Debug_pin_list)

For example, pins for an ADC model that uses 2 standard ports + 2 extra pins (4 channels) would be written like this in the DLL:

```
DECLARE_PINS
   MICRO_PORT(A0,   1);   // Port shared channels
   MICRO_PORT(A1,   2);   //

   ANALOG_IN(A2,    3);   // Stand alone channels, non-port
   ANALOG_IN(A3,    4);   //
   ANALOG_IN(v_REF, 5);   // A voltage reference pin
END_PINS
```

Then, in the INI file, there must be a declaration for the extra 'non-port' pins, together with the 'Port_8_list':

```
Port_8_list = "PA, PB, PC"                 ; Generic ports
Other_pin_list = "AREF, ADC2, ADC3,. . ."  ; Standalone pins
```

The assignment from the DLL sequence to the actual ports/pins has to be done in the corresponding 'Port_map' for such ADC peripheral:

```
[Peripheral:MY_ADC]
. . .
Port_map = "A0=PBO, A1=PB1, A2=ADC2, A3=ADC3, v_ref=AREF"
```

Handling I/O ports. Retrieving logic / analog signals

No special functions apply for these tasks. The standard I.F. GET_LOGIC( ) and GET_VOLTAGE( ) also work with ports. See the User Defined Components Manual.

Handling I/O ports. Port change notifications

In order to implement features like external interrupts, timers input capture, external clocks, serial peripherals, etc, it is necessary to process port change notifications. The C.F. managing this feature is **On_port_edge( )**.

There are two different versions of the same function, depending on the type of peripheral code: the DUMMY one (only one instance), or any other (TIMER, etc) normal type. The reason for this is the fact of port mapping. As this is not allowed in the DUMMY peripheral, the original port node name ("PB2", "PD4", etc) must be handled directly by the callback function.

In a normal peripheral (#define IS_PERIPHERAL)

```
void On_port_edge(PORT pPort, EDGE pEdge, double pTime)
//****************************************************
// A external clock based counter. T0 defined as MICRO_PORT
{
   if(pPort == T0 && pEdge == RISE) Counter++;
   ...
}
```

In the DUMMY peripheral (#define IS_DUMMY_PERIPHERAL)

```
void On_port_edge(const char *pName, int pBit, EDGE pEdge, double pTime)
// *************************************************************
// A external interrupt coming from PD3
{
   if(strcmp(pName, "PD") == 0 && pBit == 3 && pEdge = RISE)
      SET_INT_FLAG(INT0, FLAG_SET);
}
```

Handling registers. The WORD8 data type

VMLAB codes all the internal hardware registers or data with the possibility to have three logic states: 0, 1 or X (undefined). This insures a top quality simulation, catching situations where non-initialized variables, RAM, etc, can lead to nasty bugs, difficult to find out otherwise.

In order to implement this feature, it is available a special data type, **WORD8**, coding a 8 bits variable in which each bit can be 0, 1 or X. This data type is implemented as a C++ class inside the "blackbox.h" header file. Example:

```
WORD8 Register1, Register2; // Bits can be 0, 1 or X here
```

WORD8 declared variables can be and-ed, or-ed, x-ored, and complemented with other of same type or int/char, or assigned to standard integer or char types

```
WORD8 myResult = Register1 & Register2; // This is OK
WORD8 myData = 7;                       // This is OK
```

In addition, the WORD8 class provides the following bit or bit field setting/extracting functions:

The **LOGIC get_bit(int bit)** function extracts a single bit as a LOGIC type value. The **[ ] operator** does exactly the same function in a WORD8 type variable.

The **int get_field(int msb, int lsb)** function provides a bit field extraction specified by MSB and LSB parameters, with the result shifted to the bit-0 position. If any bit in the specified field is unknown (X), the return is -1. This function is particularly useful for extracting the different bit groups, which define a micro peripheral behavior.

The **void set_bit(int bit, LOGIC logic)** function sets the given bit to a LOGIC type value.

Examples:

```
WORD8 myData = 0x3A;                   //
LOGIC the3rdBit = myData.get_bit(3); // Extracts 3rd bit, or ..
the3rdBit = myData[3];                 // .. [ ] operator does the same
myData.set_bit(4, 1);                  // Sets 4th bit to value = 1
int bits7_5 = myData.get_field(7,5); // Value of bits 7 to 5
```

Note for C users: Te above syntax is really C++, not C. The source code implementing the WORD8 type is available at the “blackbox.h” header file.

Handling registers. Declatation and mapping

In order to write the code for a DLL model, it is necessary to distinguish three different concepts for any related data register:

- **The register ID code**. This is an integer index variable starting at zero (enum type REGISTER_ID) that VMLAB uses to identify registers in On_xxx( ) parameters. ID codes must follow the same sequence as the declared in the .INI file corresponding section’s ‘Register_map’
- **The register data**. It is WORD8 type variable.
- **The Windows resource code** for the window display (.RC file).

Use the **DECLARE_REGISTERS / END_REGISTERS** macros to declare the registers associated with a peripheral. The sequence must be the same as in the INI file:

```
Register_map = “TCCR0=$41, TCNT0=$42, OCR0A=$43”  ; in .INI file

DECLARE_REGISTERS                // in DLL source .cpp file
   TCCR0, TCNT0, OCR0A          //
END_REGISTERS                    //
```

Note that in this way, the DLL does not care about register addresses. This information is only supplied in the INI file ‘Register_map’, allowing the use of several instances of the same DLL, if needed.

Use the **REG( ) macro** to handle any register declared within DECLARE_REGISTERS / END_REGISTERS block. REG( ) returns a WORD8 data type. Registers are actually handled as a WORD8 type array (see “blackbox.h” source for details)

```
REG(TCCR0) = 0;
WORD8 registerCopy = REG(TCCR0);
```

Handling registers. Display

Registers declared DECLARE_REGISTERS, can also be displayed in the peripheral dedicated window. In order to do this, there must be a .RC resource identifier (GADGETx) associated to each register. Use the **REGISTERS_VIEW / END_VIEW** block to this purpose, and to define individually each bit name, which will show up when the mouse is over the bit. The **DISPLAY( ) and HIDDEN( )** macros will perform the ID_REGISTER mapping to a .RC identifier and bit names:

```
REGISTERS_VIEW
   //       ID      .RC     bit7          .    .    .              bit0

   DISPLAY(TCCR0, GADGET1, FOC0A, FOC0B, *, *, WGM02, CS02, CS01, CS00)
   DISPLAY(TCNT0, GADGET2, *, *, *, *, *, *, *, *)

   HIDDEN(OCR0A) // This register won’t be displayed

END_VIEW
```

Registers declared in DECLARE_REGISTERS must also appear in the REGISTER_VIEW block, either with DISPLAY( ) or HIDDEN( ) macros, otherwise they will be considered as plain RAM, and no notifications (On_register_read/write) will occur for such registers. This situation will cause a warning message.

Handling interrupts

Declare the interrupts used by a peripheral with the **DECLARE_INTERRUPTS / END_INTERRUPTS** macros, with the identifiers that will refer to such interrupts in the code. Identifiers must be valid C/ C++ names.

```
DECLARE_INTERRUPTS
   Overflow, CompA, CompB                // Declare interrupts here
END_INTERRUPTS
...

SET_INTERRUPT_FLAG(Overflow, FLAG_SET);    // Usage example
...
```

The mapping from such names to actual AVR interrupts is handled by the **‘Interrupt_map’** parameter in the INI file, **following the sequence order**. For the above example, assume that the INI file contains:

```
[AVRCORE]
. . .
Interrupt_list = “...COMPA1, ... COMPB1, ... OVF1”
. . .
[Peripheral:TIMER1]
DLL_model = “my_timer_model”  ; code defined in my_timer_model.dll
Interrupt_map = “Overf=OVF1, Cmp_A=COMPA1, Cmp_B=COMPB1”
```

Then, the DLL code identifier ‘Overflow’ will refer to the ‘OVF1’ interrupt; the ‘CompA’ to ‘COMPA1’ and the ‘CompB’ to the ‘COMPB1’ interrupt. Note that parameter names are ignored (‘Overf’, ‘Cmp_A’ and ‘Cmp_B’) and must not necessary match the DLL C ones, since **the assignment is perforemed by the sequence order.** Its presence is just for better readability, and therefore, it is recommended to use the same names as in the C code.

This approach allows multiple peripherals placement, where more instances of the same DLL timer, for example, would use a different interrupts/registers set.

Callback Functions Reference

New Callback Functions

The following list contains the new functions for DLL peripheral coding, in alphabetical order, note that besides, all of the Part I functions are also available, with the exception of GET_PARAMETER( ).

```
void On_clock_change(double pValue);
int  On_instruction(int pCode);
void On_interrupt_start(INTERRUPT_ID pInt);
void On_notify(int pWhat);
void On_port_edge(PORT, EDGE, double);
void On_port_edge(const char *, int, EDGE, double pTime);
WORD8 *On_register_read(REGISTER_ID);
void On_register_write(REGISTER_ID, WORD8);
void On_reset(int pMode);
void On_sleep(int pMode);
```

On_clock_change( )

```
void On_clock_change(double pValue);
```

VMLAB notifies that a clock change has occurred, either due to some SET_CLOCK( ) function, or by a manual clock change in the Control Panel window.

Peripherals whose behavior depends on supplied clock must handle this CF, like UARTs who will have to update the baudrate, or give a convenient warning, etc.

Example:

```
void On_clock_change(double pValue) {  // Some peripherals
   . . .
   Update_baudrate(pValue);            // Must handle this C.F.
}
```

On_instruction( )

```
int On_instruction(int pCode);
```

VMLAB notifies that some instruction that needs special handling is being executed. This C.F. is intended for the use in the DUMMY peripheral to code the behavior of some instructions dependant of some register contents, whose operation mode is not constant among the AVR family members.

The parameter ‘pCode’ contains a special instruction ID code. In the current release, VMLAB notifies three special instructions with the following codes: INSTR_SLEEP, INSTR_SPM and INSTR_WDR

The return must be an integer that will specify the result action, and it will depend on the instruction:

- Return codes for INSTR_SLEEP

SLEEP_DENIED	To deny entering in sleep mode
SLEEP_EXIT,	To indicate the end of sleep mode
SLEEP_IDLE	To enter in such sleep modes
SLEEP_NOISE_REDUCTION	
SLEEP_POWERDOWN	
SLEEP_POWERSAVE	
SLEEP_STANDBY	

- Return codes for INSTR_SPM

SPM_WRITE_BUFFER	To write the temporary buffer
SPM_ERASE_PAGE	To erase the flash page
SPM_WRITE_PAGE	To write the flash pag

- Return codes for INSTR_WDR
  - Ignored, can be 0

Example:

```
int On_instruction(int pCode)
//**************************
// Some DUMMY peripheral code draft. Actions depend on the related
// registers contents
{
   if(pCode == INSTR_SLEEP) {       // Identify instruction.
      if(MCUCR_reg[SPE] == 1) {     // Check bit in MCUCR
         return SLEEP_IDLE;         // Allow sleep
      else
         return SLEEP_DENIED;       // Deny
   } else if(pCode == INSTR_SPM) {
      ...
   }
```

On_interrupt_start( )

```
void On_interrupt_start(INTERRUPT_ID pId);
```

This C.F. notifies that the interrupt service identified in ‘pId’, which must have been defined at DECLARE_INTERRUPTS, has started to be served.

The typical use of this function is to clear an interrupt flag that resides in a peripheral register. Clearing an interrupt flag derived from On_interrupt_start(..) **needs no additional SET_INTERRUPT_FLAG(..)** action, since the micro already has the information to clear it in the table.

Example:

```
void On_interrupt_start(INTERRUPT_ID pId)
{
   REG(TWCR).set_bit[0] == 0)      // Clear interrupt flag
      .. ..                        // in peripheral register
   }
```

On_notify( )

```
void On_notify(int pWhat);
```

Notifcation derived from a NOTIFY( ) I.F called by another DLL. This is used for inter DLL communication

Example, in calling DLL:

```
NOTIFY(“TIMER0”, 7) // Pass a notification to “TIMER0”
```

In the receiving DLL , instance name “TIMER0:

```
Void On_notify(int pWhat) // Catch a NOTIFY() from antother DLL
//*********************
{
   If(pWhat == 7)
      PRINT(“Got it”);
```

See also NOTIFY( ) I.F.

On_port_edge( ) I

```
void On_port_edge(PORT pPort, EDGE pEdge, double pTime)
```

A port notification, specifying that a RISE or FALL edge, specified by the parameter ‘pEdge’, has occurred at a port at the given time = pTime. The ‘pPort’ parameter contains the value of a previously defined port declared at DECLARE_PINS as MICRO_PORT.

Example:

```
void On_port_edge(PORT pPort, EDGE pEdge, double pTime)
//***********************************************
// Could be used to implement an external clock counter
{
   if(pPort == T0 && pEdge == RISE) {
      Counter++;
   }
}
```

For the DUMMY peripheral, this function must be used with a different parameter set, as follows.

On_port_edge( ) I I

```
void On_port_edge(const char *pName, int pBit, EDGE pEdge, double pTime)
```

Second version of On_port_edge( ), available only for the DUMMY peripheral. This C.F is similar to the previous one, but the notification is somewhere different: instead of a port defined as MICRO_PORT, the notification is based in the following parameters:

- pName : the port name "PA", "PB", "PC", etc
- pBit: the bit name within the port. If PA7, this number will be 7
- pEdge: it can be RISE or FALL

This C.F. is especially useful to handle external interrupts in the DUMMY peripheral;

Example:

```
void On_port_edge(const char *pName, int pBit, EDGE pEdge, double pTime)
//*********************************************************************
// Catch a falling edge external interrupt in the PA7 port
{
  if(strcmp(pName, “PA”) == 0) && pBit == 7 && pEdge == FALL) {
      SET_INTERRUPT_FLAG(INT0, FLAG_SET);
   }
}
```

On_register_read( )

```
WORD8 *On_register_read(REGISTER_ID pRegister);
```

This C.F. needs to be defined only if some special process or action is necessary in the peripheral upon a register read operation, for example: clear-on-read, read through temporary register, etc, otherwise it use is unnecessary. The parameter ‘pRegister’ contains the affected register ID. The return value must be the register WORD8 data address involved in the read operation. A NULL return is interpreted as the normal address for the given register: &REG(pRegister), so an standard read operation.

Example:

```
. . .
WORD8 Temp_register;  // Temporary buffer
. . .

WORD8 *On_register_read(REGISTER_ID pRegister)
//******************************************
// Handling temporary register of AVR 16 bits timer
{
   if(pRegister == TCNT1H)    // For all the “xxxH” registers
      return &Temp_register;
   else
      return NULL;            // For the rest of registers no special
}                             // action; normal read.
```

On_register_write( )

```
void On_register_write(REGISTER_ID pRegister, WORD8 pData);
```

On_register_write( ) is perhaps the most important C.F, since it is the one used to activate or deactivate peripherals, establish the configuration, working modes, make diagnostics, etc.

The ‘pRegister’ parameter contains the register ID code, defined in the REGISTER_ID macro, and the WORD8 type ‘pData’, the byte to be written into the register.

Example:

```
void On_register_write(REGISTER_ID pRegister, WORD8 pData)
//*********************************************************
// Activates a timer, if bit 3 is 1. Gives some warning messages
// Shows also the implementation of a read-only bit (at bit 7)
{
   if(pRegister == TCCR1) {

      if(pData[3] == 1) {
         if(Timer_status == COUNTING) {
            BREAK(“Warning: timer already counting”);
         } else {
          Timer_status = COUNTING;
            REMIND_ME(Timer_period);  // Catch count at On_remind_me(..)
         }
      } else if(pData[3] == X) {
         BREAK(“Unknown bit (X) written at STA bit”);
      }
      TCCR1_reg = pData & 0x7F;  // Transfer the data to the real register
                                 // bit 7 is read-only, it will
                                 // become always zero.
   } else if(pRegister == TCOMP) {

      // Some other action at other register, etc
      .. .. ..
   }
}
```

On_reset( )

```
void On_reset(int pResetType);
```

Notification about a micro RESET. The ‘pResetMode’ parameter specifies the type of reset; it can be any of the following constants:

RESET_UNKNOWN
RESET_POWERON
RESET_EXTERNAL
RESET_BROWNOUT
RESET_WATCHDOG

See also RESET( ) I.F.

Example:

```
void On_reset(int pResetType)
//***************************
// Initialize registers. Unused pResetType
{
   TCCR_reg = 0;
   TCOMPA_reg = 0;
   TCOMPB_reg = 0;
}
```

On_sleep( )

```
void On_sleep(int pSleepMode);
```

Notification sent when the micro is entering SLEEP mode. The parameter ‘pSleepMode’ identifies the sleep type and it may be one of the following values, self-explanatory:

SLEEP_IDLE	Enters sleep mode IDLE
SLEEP_NOISE_REDUCTION	
SLEEP_POWERDOWN	
SLEEP_POWERSAVE	
SLEEP_STANDBY	
SLEEP_EXIT	Exits sleep mode

The code must take the necessary actions upon the reception of this notification, depending on the peripheral behavior during SLEEP mode.

Example:

```
void On_sleep(int pSleepMode)
// *************************
{
   switch(pSleepMode) {
      case SLEEP_IDLE:            //  SLEEP modes
      case SLEEP_POWER_DOWN:      //  affecting this peripheral
         Timer_status = STOPPED;
         break;
      case SLEEP_EXIT:
         Timer_status = RUNNING;  // Reactivate counting events
         . . .
}
```

Interface Functions Reference

Some new I.F. provide the necessary communication with the microcontroller: handling interrupts, ports, clock based timing, etc.

List of new I.F. prototypes, in alphabetical order:

```
int GET_FUSE(const char *pName);
WORD *GET_MICRO_DATA(int pWhat, ADDRESS pAddress);
int GET_MICRO_INFO(int pWhat);
int NOTIFY(const char *pInstName, int pCode);
REMIND_ME2(int pCycles, int pParam = 0)
void RESET(int pType)
BOOL SET_CLOCK(double pValue, int pResetCycles);
void SET_INTERRUPT_ENABLE(INTERRUPT_ID pId, BOOL);
void SET_INTERRUPT_FLAG(INTERRUPT_ID pId, int pAction);
BOOL SET_INTERRUPT_VECTORS(int pSelect);
int SET_PORT(PORT pPort, LOGIC value) ;
int SET_PORT_ATTRI(PORT pPort, UINT pValue) ;
int TAKEOVER_PORT(PORT pPort, BOOL pAction) ;
int  VERSION();
void WARNING(const char *pText, int pCategory, int pFlags);
```

GET_FUSE( )

```
int GET_FUSE(const char *pName);
```

Retrieves the fuse value, which must have been defined at the .INI file ‘Fuse_list’ and in the corresponding section

Example:

In the .INI file

```
[SUT]
Bit_position = 4         ; SUT0 stars at bit 4
N_of_bits = 2            ; SUT1, SUT0
Default_value = b10      ; numbers beginning with "b" are binary
```

In the DLL source code .cpp file

```
if(GET_FUSE(“SUT”) == 2)  // 2 is the default value = b10
```

GET_MICRO_DATA)

```
WORD8 *GET_MICRO_DATA(int pWhat, ADDRESS pAddress);
```

I.F. which retrieves data from the host microcontroller, initially issued to model EEPROM peripherals, although it could be extended to more options in the future (RAM, Flash, etc)

EEPROM peripherals do not really need this function, if the user DLL takes care of all the data management related to an EEPROM. However, if the .INI file contains the parameter **EEPROM_size,** VMLAB will automatically allocates this memory space and it will implement the EEPROM persistence by creating a <projectName>.EEP file, which will be read and written at the proper time. So, the use of GET_EEPROM( ) is strongly recommended.

The parameters controlling this function are:

'pWhat': specifies the object whose data has to be retrieved. It can be one of the following self-explanatory constants defined in <blackbox.h>,

- DATA_EEPROM
- DATA_RAM (pending implementation)
- DATA_FLASH (pending implementation)
- DATA_REGISTER (pending implementation)

'pAddress': the relevant address within the specified object

The return value is a WORD8 pointer, being NULL, if the address is out of range or an invalid/unsupported object is specified

Notes:

- If a EEPROM viewer is necessary, it must be implemented in the DLL code
- It is the DLL code responsibility to implement access time features, etc, in EEPROM.
- **Do not assume a WORD8 array structure for the returning WORD8 pointer.**

Example:

```
void On_register_write(REGISTER_ID pId, WORD8 pData)
{
   . . .
   WORD8 *myW8 = GET_MICRO_DATA(DATA_EEPROM, myAddress);
if(myW8) {
      *myW8 = myData;
      ... refresh viewer
   } else
      WARNING("EEPROM address out of range", ... );
}
```

GET_MICRO_INFO( )

```
int GET_MICRO_INFO(int pWhat);
```

Use this function to retrieve some of the micro-controller parameters, specified by ‘pWhat’, which can be one of the following self explanatory constants:

INFO_RAM_SIZE	
INFO_FLASH_SIZE	
INFO_EEPROM_SIZE	
INFO_PROGRAM_COUNTER	
INFO_CPU_CYCLES	Cclock cycles; cleared at Reset
INFO_ADDR_MODE	Addressing mode of current instruction.

The return code -1 must be interpreted as invalid request.

NOTIFY( )

```
int NOTIFY(const char *pInstName, int pCode);
```

I.F. used to send a notification to other DLL instance. This allows an inter-DLL communication mechanism, which is necessary to implement some advanced simulation features. The DLL receiving the notification must catch it with the On_notify( ) C.F.

A typical use of NOTIFY( ) is the activation/deactivation of a peripheral by some register/bit which are not in this peripheral. Example, PRR register bits.

In DUMMY peripheral:

```
. . .
NOTIFY(“TIMER0”, DISABLE_PRR); // DISABLE_PRR = user defined constant
. . .
```

TIMER0 peripheral will receive this notification through ‘On_notify( )

```
void On_notify(int pCode) {    // To catch the notification

   if(pCode == DISABLE_PRR)
   . . .
}
```

REMIND_ME2( )

```
void REMIND_ME2(int pCycles, int pAux)
```

This is a second version of REMIND_ME( ) available only for micro-peripherals. The passed parameter is **int** instead of **double**, as in the original function, and then, the delay is interpreted as a number of micro clock cycles. The ‘pAux’ parameter is an integer value that can be used at your convenience, being reported in the subsequent On_remind_me( ) C.F.

Example:

```
REMIND_ME(250.0e-6, 5) // Will cause a On_remind_me(.., 5) in 250uSec
REMIND_ME2(250, 9)     // Will cause a On_remind_me(., 9), in 250
                       // clock cycles
```

The cycle-based REMIND_ME2( ) C.F. is the right one to model clock-delayed events like timers/counters, watchdogs, A/D conversion time, etc.

RESET( )

```
int RESET(int pType);
```

Forces a reset in the microcontroller, specified by the parameter ‘pType’. This can be one of the following constants (the same as in On_reset( ) C.F.

RESET_UNKNOWN
RESET_POWERON
RESET_EXTERNAL
RESET_BROWNOUT
RESET_WATCHDOG

Note that whenever a peripheral issues a RESET( ), all the remaining peripherals, including the calling one, will receive the On_reset( ) notification. So, avoid calling RESET within On_reset( ), otherwise an infinite loop can be created.

The main use of this function is to code watchdog peripherals.

Example:

```
void On_remind_me(double pTime, int pAux) {  // Tick counter
   . . .
   if(++Counter == TIMEOUT) RESET(RESET_WATCHDOG);
   . . .
}
```

SET_CLOCK( )

```
BOOL SET_CLOCK(double pValue, int pResetCycles = 0);
```

I.F. used to change the clock value of the hosting microcontroller. It is intended to model clock management registers (clock pre-scaling, calibration, etc). The parameter 'pValue' must be in Hertz. The optional parameter 'pResetCycles' allows specifying the initial delay after reset.

The call with the 2nd optional parameter 'pResetCycles' **is only allowed within the On_simulation_begin( )** C.F, otherwise an error will be flagged. Use this call style to process the fuse options determining the initial clock values, reset delay, etc.

The function will return 'true' upon a successful completion and 'false', if the solicited values are out-of-specs (Parameter 'Max_clock_MHz' in .INI file). Note as well that VMLAB restrict the minimum clock to 32KHz, due to some limitations derived from analog simulation, therefore any attempt to set the clock under this value will return 'false'. The 'pResetCycles' is restricted to a max value of 65536.

The function SET_CLOCK is only allowed in the DUMMY peripheral, and it will produce a On_clock_change( ) notification visible in all the DLL that have defined this C.F.

Example:

```
if(Clk_divided_by_2) {
   BOOL clockOK = SET_CLOCK(GET_CLOCK() / 2);
   if(!clockOK)
      WARNING("Clock out of range",. . .);
}
```

SET_INTERRUPT_ENABLE( )

```
void SET_INTERRUPT_ENABLE(INTERRUPT_ID pInt, BOOL);
```

I.F. used to enable/disable the specified interrupt 'pInt', which must have been included in the DECLARE_INTERRUPTS section.

This function is necessary if the interrupt flag is in some peripheral register, and due to a register-write operation, this flag is modified.

Example (in a TWI):

```
void On_register_write(REGISTER_ID pRegister, WORD8 pData) {
   . . .
   if(pRegister == TWCR) {                         // Bit 0 is TWIE
      if(pData[0] == 0 && REG(TWCR)[0] == 1)      // If changes to 1
         SET_INTERRUPT_ENABLE(TWI_INT,  true);
      else if(pData[0] == 1 && REG(TWCR)[0] == 0) // If changes to 0
         SET_INTERRUPT_ENABLE(TWI_INT,  false);
   } else . . .
```

```
    . . .
}
```

Peripherals whose flags are located in special dedicated registers specified in ‘Flag_register_list’ (.INI file), as for example TIMSKx, do not need to use this function.

SET_INTERRUPT_VECTORS( )

```
BOOL SET_INTERRUPT_VECTORS(int pSelect);
```

This I.F. has been included to model features like the reallocation of interrupt vectors, based on some register contents (e.g. IVSEL bits). The return is ‘true’ if the operation is successful and ‘false’ if fails. The parameter ‘pSelect’ can be one of the following constants:

IV_STANDARD_RESET
IV_BOOT_RESET

This I.F. works in accordance with the parameters controlling the reset vector in the .INI file: ‘Default_vectors_start’, ‘Boot_reset_fuse’ , ‘Boot_size_fuse’ and ‘Boot_size_map’.

Example:

```
. . .
if(IVSEL_bits == 0x3
   SET_INTERRUPT_VECTORS(IV_BOOT_RESET)

. . .
```

SET_INTERRUPT_FLAG( )

```
void SET_INTERRUPT_FLAG(INTERRUPT_ID pInt, int pAction);
```

The parameter ‘pInt’ specifies the interrupt ID for the flag to manage. It must have been defined in the DECLARE_INTERRUPTS section. The ‘pAction’ parameter defines the action to perform, and it can be one of the following values (defined in balckbox.h):

FLAG_SET:	Sets the flag; used for normal interrupts, cleared upon completion
FLAG_CEAR:	Clears the flag (except if it is locked).
FLAG_LOCK:	Sets and locks the flag. Use this mode to code level-based external interrupts, which remain active as long as the logic level persists.
FLAG_UNLOCK:	Clears and unlocks a flag set by FLAG_LOCK

Examples:

```
. . .    // Normal flag set
   if(++Counter_value == Output_compare)
      SET_INTERRUPT_FLAG(OCMPA, FLAG_SET);
. . .

void On_port_edge(const char *pName, int pBit, EDGE pEdge)
//**********************************************************
// Handling an external zero level-based interrupt on PA4
// in the DUMMY peripheral DLL
{
   if(pName[1] == 'A' && pBit == 4) {
      if(pEdge == FALL)
         SET_INTERRUPT_FLAG(INT0, FLAG_LOCK);   // Will remain active
      else if(pEdge == RISE)                    // ...
         SET_INTERRUPT_FLAG(INT0, FLAG_UNLOCK); // till this unlocks it
}
```

SET_PORT( )

```
int SET_PORT(PORT pPort, LOGIC pValue);
```

Use it to set a logic value at a given micro port. As ports are shared resources, this operation could not be always successful, and therefore, there is a return value indicating the exit status. The function returns one of the following values:

PORT_OK	Successful operation
PORT_NOT_OUTPUT	The port has not been defined as output
PORT_NOT_OWNER	The current peripheral is not the port's owner
PORT_INVALID	The referred port is invalid

The codes PORT_NOT_OWNER and PORT_INVALID are likely to be received only during the DLL development/debugging phase, therefore, error messages are provided by default.

Unlike the SET_LOGIC( ) function, SET_PORT( ) does not admit delayed settings. The reason for this is, again, that ports are shared resources, and the success cannot be guaranteed, since the status could change during the delay.

Example:

```
void On_remind_me(double pTime, int pAux)
// **************************************
// Sets 0 on output compare
{
   .. ..
   if(++Counter_value == Out_comp_B) {
      if(SET_PORT(PB4, 0) != PORT_OK) {
         BREAK("Warning: some trouble with the port");
      }
   }
   .. ..
}
```

Note: to retrieve the logic value or analog voltage in a port, use the standard I.F GET_LOGIC( ) and GET_VOLTAGE( ). See Part I.

SET_PORT_ATTRI( )

```
BOOL SET_PORT_ATTRI(PORT pPort, UINT pAdd, UINT pRemove = 0);
```

Adds or removes an attribute specified at ‘pAdd’ / ‘pRemove’ at the given port. Attributes may have different meaning, and they should be combined by ORing them. Available in current version are:

ATTRI_DISABLE_DIGITAL: Disables the digital functionality in a port, being only able to read the analog voltage. This option is specified in some micros to improve ADC performance. Ports with this option will not produce edge notifications.

ATTRI_OPEN_DRAIN: Changes a default ‘push-pull’ type port to ‘open-drain’. This feature is often necessary to code certain types of peripherals, like TWI. Changing an output port to open-drain is only possible after becoming the port owner, otherwise an error message will be flagged.

The return value is ‘false’ if any errors occur; ‘true’ otherwise.

Example:

```
. . .
   SET_PORT_ATTRI(PB0, ATTRI_DISABLE_DIGITAL); // Adds attribute
. . .
   SET_PORT_ATTRI(PB0, ATTRI_OPEN_DRAIN, ATTRI_DISABLE_DIGITAL);
   // Adds ‘open drain’ and removes ‘disable digital’
```

TAKEOVER_PORT( )

```
int TAKEOVER_PORT(PORT pPort, BOOL pTakeIt, UINT pOptions = 0);
```

Request to be the owner of the specified ‘pPort’, if the ‘pTakeIt’ parameter is ‘true’, and release it if it is ‘false’. The parameter ‘pPort’ must have been declared as MICRO_PORT(..).

The return value can be one of the following constants:

PORT_OK	Successful request
PORT_NOT_OUTPUT	Port is not defined as output. Up to the user consideration
PORT_NOT_OWNER	Failed; the port is owned by another peripheral

PORT_INVALID	Failed; invalid port ID

The parameter ‘pOptions’ can be one of the following constants:

FORCE_NONE	Default. No I/O direction is forced in the port.
FORCE_OUTPUT	The port direction will be set to ‘output’, regardless of how is defined by the port register
FORCE_INPUT	The port direction will be set to ‘input’, regardless of how is defined by the port register

The previous values can be additionally qualified by ORing them with the TOP_OWNER constant. A peripheral invoking TOP_OWNER will withdraw other’s ownership, unless it was already the TOP_OWNER. In such case, a PORT_NOT_OWNER error will be returned.

The codes PORT_NOT_OWNER and PORT_INVALID are likely to be received only during the DLL development/debugging phase, therefore, error messages are provided by default. It is up to the DLL code to handle the PORT_NOT_OUTPUT return, happening only when the FORCE_NONE option is applied.

Example:

```
...
if(TAKEOVER_PORT(OCA, true) == PORT_NOT_OUTPUT)
   WARNING("Output compare not output", CAT_TIMER, WARN_TIMERS_OUTPUT);
...
TAKEOVER_PORT(TX, true, FORCE_OUTPUT | TOP_OWNER).
```

VERSION( )

```
int VERSION();
```

Use the VERSION( ) I.F. to retrieve the corresponding ‘x’ parameter, ‘Version = x”, as defined in the INI file for the peripheral DLL. In this way, a peripheral DLL can be used with several working modes, options, etc for different microcontrollers, simplifying the code development.

Example:

In the INI file,

```
...
[TIMER1]                               ; Peripheral called TIMER1
DLL_model = timer1                     ; the DLL = timer1.dll
Register_map = "TCCR0=$53, TCNT0=$52"  ; Registers
Port_map = "T0=PD4"                    ; Ports
Version = 2                            ; This is passed to the DLL
...
```

In the DLL source code ‘timer1.cpp’:

```
. .
if(VERSION() == 1)
   Top_count = 0xFF;
else if(VERSION() == 2)
   Top_count = 0x7F;      // This case will apply, as selected in .INI
...
```

Note that the similar I.F. GET_PARAM( ) is not allowed for micro peripherals code. Use VERSION( ) instead.

WARNING( )

```
void WARNING(const char *pText, int pCategory, int pFlags);
```

Warnings about abnormal situations in a peripheral caused by a wrong programming sequence, etc, can be directly implemented with the PRINT( ) and BREAK( ) functions.

However, the WARNING( ) I.F. does this job but working in accordance to the VMLAB menu '**Options | Error reporting'** feature, by which different kind of actions and diagnostic categories can be enabled/disabled.

The parameters 'pCategory' and 'pFlags' specify such diagnostic classification, and can be one of the following constants, defined in 'blackbox.h"

Parameter 'pCategory':

CAT_MEMORY	Memory-related
CAT_UART	UART/USART related peripherals
CAT_ADC	Analog-to-Digital Converter errors
CAT_WATCHDOG	Watchdog related errors
CAT_STACK	Stack pointer related errors
CAT_EEPROM	EEPROM related errors
CAT_SPI	Serial Peripheral Interface related errors
CAT_TWI	Two Wire Interface related errors
CAT_TIMER	Timers related errors
CAT_CPU	Core/CPU related errors
CAT_PORT	I/O port related errors
CAT_COMP	Analog comparator related errors

Parameter 'pFlags':

The following constants report generic situations that apply to several peripherals:

WARN_READ_OVERRUN	Read faster that allowed
WARN_WRITE_OVERRUN	Write faster than allowed
WARN_READ_BUSY	Read peripheral while busy
WARN_WRITE_BUSY	Write to peripheral while busy

WARN_PARAM_BUSY	Setting parameters/modes while busy
WARN_PARAM_RESERVED	Setting reserved parameters/modes
WARN_PARAM_BITRATE	Wrong bit-rate
WARN_MISC	Miscellaneous, unclassified

Data space (memory) access related flags:

WARN_MEMORY_READ_INVALID	Reading from an invalid address
WARN_MEMORY_WRITE_INVALID	Writing to " " " "
WARN_MEMORY_WRITE_X_IO	Writing unknown bits (X) to I/O register
WARN_MEMORY_INDEX_X	X bits in index register
WARN_MEMORY_INDEX_IO	Access to I/O area with indexed addressing
WARN_MEMORY_INDEX_INVALID	Index pointing to invalid address

Specific constants for ADC peripherals:

WARN_ADC_CLOCK	Improper ACD clock rate
WARN_ADC_REFERENCE	Improper reference voltage
WARN_ADC_SHORT	Short circuit of some ADC channel
WARN_ADC_CHANNEL	Improper, inexistent or reserved channel
WARN_ADC_POWDOWN	The ADC is in power-down
WARN_ADC_UNSTABLE	Unstable conversion, due to several causes

Specific constants for TWI peripherals

WARN_TWI	Two Wire Interface related errors
WARN_TIMER	Timers related errors
WARN_TIMER	Timers related errors

Specific constants for EEPROM peripherals

WARN_EEPROM_ADDRES_OUTSIDE	Address outside range
WARN_EEPROM_DANGER	Dangerous operation for data integrity
WARN_EEPROM_SIMULTANEOUS_RW	Simultaneous R/W

Specific constants for timers:

WARN_TIMERS_OUTPUT	Wrong configuration for cap/com signal
WARN_TIMERS_16BIT_READ	Wrong sequence for 16 bits read operation
WARN_TIMERS_16BIT_WRITE	Wrong sequence for 16 bits write operation

Specific constant for UARTs:

WARN_UART_FRAMING	Frame error
WARN_UART_BAUDRATE	Non-standard baudrate

Actions derived from WARNING (break, beep, log, etc) will be determined in the **Options | Error reporting** menu.

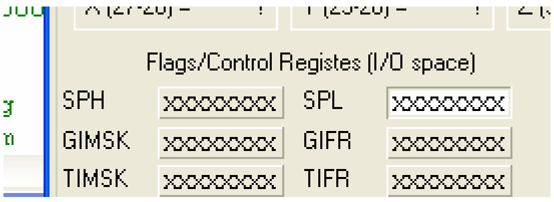
Example:

```
WORD8 myData;      // Give a warning about undetermined bits (X)
. . .
if(wordData.x != 0xFF)
   WARNING(“Detected X bits !”, CAT_MEMORY,  WARN_XBITS)
. . .
```

The Win32 Resources File (.RC)

Although the basic principle and structure is the same as for general user components (Part I), the design of a micro peripheral window requires some additional comments.

VMLAB displays all the register-based information in a special Win32 control that provides binary/hex/decimal view, optional analog bar, bit hints, editing, etc.



This control is modeled in VMLAB by a dedicated Win32 class name called **"WORD_8_VIEW_c"**, and therefore, DLL resources must use this class name in order to display a register.

Displaying a given register needs a pair of Win32 controls: a "static" type to show the register name, and a "WORD_8_VIEW_c" to show/edit the value. The ID for the "static" control must be the same as the "WORD_8_VIEW_c" plus 100, like:

```
CONTROL "", GADGET1, "WORD_8_VIEW_c", ... // The register display
CONTROL "", GADGET1 + 100, "static", ...  // The register name (Id +100)
```

See the provided examples.

General tips and advices

Coding user components and peripherals is not difficult. To ease the startup time in this task and overcome the initial difficulties, take into account the following advices.

- Start always from an already working component, using it as a "template".
- Once compiled, DLLs sometimes need additional DLLs (libraries, etc) to work properly. It is recommended to compile/link component DLLs as "static"/"standalone", otherwise taken into account to deploy also the additional DLLs.
- Use the internal debugging facilities at the beginning: PRINT( ) , TRACE( ).
- Report always your difficulties. Maybe there is another user that has found the solution, and in any case, your input will be considered to improve future VMLAB versions. (visit the AMcTools forum).
- You can share your work. A forum entry exists for users who wish to share their work with the rest of the community. Before coding a component/peripheral, see if someone else has already been working on this. Likewise, you can use this forum to announce your intention to code some peripheral/component, in order to other users do not repeat the work.