OBJECTIS DEVELOPER

Ocf Server

(OcfEmbedded + OcfApiServer 2013 sp1)

-

User Guide

Rev. 1.3

Date : 14th of January, 2014

Ocf Server's Getting Started

www.objectis-software.com
www.objectis-software.com/support
software@objectis.ch

Table of contents

Ocf Server (OcfEmbedded)

1 Introduction

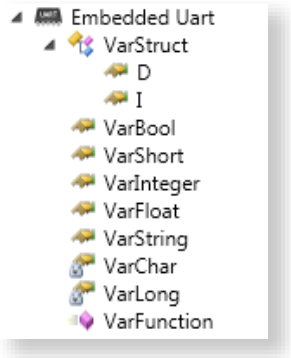
Ocf Server (OcfEmbedded) is necessayry to use any Objectis products in order to conntect your Client (it can be oStudio – Live Tuning or your own application). You will find bellow the basic informations to set your Server.

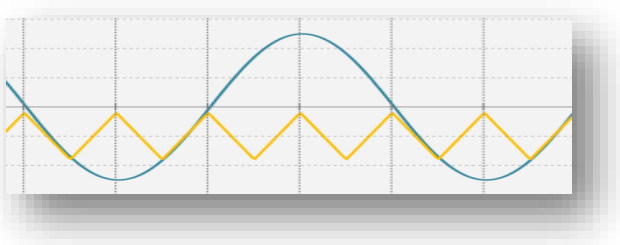
2 Principles

```
OCFE_BEGIN(ocfEntries)
	OCFE_VARIABLE(VarStruct.D, OCFE_TYPE_SHORT)
	OCFE_VARIABLE(VarStruct.I, OCFE_TYPE_CHAR)
	OCFE_VARIABLE(VarBool, OCFE_TYPE_BOOL)
	OCFE_VARIABLE(VarShort, OCFE_TYPE_SHORT)
	OCFE_VARIABLE(VarInteger, OCFE_TYPE_INT)
	OCFE_VARIABLE(VarFloat, OCFE_TYPE_FLOAT)
	OCFE_VARIABLE(VarString, OCFE_TYPE_STRING)
	OCFE_VARIABLE_READONLY(VarChar, OCFE_TYPE_CHAR)
	OCFE_VARIABLE_READONLY(VarLong, OCFE_TYPE_LONG)
	OCFE_FUNCTION_VOID(VarFunction)
OCFE_END
```

The embedded client application publishes variables and functions through helpful macros:

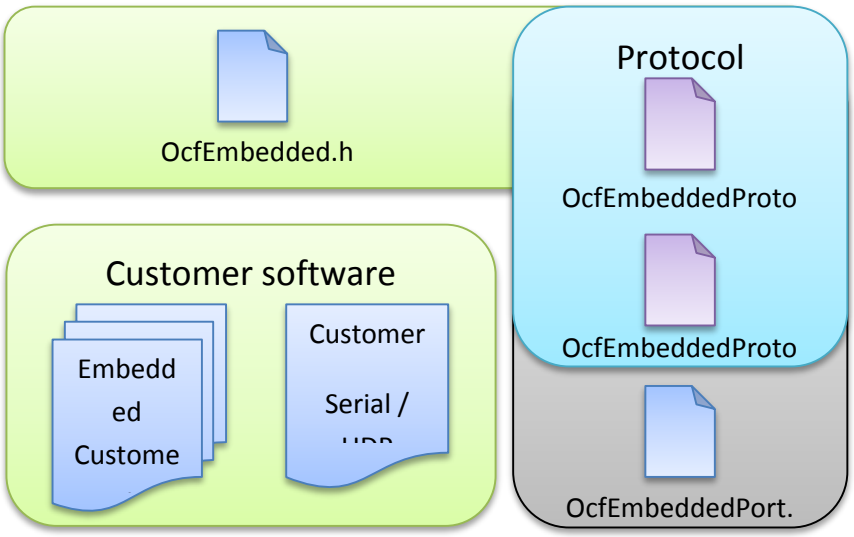
After connecting the target to oStudio Live Tuning, the live debugger allows browsing of all the dictionary content and to use the variables or functions in watches, traces, etc.





3 Package files

3.1 Structure



3.2 Descriptions

The first step is to integrate the four OcfeEmbedded files into a project.

- OcfEmbeddedPort.h (Porting file, editable to suit the platform)
- OcfEmbedded.h (File to include in the main program file)
- OcfEmbeddedProtocol.h (Protocol process header file)
- OcfEmbeddedProtocol.c (Protocol process body file)

4 Porting procedure

1) Make the protocol compile using the OcfEmbeddedPort file to adjust target specification (optional).

   See chapter 5. : Porting file adaptation *(OcfEmbeddedPort.h)*

2) Program the Serial or UDP server to establish a link between the server and the client.

   See chapter 7. : Server protocol

3) Publish variables and functions and connect oStudio Live Tuning

   See chapter 6. : Protocol integration, which shows how to publish variables and functions

   See *oStudio Live Tuning User's guide* to discover the full potential of this tool.

5 Porting file adaptation *(OcfEmbeddedPort.h)*

The *OcfEmbeddedPort.h* porting file is in most cases ready to use. But if necessary, you can adapt the parameters of OcfEmbedded to suit your needs.

5.1 Choosing memory area

With the compilation constant OCFE_MEMORY_AREA you can choose where the OcfEmbedded protocol process data is stored.

If you leave it blank, the data is placed to the default memory area.

Standard declaration:

- You can let the compiler choose (most common use) :
  ```
  #define OCFE_MEMORY_AREA
  ```
- You can also precise it :
  ```
  #define OCFE_MEMORY_AREA __xdata        (example for SDCC)
  ```
  or
  ```
  #define OCFE_MEMORY_AREA pdata          (example for Keil)
  ```

5.2 Features enable/disable

Several features are activated by default. But they can be disabled depending on the needs or capabilities of the processor.

To disable a feature, just comment it.

- `#define OCFE_ENABLE_FLOAT`
  Enable float capability for the protocol. It must be activated to publish float variables.
  Saves a little code memory space when disabled.
- `#define OCFE_ENABLE_DOUBLE`
  Enable double capability for the protocol. It must be activated to publish double variables.
  Saves a little code memory space when disabled.
- `#define OCFE_ENABLE_STRING`
  Enable string capability for the protocol. It must be activated to publish string variables (char array).
  Saves a little code memory space and a little data memory space when disabled.
- `#define OCFE_ENABLE_FUNCTION`
  Enable function invocation for the protocol. It must be activated to publish functions.
  Saves a little code memory space when disabled.
- `#define OCFE_ENABLE_FRAME_NUMERATOR`
  Enable frame numerator security for the protocol. This activates the frame number copy for the response to the client when it is queried, and thus enhances the robustness of the protocol.
  Saves a little data memory space when disabled.
- `#define OCFE_ENABLE_CHECKSUM`
  Enable checksum control security for the protocol. This activates the checksum for the reception and the transmission of each communication, which considerably enhances the robustness of the protocol
  Saves a little data memory space and significant processing time when disabled.

5.3 Required functions

Other parameters are required functions for the protocol process. Normally they are all contained in the standard libraries compilers. But if needed, you can change the functions involved and replace them with your own.

6 Protocol integration

6.1 Dictionary publication

First you have to define the dictionary of variables and functions that you want to publish. There are six macros used to do this. Example:

```
OCFE_BEGIN(ocfEntries)
      OCFE_VARIABLE(VarStruct.D, OCFE_TYPE_SHORT)
      OCFE_VARIABLE(VarStruct.I, OCFE_TYPE_CHAR)
      OCFE_VARIABLE(VarBool, OCFE_TYPE_BOOL)
      OCFE_VARIABLE(VarShort, OCFE_TYPE_SHORT)
      OCFE_VARIABLE(VarInteger, OCFE_TYPE_INT)
      OCFE_VARIABLE(VarFloat, OCFE_TYPE_FLOAT)
      OCFE_VARIABLE(VarString, OCFE_TYPE_STRING)
      OCFE_VARIABLE_READONLY(VarChar, OCFE_TYPE_CHAR)
      OCFE_VARIABLE_READONLY(VarLong, OCFE_TYPE_LONG)
      OCFE_FUNCTION_VOID(VarFunction)
OCFE_END
```

- `OCFE_BEGIN(dictionaryName)`
  Declaration of a dictionary with name *dictionaryName*. Dictionary must be declared in an area accessible by the function that will publish it.
  NB. You can use a specific memory area placement with *xdata, pdata, __idata etc.* keywords if necessary. e.g. `OCFE_BEGIN(xdata ocfEntries)`
- `OCFE_END`
  Must be placed at the end of the dictionary declaration
- `OCFE_VARIABLE(variableName, ocfeType)`
  Allows the publication of a variable in read/write mode. Specify the variable name and the *ocfeType* (c.f. 6.1.1 Ocfe variables type).
  NB. The published variable must be accessible where the dictionary is declared.
- `OCFE_VARIABLE_READONLY(variableName, ocfeType)`
  Allows the publication of a variable in read only mode. Specify the variable name and the *ocfeType* (c.f. 6.1.1 Ocfe variables type).
  NB. The published variable must be accessible where the dictionary is declared.

- `OCFE_FUNCTION_VOID(functionName)`
  Allows the publication of a function with no return value to be invoked by a client (oStudio). Specify the function name.
  NB. The published function must be accessible where the dictionary is declared.
- `OCFE_FUNCTION_RET(functionName)`
  Allows the publication of a function with a return value to be invoked by a client (oStudio). Specify the function name.
  NB. The published function must be accessible where the dictionary is declared.

6.1.1 Ocfe variables type

There are several publishable types of variable. You must precise them in the dictionary declaration.

- `OCFE_TYPE_BOOL`
  To publish a Boolean representation of a variable. The source variable can be a BOOL or a char variable.
- `OCFE_TYPE_CHAR`
  To publish a char ASCII character representation of a char variable.
- `OCFE_TYPE_SHORT`
  To publish a short variable.
- `OCFE_TYPE_INT`
  To publish a int variable.
- `OCFE_TYPE_LONG`
  To publish a long variable.
- `OCFE_TYPE_FLOAT`
  To publish a float variable.
- `OCFE_TYPE_DOUBLE`
  To publish a double variable.
- `OCFE_TYPE_STRING`
  To publish ASCII string representation of a char array variable.

6.1.2 Example of declaration

//Variables declaration :

```
BOOL Light = TRUE;
double MainVelocity = 2.12;
long HoursUsed = 23;
char Type[] = "Robot X22";
char RobotVersion[] = "V4.15";
struct
{
	struct
	{
		int X;
		double Y;
		long Z;
	} AxisA;
	struct
	{
		int X;
		double Y;
		long Z;
	} AxisB;
	int Rotate;
} Robot;

void PowerEnable()
{
	printf("Power enabled\n");
}

char* FunctionWithReturn()
{
	printf("Function ret called correctly\n");
	return "Function ret called correctly";
}
```

// Dictionary declaration:

```
OCFE_BEGIN(ocfEntries)
	OCFE_VARIABLE(Light,  OCFE_TYPE_BOOL)
	OCFE_VARIABLE(MainVelocity, OCFE_TYPE_DOUBLE)
	OCFE_VARIABLE(HoursUsed, OCFE_TYPE_INT)
	OCFE_VARIABLE_READONLY(Type, OCFE_TYPE_STRING)
	OCFE_VARIABLE_READONLY(RobotVersion, OCFE_TYPE_STRING)
	OCFE_VARIABLE(Robot.AxisA.X, OCFE_TYPE_INT)
	OCFE_VARIABLE(Robot.AxisA.Y, OCFE_TYPE_DOUBLE)
	OCFE_VARIABLE(Robot.AxisA.Z, OCFE_TYPE_LONG)
	OCFE_VARIABLE_READONLY(Robot.AxisB.X, OCFE_TYPE_INT)
	OCFE_VARIABLE(Robot.AxisB.Y, OCFE_TYPE_DOUBLE)
	OCFE_VARIABLE(Robot.AxisB.Z, OCFE_TYPE_LONG)
	OCFE_VARIABLE(Robot.Rotate, OCFE_TYPE_INT)
	OCFE_FUNCTION_VOID(PowerEnable)
	OCFE_FUNCTION_RET(FunctionWithReturn)
OCFE_END
```

7 Server protocol

7.1 Server C++

7.1.1 Serial server

There are two functions and one variable used for serial communication

- `OcfeStreamContext`
  Is the type that must be used to declare the context variable for serial communication.

- `OcfeUInt16 OcfeCharacterReceived(`
  `OcfeVariableEntry* ocfeDictionnary,`
  `OcfeStreamContext *context,`
  `char c);`
  When you receive a character by a serial connection, you must pass it to the Ocfe protocol with this function.

  `OcfeVariableEntry* ocfeDictionnary`: Is the pointer to the dictionary used.
  `OcfeStreamContext *context`: Is the pointer to the StreamContext variable used.
  `char c` Is the received character.
  Return : A short representation of the amount of characters that are ready to be sent.
  You can use this value to know if the system is ready to send a response.

- `OcfeUInt16 OcfeCharacterToSend(OcfeStreamContext *context, char* c);`
  This function is used to get the next Ocfe character to send.

`OcfeStreamContext *context`	Is the pointer to the current stream context.
`char* c`	Is the received character.
Return :	A short representation of the amount of characters that are ready to be sent.

7.1.2 Polling Example

This is an example in polling mode. Simply check the new received character from a serial connection and send it to the Ocf Embedded process.

When a character is ready to be sent by the Ocf Embedded process, it will be sent.

```
OcfeStreamContext streamContext;
char c;

//Main loop
void ReadSendOcfeCycle()
{
	if(ReadChar(&c)) //Read serial character received
		//Transmit character in ocfe processing
		OcfeCharacterReceived(ocfEntries, &streamContext, c);

	//Get character to send in ocfe processed buffer
	if(OcfeCharacterToSend(&streamContext, &c))
		WriteChar(c); //Write next character on serial com

}

while(1)
{
	ApplicationCycle(); //Main application process
	ReadSendOcfeCycle(); //Ocfe protocol process (non blocking)
}
```

7.1.3 RX only interrupt Example

This is an example in *half* interrupt mode. When a new character is received from a serial connection it is sent to the Ocf Embedded process.

When a character is ready to be sent by the Ocf Embedded process, it will be sent directly from the Rx ISR handle.

```
//EUSART isr process
void interrupt ISR()
{
	static OcfeStreamContext streamContext;
	char c;

	if(RXInterrupt) // Check if receive flag is set
{
//Transmit character to Ocfe process
		if(OcfeCharacterReceived(ocfEntries, &streamContext,
ReceivedChar))
		{	//while Ocfe is ready to send
//Get next character to send
while(OcfeCharacterToSend(&streamContext, &c)
			WriteChar(c); //Send next character
		}
	}
}
```

7.1.4 RX/TX interrupt Example

This is an example in *full interrupt* mode. When a new character is received from a serial connection, it is sent to the Ocf Embedded process.

When a character is ready to be sent by the Ocf Embedded process, it will be sent directly from the Rx ISR handle. But only the first one will result in triggering Tx interrupts. The remaining characters will be sent by the Tx ISR handle.

```
//EUSART isr process
void interrupt ISR()
{
	static OcfeStreamContext streamContext;
	char c;

if(RXInterrupt) // Check if receive flag is set
	{	//Transmit character to Ocfe process
		if(OcfeCharacterReceived(ocfEntries, &streamContext, RCREG))
		{	//if Ocfe is ready to send
//Get first character to send
			OcfeCharacterToSend(&streamContext, &c);
				WriteChar(c); //Send first character
			TxInterruptEnable = 1; // Enable TX interrupt
		}
	}
	if(TXInterrupt) // Check if transmit flag is set
	{
		if(OcfeCharacterToSend(&streamContext, &c)) //if Ocfe is
ready to send
			WriteChar(c); //Send next Ocfe character
		else
			TxInterruptEnable = 0; // Disable TX interrupt
	}
}
```

7.1.5 UDP server

There is only one variable and only one function used for UDP communication.

- `OcfePacketContext`
  Is the type which must be used to declare the context variable for UDP communication.

- `OcfeUInt16 OcfeExecute (`
  `OcfeVariableEntry* ocfeDictionnary,`
  `OcfeContext *context,`
  `OcfeUInt16 inputDataLength);`
  When you receive a packet from a UDP connection, you must pass it to the Ocfe protocol with this function.

  ocfeDictionnary: Is the pointer to the used dictionary.
  context: Is the pointer to the used PacketContext variable.
  inputDataLength: Is the received characters count.

Return : A short representation of the number of character that are ready to be sent.

7.1.6 Example

This is an example in polling mode. Simply check the new received packets from the UDP connection and send it to the Ocf Embedded process.

You have to directly use the buffer of *packetContext (packetContext.Buffer)* and fill it with the new packet received from the UDP connection.

When a response packet is ready to be sent by the Ocf Embedded process, it will be sent.

```
int count;
	OcfePacketContext packetContext;

	OpenUdp();

	while(1)
	{
//Read udp message received
count = ReadUdp(&udpSocket, packetContext.Buffer, sizeof(OcfeBuffer))
//Process message
		count = OcfeExecute(ocfEntries, &packetContext, count);
//Send udp response
SendUdp(&packetSocket, packetContext.Buffer, count);

	}

	CloseSocket(&udpSocket);
```

7.2 Server C#

7.2.1 Create a server in .NET application

➔ TCP connexion also available.

```
Prototype :
public static OcfApiServer CreateOcfNativeUdpServer(
    Int32 port, object targetObject,
    InvocationAccess access = InvocationAccess.Public);
Sample :
OcfApi.CreateOcfNativeUdpServer(10000, root, out server);
```

7.2.2 Create a secure native server for .NET application (communication with AES encryption)

➔ TCP connexion also available

```
Prototype :
public static OcfApiServer CreateOcfNativeUdpEncryptedServer(
    Int32 port, string passphrase, object targetObject, InvocationAccess
    access = InvocationAccess.Public);
```

7.2.3 Execute the Ocf server (call cyclically)

```
Prototype :
public static void ExecuteOcfServer(OcfApiServer server, TimeSpan timeout =
  TimeSpan.FromMilliseconds(1))
Sample :
for (;;)
  OcfApi.ExecuteOcfServer(server);
```