DSM API MANUAL

Ver 2.2

Copyright © 2008 Euvis Inc.

Getting Started -

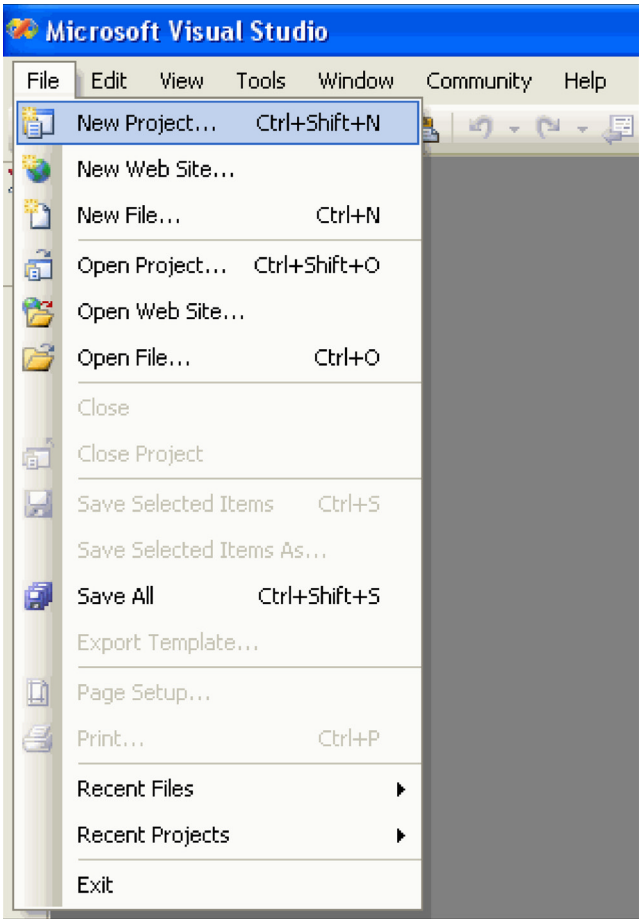
This section will give you a general overview of the program writing process. The API Reference page will show you the necessary steps needed to include the API in your program. The Intermediate Class page gives examplesthat will help you in writing your own intermediate classes. The Basic Structure page will show you the general order of commands that every chirp should adhere to. Please use the menu on the left to navigate to the desired page.

API Reference -

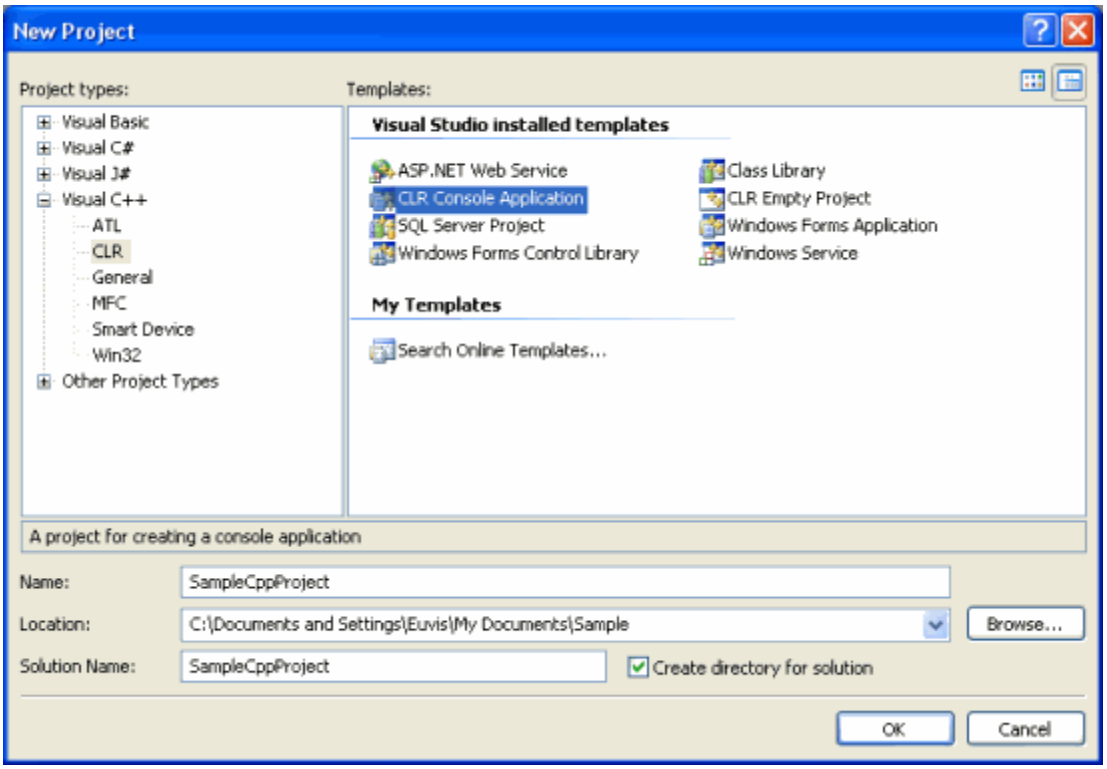
The API file, **DSM_CLR.DLL** is implemented with the **CLR** (Common Language Runtime) support. Languages targeting the runtime, such as C++/CLI, C#, Visual Basic, Jscript, J# and IL (Intermediate Language) assembler, can be used to reference the DSM API. In this manual, we use C++/CLI (with CLR-support) to demonstrate examples. Other languages can be applied in similar ways.

There is no need for header files or declarations of the API in CLR-supported code. The API can be referenced by adding the DSM_CLR.DLL to References (see below). All the available members, properties and methods, can be identified by using the object browser. In order to use the DSM API, the users' applications need to use the CLR-support option in compiling the codes which use the DSM API. It's not required to recompile all the code but only the code using the API need to be compiled with the CLR-support option. To compile the existing code with CLR-support, simply open the ***Property*** of the concerning source codes and select **Common Languag Runtime Support** under **C/C++**:**General**:**Compile with Common Language Runtime Support**. If you are just beginning a new project, simply create the project with CLR support. The following is an example to create a new CLR Console C++ project in Visual Studio.

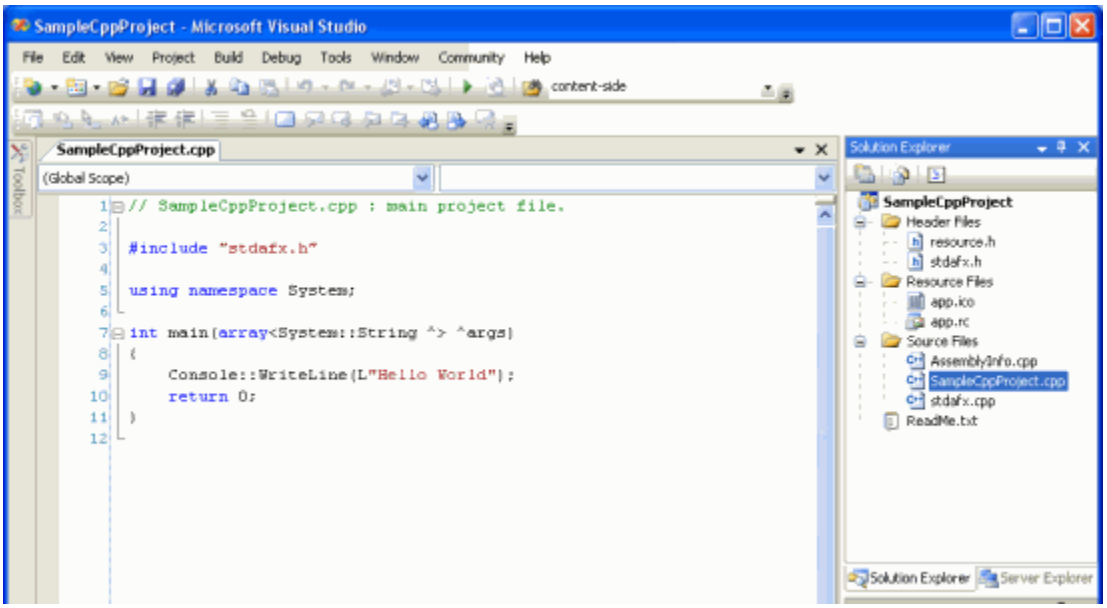
» Go to File --> New Project.



» The New Project window will open. Expand **Visual C++** then click on **CLR**. Select **CLR Console Application** under Templates. Make sure the Location is correct and give the project a name. Once you are done, click on OK.

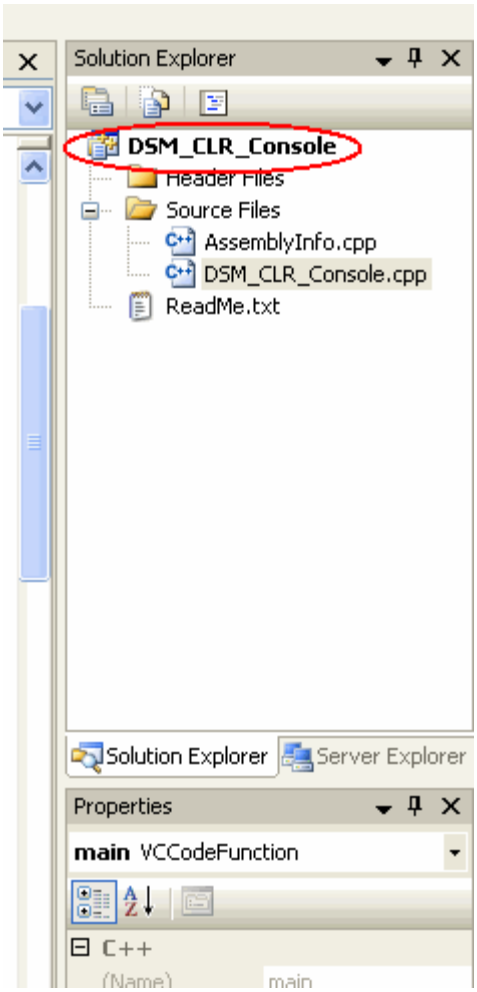


» The code source should open with a sample "Hello World" program.

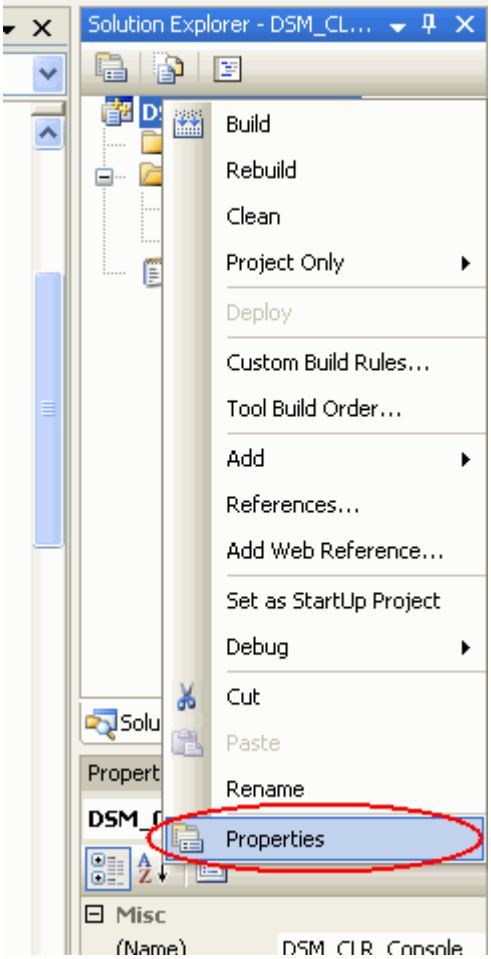


Now we need to add **DSM_CLR.dll** to the references of your solution in Visual Studio.

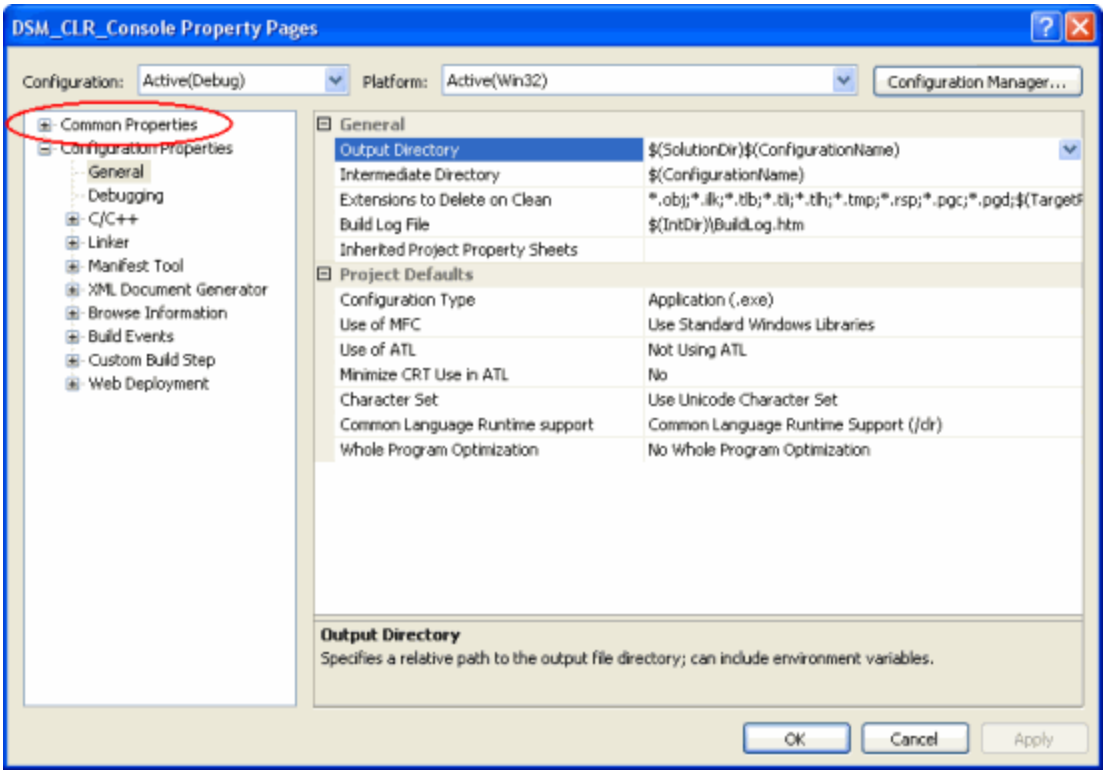
» Right click your solutions file in the Solution Explorer. Your solution will probably be called something different than the one shown.



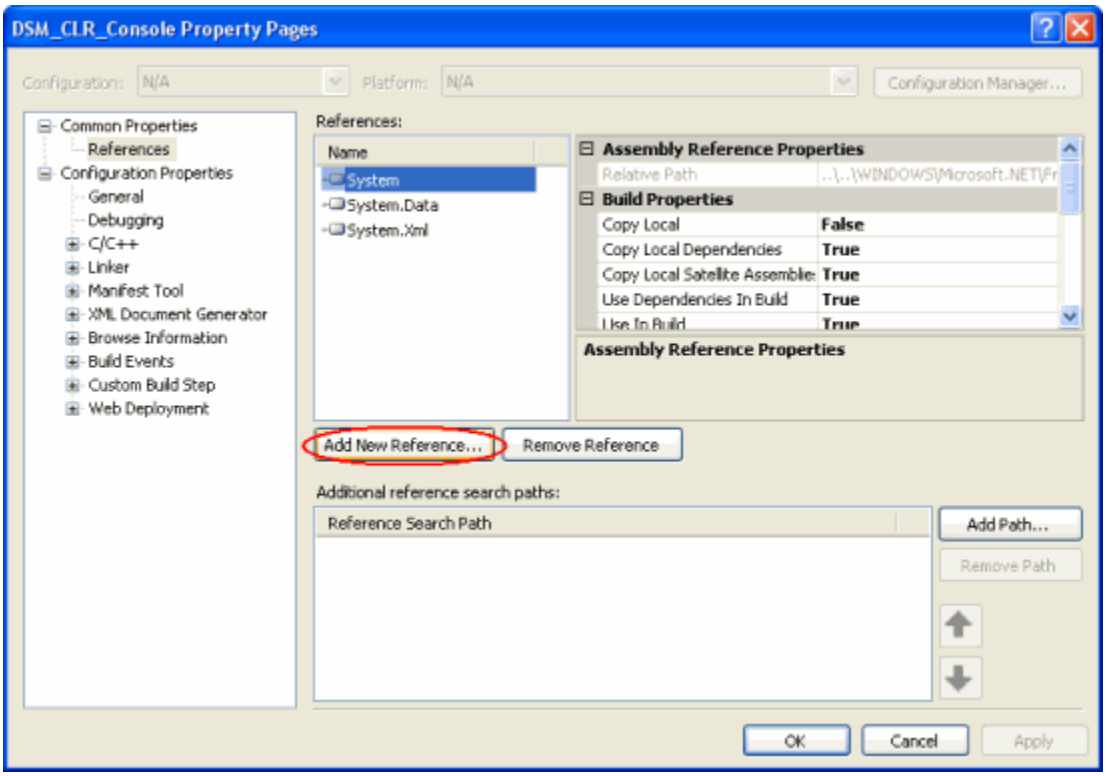
» Click on **Properties** near the bottom of the pop out menu.



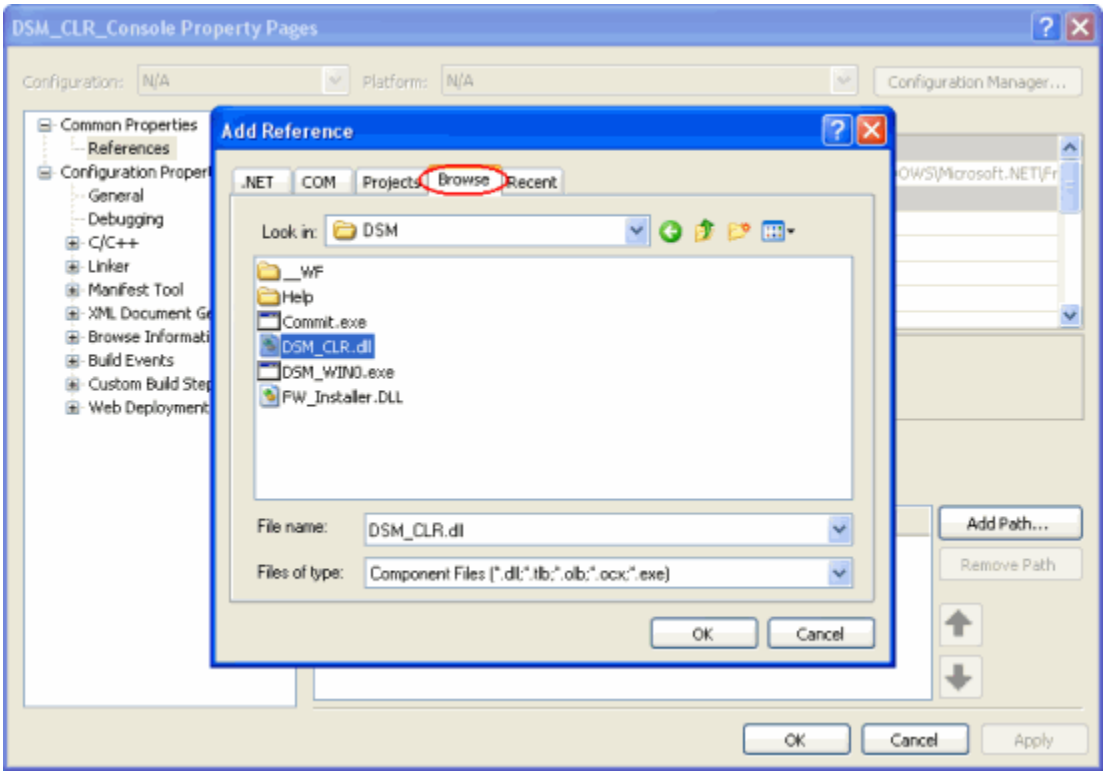
» A new Property window will appear. Click on the "+" next to Common Properties at the top on the left column.



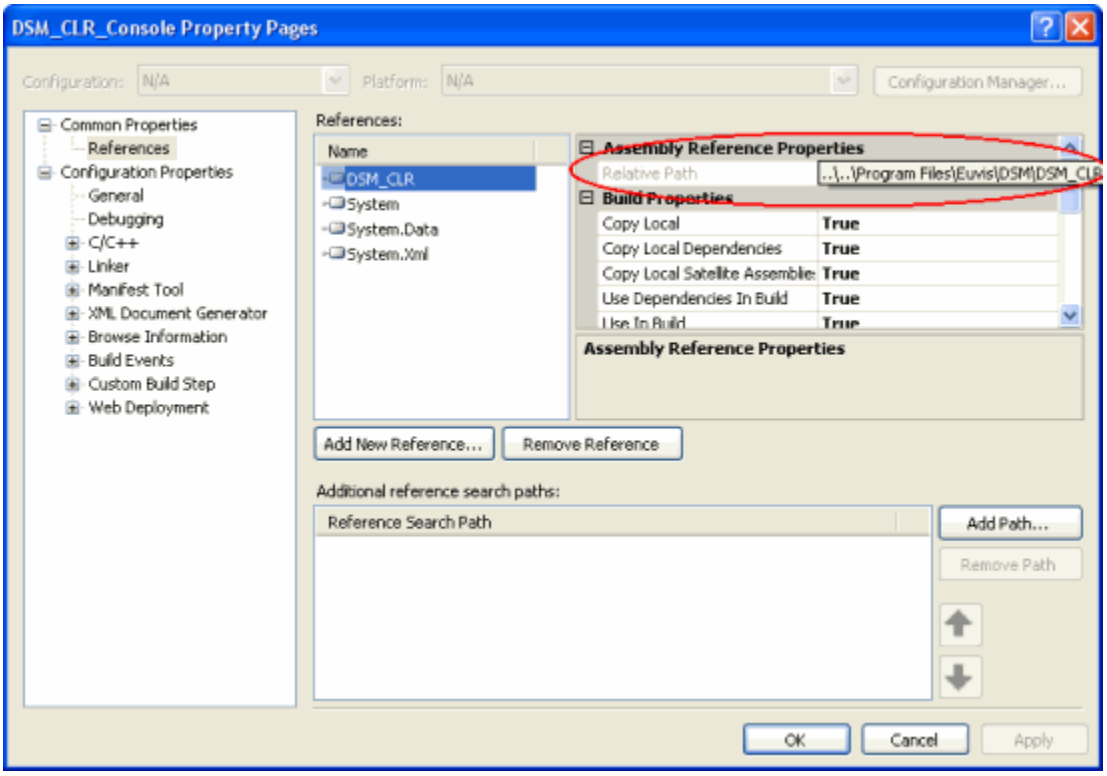
» Click on **References**. Now click on the **Add New Reference** button.



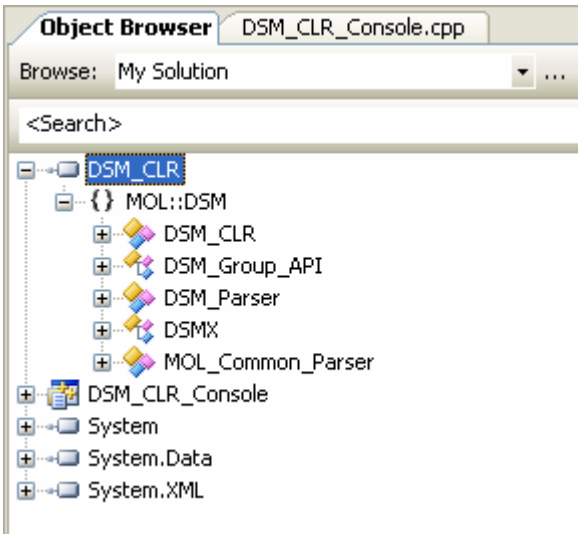
» Click on the **Browse** tab and then browse to the main DSM directory. By default the directory is C:\Program Files\Euvis\DSM. Once you get to the directory, double click on **DSM_CLR.dll** then click OK.



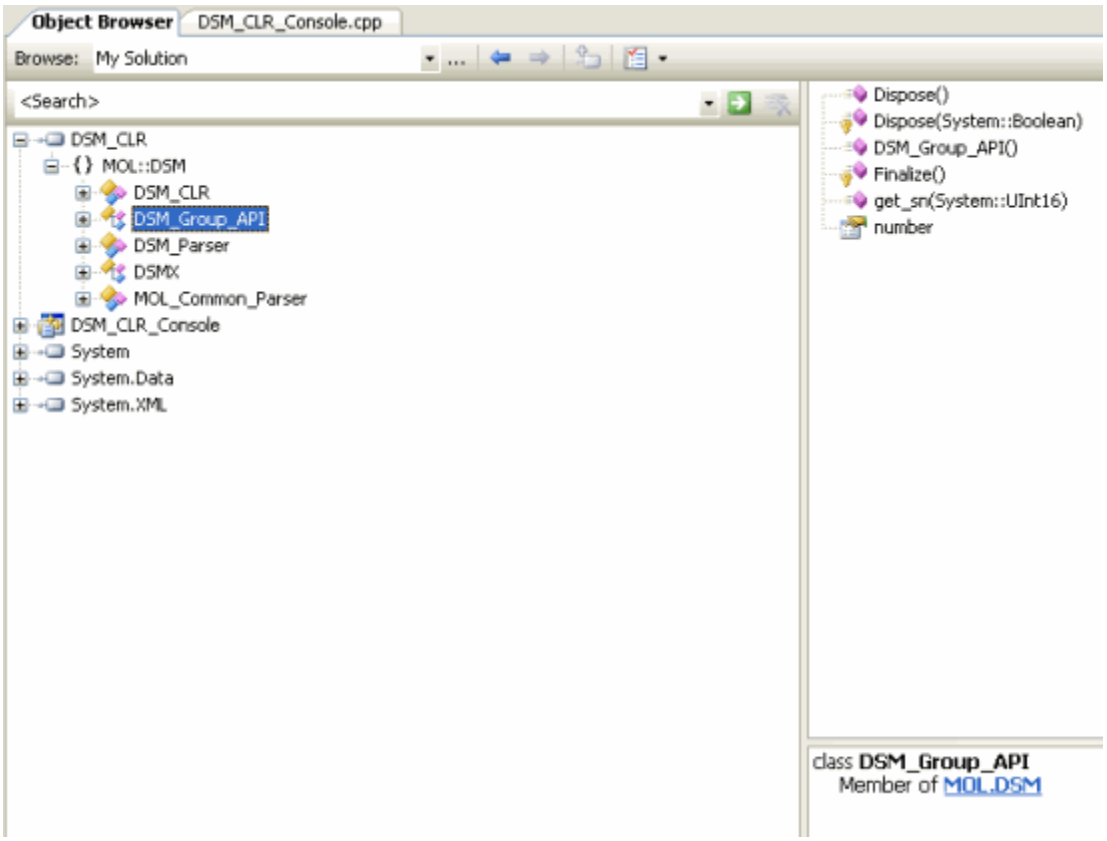
» DSM_CLR should now be listed under References along with its path.



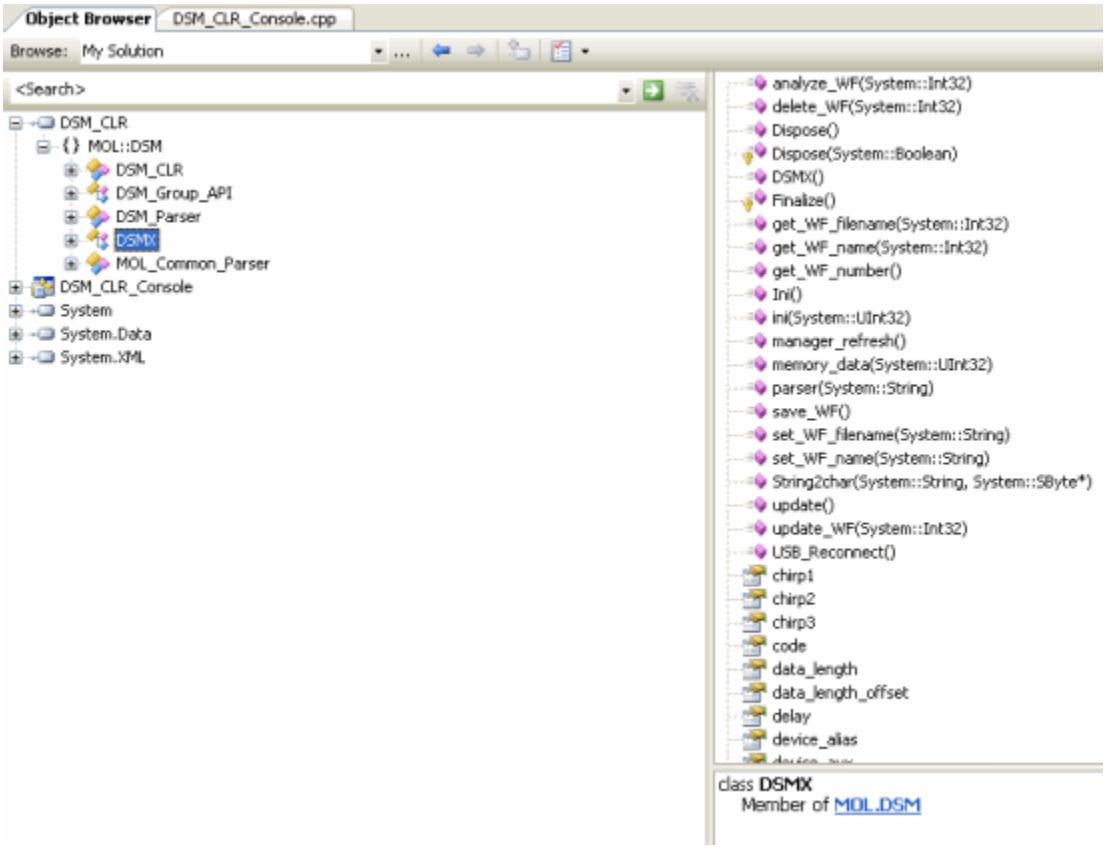
The **DSM_CLR** file provides you with two classes to create and control waveforms. The two classes are *DSMX* and *DSM_Group_API*. To view these two classes open up the Object Browser in Visual Studio. Expand **DSM_CLR** and then expand **MOL::DSM**. You should now see five items under **MOL::DSM** including the two classes of interest: *DSMX* and *DSM_Group_API*.



If you click on *DSM_Group_API* you can see all its members.



Likewise for *DSMX*.



In order to use the two classes, you most use the "using" directive to include the DSM namespace.

```
using namespace MOL::DSM;
```

Intermediate Class -

The API provided to you has direct access to the hardware functions on the DSM board. It gives you access not only to methods but also properties as well. We recommend that you create intermediate classes to interact with the API methods and properties. All of our examples in the Examples section utilizes two intermediate classes: "MyGroup" and "MyDSM". The intermediate "MyGroup" class interacts with the *DSM_Group_API* class while the "MyDSM" class interacts with the *DSMX* class. Notice that in both "MyGroup" and "MyDSM" the underlying API classes, *DSM_Group_API* and *DSMX*, are marked as "private" so that users do not see them at all. These two intermediate classes are declared in the header files below. The dsmgroup.h file contains the "MyGroup" class while the cldsm.h file contains the "MyDSM" class.

dsmgroup.h
cldsm.h

Implementation files:

dsmgroup.cpp
cldsm.cpp

The above two classes of course are just examples. You can make the intermediate classes however you want. For a description of all the methods and properties please go to the Methods section and Properties section respectively.

Basic Structure -

For each chirp that is done by the DSM, the controlling program should adhere to the following structure:

1. **Intialization of board(s)**
2. **Configure control settings**
3. **Waveform parameter definitions**
4. **Download data**
5. **Start DSM memory**
6. **Stop memory when complete (optional)**

Intialization of board(s)

The initialization of the DSM board only needs to be done once in a program assuming that the DSM is not disconnected. Initialization must be done first before all other steps. Initialization is a multiple step process. First you should find out how many DSM boards are connected by reading the `number` property in the *DSM_Group_API* class. Then call the `get sn` member function in the *DSM_Group_API* class to get the board Series Number. The next step is the most important step in the entire DSM program writing process: after you have instantiated a *DSMX* object, you must initialize it by calling the `ini` with the board Series Number as argument. Simply having an *DSMX* object without calling the `ini` method will likely cause your program to fail.

Always call the `ini` function immediately after instantiating a *DSMX* object!

Configure control settings

Control instructions should be redefined everytime a new chirp is implemented. The reason for this is that the DSM will carry over settings from one chirp to the next chirp. Even if two chirps have the same settings you should redefine the settings just in case.

First set the mode (either Free Run, Master or Slave) and the related multi-board settings:

Mode Settings	**Implementation**
Slave Enable/Disable:	`slave mode`
Loop Count:	`loop count`
Auto-Arm Enable/Disable	`auto armed`
SYNCO Enable/Disable	`synco enable`
Internal SYNCI Enable/Disable	`synci enable`

Then set the general control settings that apply to all modes:

Control Settings	**Implementation**
Oversampling Factor:	`over sampling rate`
Marker Enable/Disable:	`marker enable`

Page:	page
Data Length Enable/Disable:	data lenght enable
Phase Reset by Memory Enable/Disable:	dds reset by memory

Waveform parameter definitions

There are three ways you can set up your waveforms. The easiest way is to just define waveform parameters using the built-in waveform properties:

Built-In Waveform Parameters	**Implementation**
Start frequency:	chirp1
Stop frequency:	chirp2
Step frequency:	chirp3
Waveform code:	code
Delay	delay

When we talk about the "built-in waveform" we are referring to a waveform that was constructed using these five parameters.

The second way to define a waveform is to use the user define bulk function.

The final way to define a waveform is to use the user define file function.

There are two mandatory parameters that you must set for ALL waveforms.

Mandatory Parameters	**Implementation**
Data Length	data length
Memory Depth	memory depth

Finally, there are five optional parameters that may be useful for your waveform.

Optional Parameters	**Implementation**
Marker 1	marker1
Marker 2	marker2
Phase Reset Time 1	RESET_T1
Phase Reset Time 2	RESET_T2
Phase Reset Time 3	RESET_T3

Download data

The next step is to download all of the information stored in the computer memory to the memory on the DSM. To do this for the built-in waveform you should use the download method.

For the User-Defined functions, whenever the functions are called the data is automatically downloaded to

the DSM memory.

Start memory

When in Free Run mode, start the memory by using the `restart` method.

For Master mode, use the `arm` method to arm the board. When armed, the Master board will wait for the trigger signal then start the memory.

For Slave mode, use the `slave mode` property to arm the board. When armed, the Master board will wait for the trigger signal then start the memory.

Stop memory (optional)

With the absence of a stop command, the DSM will continue to chirp until the power is shut off. If you want to stop the DSM memory use the `stop` method.

Methods -

Please click on a method in the menu on the left for a description of that method. All methods with the exception of **get_sn** is part of the *DSMX* class. You must instantize an object of the *DSMX* class in order to use any of those methods and you must instantize an object of the *DSM_Group_API* class to use the **get_sn** method. You must use the "." operator to access any of the methods. In the examples, we have used an example **dsm** object of the *DSMX* class and an example **ug** object of the *DSM_Group_API* class.

get_sn

Description

Retrieves the Series Number from the physical device.

Syntax

C++

```
UInt16 get_sn(UInt16 deviceIndexNum)
```

Arguments

`deviceIndexNum`	the index number of the DSM device

Return Value

`seriesNumber`	Series Number of DSM device

Example

```
int seriesNumber;
seriesNumber = ug.get_sn(0);
```

Notes

This method is part of the *DSM_Group_API* class. The Series Number is an unique number that identifies each DSM board. You should always use this method to get the Series Number in order to initialize the DSM board.

When this method returns with "0" it means that there were no DSM boards found. If you try to initialize a *DSMX* object (using the **ini** method) with argument "0", your program will fail. You should have a conditional test to test whether the **get_sn** function returns with "0" and if it does, to stop further execution and display a message.

arm

Description

Arms DSM board in Master mode.

Syntax

C++

```
void arm()
```

Arguments

None

Return Value

None

Example

```
dsm.arm();
```

Notes

This function only arms the board in Master mode. It will not arm the board in Slave mode. To arm the board in Slave mode, use the **slave mode** property.

download

Description

Downloads the built-in waveform parameters to the DSM board.

Syntax

C++

```
void download()
```

Arguments

None

Return Value

None

Example

```
dsm.download();
```

Notes

This function is only needed to download the built-in waveform parameters (**chirp1**, **chirp2**, **code**, etc). If you use the **user define bulk** or the **user define file** functions then you should NOT use this function.

flush

Description

Resets memory to starting memory address.

Syntax

C++

```
void flush()
```

Arguments

None

Return Value

None

Example

```
dsm.flush();
```

Notes

It is a good idea to reset the memory after you stop it. This member method along with the memory stop method, **stop**, should usually be used together.

ini

Description

Initializes the DSM board.

Syntax

C++

```
void ini(UInt32 series_number)
```

Arguments

`series_number`	Series Number of the DSM board

Return Value

None

Example

```
int seriesNumber = ug.get_sn(0);
dsm.ini(seriesNumber);
```

Notes

Please use the **`get sn`** method to get the Series Number.

Always call the `ini` function immediately after instantiating a *DSMX* object!

When you call the **`ini`** function, some of the member properties will be initialized to their default values. To see these default values, please see the Properties section main page.

memory_clock_toggle

Description

Toggles memory clock high or low.

Syntax

C++

```
void memory_clock_toggle()
```

Arguments

None

Return Value

None

Example

```
dsm.memory_clock_toggle();
```

Notes

When memory clock is high, this function will set it low; if memory clock is low, this function will set it high.

memory_data

Description

Retrieves the frequency word from memory

Syntax

C++

```
UInt64 memory_data(UInt32 memAddress)
```

Arguments

`memAddress`	memory address

Return Value

`freqWord`	frequency word

Example

```
unsigned freqWord;
unsigned memAddress = 0x0000A;
freqWord = dsm.memory_data(memAddress);
```

Notes

This method would be a helpful debugging tool should you ever need it.

parser

Description

Feeds commands to the DSM device

Syntax

C++

```
void parser(String command)
```

Arguments

`command`	command being sent to the DSM device

Return Value

None

Example

```
dsm.parser ("d1 1000000");
```

Notes

When entering hexadecimal numbers using this method, you do not need to include the "0x" in front of the number. Due to the way the **`parser`** is implemented, you should always append a newline character following a parser command. You can either do this using the a member function in the intermediate class discussed in the Getting Started section or you can use *#define*. For example, you can use the following code:

```
#define Command(a) dsm.parser(a + "\n")
```

Now instead of typing **`dsm.parser("...")`** all the time, you can just type **`Command("...")`**.

The recommended way of course is to just use a member function in the intermediate class to implement a custom parser command function. An example of this is given in the MyDSM class in the example cldsm.cpp file.

parser Commands

The use of the parser is greatly reduced in the new DSM firmware. Most of the commands are now implemented as either methods or properties in the API classes. You should use those to manipulate the various DSM properties and functions.

Command	*X*	Description
@ate *X*	0	Turns ASIC ATE off. (for fine timing adjustment)
	1	Turns ASIC ATE on.
?status		Inquire the DSM status

restart

Description

Starts memory.

Syntax

C++

```
void restart()
```

Arguments

None

Return Value

None

Example

```
dsm.restart();
```

Notes

When in Master or Slave mode, do not use this method to start the memory. Instead use the **arm** method to arm the Master mode DSM or the **slave mode** property to arm the Slave mode DSM. After arming, the boards will wait for the trigger signal before starting the memory.

stop

Description

Stops the DSM memory.

Syntax

C++

```
void stop()
```

Arguments

None

Return Value

None

Example

```
dsm.stop();
```

Notes

It is a good idea to reset the memory after you stop it. This member method along with the memory reset method, **flush**, should usually be used together.

USB_Reconnect

Description

Reconnects to USB after a disconnect between the computer and the DSM

Syntax

C++

```
void USB_Reconnect()
```

Arguments

None

Return Value

None

Example

```
dsm.USB_Reconnect();
```

Notes

You can use this method to reconnect to the DSM board if you accidently disconnect the USB cable or if DSM was accidently turned off.

user_define_bulk

Description

Chirping with bulk memory.

Syntax

C++

```
void user_define_bulk(UInt32* bulkMemory, UInt32 numOfPoints)
void user_define_bulk(UInt32* bulkMemory, UInt32* controlMemory, UInt32
numOfPoints)
```

Arguments

`bulkMemory`	Bulk memory that will be used for chirping
`controlMemory`	Control bit words for each frequency
`numOfPoints`	Number of frequencies to be chirped

Return Value

None

Example

```
int i;
unsigned *u = new unsigned[16];
unsigned *c = new unsigned[16];
const unsigned FC_START =  0x800000;
const unsigned FC_STEP  = 0x1000000;

for( i=1, u[0]=FC_START; i<16; i++ )
{
        if ( (i-1) < 4 )
        {
                c[i-1] = 3;
                u[i-1] = 0;
                continue;
        }

        u[i]=u[i-1]+FC_STEP;
        }

dsm.user_define_bulk( u, c, 16 );

delete u;
delete c;
```

Control Bit Words

The control bit words are basically two control bits that represent phase reset and marker. Bit 0 (LSB) controls whether the marker is high or low while bit 1 (second LSB) controls the phase reset. For the phase reset bit, "0" turns phase reset off while "1" turns phase reset is on. For the marker bit, "0" turns marker low while "1" turns marker high. Here is a table showing the possible combinations of the control bit:

Dec / Hex	Binary	Phase Reset	Marker
0 / 0x0	00	Off	Low
1 / 0x1	01	Off	High
2 / 0x2	10	On	Low
3 / 0x3	11	On	High

Notes

This method will essentially chirp the *frequency words* that are stored in an array. Note that there are two versions of this method. The method with two arguments will chirp with the frequency words in the bulk memory array. The overloaded method with three arguments does the same but you also have the option of setting the markers as well as reseting the phase on a point-by-point basis. This gives you more control than setting the maker properties (`marker1` and `marker2`) and the reset time properties (`RESET_T1`, `RESET_T2`, and `RESET_T3`) that only allow you to have a range of points.

For example if you used the `user_define_bulk` method with two arguments and you wanted a chirp with 10,000 points then you can only set marker high for one range, say points 2000 through 3000. But if you use the overloaded method with three arguements, then you have the option of setting marker high for points 2000 through 3000 and then points 7446 through 8564. The same goes with the phase reset option.

If you decide to use `user_define_bulk` with three arguments, the default setting if you do not specify a control bit word in the control array is marker low and phase reset off or "0".

If you are using the `user_define_bulk` method with two arguments then you must set the `marker1` and `marker2` properties. If you are using the overloaded `user_define_bulk` method with three arguments, the `marker1` and `marker2` properties will be ignored and you must specify markers using the control array.

If you are using the `user_define_bulk` method with two arguments then you must set the `RESET_T1`, `RESET_T2`, and `RESET_T3` properties. If you are using the overloaded `user_define_bulk` method with three arguments, the `RESET_T1`, `RESET_T2`, and `RESET_T3` properties will be ignored and you must specify phase reset using the control array. It is recommended that when you set phase reset using the control bit words that you also set the corresponding frequency word to "0". For example if reset is turned on for points 100 through 199, the corresponding frequency words for those 100 points should be "0".

When you call the `user_define_bulk` method, the program will automatically download the data to the DSM. There is no need use the `download` method.

user_define_file

Description

Loads custom user waveform files

Syntax

C++

```
UInt32 user_define_file(String filename, Double clock)
```

Arguments

`filename`	file name of custom user file
`clock`	frequency of the input clock

Return Value

`numOfPoints`	number of frequencies in user waveform file

Example

```
unsigned numOfPoints;
Double clock = 2e9;
numOfPoints = dsm.user_define_file("user_defined_file.uwf", clock);
```

Notes

The **`user_define_file`** will chirp the contents of an external .uwf file provided that the file is formatted the correct way. The file gives you complete control of the frequencies to chirp. Chirping is not limited to linear chirps that are only available in built-in waveforms.

If you are using **`user_define_file`** and specify either #Type "1" or "2" then you must set the **`marker1`** and **`marker2`** properties in order to get the correct markers. If you are using **`user_define_file`** and specify either #Type "5" or "6", the **`marker1`** and **`marker2`** properties will be ignored and you must specify markers with the control bit words within the .uwf file.

If you are using **`user_define_file`** and specify either #Type "1" or "2" then you must set the **`RESET_T1`**, **`RESET_T2`**, and **`RESET_T3`** properties in order to get the correct phase reset by memory options. If you are using **`user_define_file`** and specify either #Type "5" or "6", the **`RESET_T1`**, **`RESET_T2`**, and **`RESET_T3`** properties will be ignored and you must specify phase reset with the control bit words within the .uwf file.

When you call the **`user_define_bulk`** method, the program will automatically download the data to the DSM. There is no need use the **`download`** method.

For more information regarding the .uwf file and how to format it please see the DSM Manual.

Properties -

Please click on a property in the menu on the left for a description of that property. All properties are part of the *DSMX* class except `number` which is part of the *DSM_API_Group* class. You must create an object of the *DSMX* class in order to use any of the properties in that class and you must create an object of the *DSM_Group_API* class in order to use any of the properties in that class. In the examples, we have used an example **dsm** object of the *DSMX* class and an example **ug** object of the *DSM_Group_API* class.

Default Property Values

The following is a list of default values that are set when you initialize the DSM with the `ini` method. If you do not change these values in your own program, they will remain at these default values so be sure to redefine these properties if you want to change waveform parameters.

Paging	
`page_number`	4
`page`	0
Waveform	
`chirp1`	0x1000000
`chirp2`	0x10000000
`chirp3`	0x1000000
`RESET_T1`	0
`RESET_T2`	0
`RESET_T3`	0
`marker1`	0x0
`marker2`	0x7
`code`	0
`data_length`	0x10
`data_length_enable`	false
`data_length_offset`	0x192
`dds_reset_by_memory`	false
`delay`	0
`memory_depth`	0x10
Mode	
`loop_count`	0
Hardware Settings	
`over_sampling_rate`	1
`memory_dll`	true
`marker_enable`	true
`auto_armed`	true

synci_enable	true
synco_enable	true
slave_mode	false

number

Description

Specifies how many boards are connected.

Read

Gets how many boards are connected.

Example

```
int numOfBoards;
numOfBoards = ug.number;
```

Write

Read-Only

Notes

This is part of the *DSM_Group_API* class.

auto_armed

Description

Specifies Auto-Arm status.

Read

Gets status of Auto-Arm.

Example

```
bool autoArmStatus;
autoArmStatus = dsm.auto_armed;
```

Write

Sets Auto-Arm status.

Example

```
dsm.auto_armed = false;
```

Notes

Usually you should only use Auto-Arm if you are in Master mode. For a more detailed explanation of Auto-Arm please see the DSM manual.

chirp1

Description

Specifies the start frequency.

Read

Gets current start frequency.

Example

```
unsigned wfChirp1;
wfChirp1 = dsm.chirp1;
```

Write

Sets new start frequency.

Example

```
dsm.chirp1 = 0x1000000;
```

Notes

If you are using reverse chirping, the start frequency should be more than than the stop frequency.

chirp2

Description

Specifies the stop frequency.

Read

Gets current stop frequency.

Example

```
unsigned wfChirp2;
wfChirp2 = dsm.chirp2;
```

Write

Sets new stop frequency.

Example

```
dsm.chirp2 = 0x50000000;
```

Notes

If you are using reverse chirping, the stop frequency should be less than than the stop frequency.

chirp3

Description

Specifies the step frequency.

Read

Gets current step frequency.

Example

```
unsigned wfChirp3;
wfChirp2 = dsm.chirp3;
```

Write

Sets new step frequency.

Example

```
dsm.chirp3 = 0x10000000;
```

code

Description

Specifies waveform code.

Read

Gets the current waveform code.

Example

```
int wfCode;
wfCode = dsm.code;
```

Write

Sets new waveform code.

Example

```
dsm.code = 1;
```

Configurations

Code	Description
0	Frequency ramps up from start frequency to stop frequency then goes back to start frequency. Frequency vs. Time graph resembles sawteeth.
1	Frequency ramps up from start frequency to stop frequency then ramps down. Frequency vs. Time graph resembles triangles.
2	Frequency ramps down from a higher frequency to a lower frequency then goes back to the original higher frequency. In other words, the reverse of wave code "0".

Notes

For code "2" reverse chirping, `chirp1` is still the start frequency and `chirp2` is still the stop frequency. The difference between reverse chirping and normal chirping is that the start frequency will be greater than the stop frequency in reverse chirping.

data_length

Description

Specifies the Data Length which is the amount of memory addresses to make available for chirping. The larger the Data Length, the more frequencies that can be output.

Read

Gets the current Data Length.

Example

```
unsigned wfDL;
wfDL = dsm.data_length;
```

Write

Sets the new Data Length.

Example

```
dsm.data_length = 0x40;
```

Notes

The Data Length should always be either shorter or equal to the Memory Depth. If you want the DSM to go back to the starting frequency before reaching the Memory Depth, you should turn on the Data Length using the **data_length_enable** property. You would use the Data Length if the number of frequencies you want to chirp is not one of the Memory Depth settings. Please note that even if the Data Length is disabled, you should still specify the correct Data Length. For example if the number of frequencies you want to chirp is 65,536 (which is one of the Memory Depth options) then you would set the Memory Depth to 65536 and disable the Data Length but you still would need to set Data Length to the hexadecimal equivalent of 65536 (4000).

The Data Length is useful for users who want a duty cycle chirp. To do this, set the Data Length but set the **data_length_enable** to "false". The DSM will chirp up to the Data Length then will output the start frequency until it reaches the Memory Depth. To see the different patterns of chirps available, please see the DSM Manual.

For users who want a duty cycle chirp but the total number of frequencies is not equal to one of the Memory Depth settings, you can use the **user_define_bulk** or the **user_define_file** functions.

Data Length will not work for chirps that are under 1000 frequencies long even if you have Data Length enabled.

data_length_offset

Description

Specifies the Data Length Offset. At 2.5 Ghz, it is best to set this to 0x192.

Read

Gets current Data Length Offset.

Example

```
unsigned dataLengthOffset;
dataLengthOffset = dsm.data_length_offset;
```

Write

Sets new Data Length Offset.

Example

```
dsm.data_length_offset = 0x192;
```

dds_reset_by_memory

Description

Specifies whether DDS phase reset should be controlled by memory.

Read

Sees if phase reset is controlled by memory.

Example

```
bool phaseResetMem;
phaseResetMem = dsm.dds_reset_by_memory;
```

Write

Enables or disables phase reset by memory.

Example

```
dsm.dds_reset_by_memory = true;
```

Notes

When phase reset is not controlled by memory, you can manually reset the phase by using the "R1" parser command to turn on reset and "R0" parser command to turn off reset. Otherwise, when this property is set to "true", phase reset is done through memory using the Reset Time properties (**RESET T1**, **RESET T2**, **RESET T3**) or using custom phase reset definitions with the **user define bulk** and **user define file** functions.

delay

Description

Specifies the delay which is the number of memory addresses to keep at the start frequency before chirping starts.

Read

Gets current delay.

Example

```
unsigned wfDelay;
wfDelay = dsm.delay;
```

Write

Sets new delay.

Example

```
dsm.delay = 0xFF;
```

Notes

Please note that the delay length is included in the Data Length and Memory Depth. This means that if you had 100 frequencies to chirp and you had Memory Depth (or Data Length) set to 100 point and delay set to 10 points, then the DSM will output the start frequency for the first 10 points and then will only chirp the first 90 frequencies (including the start frequency) that you specified then start over again.

device_alias

Description

Specifies the Alias which is the model number of the device.

Read

Gets the Alias of the device.

Example

```
String^ deviceAlias;
deviceAlias = dsm.device_alias;
```

Write

Read-Only.

device_aux

Description

Specifies the Auxillary code of the device.

Read

Gets the Auxillary code of the device.

Example

```
int deviceAuxCode;
deviceAuxCode = dsm.device_aux;
```

Write

Read-Only.

device_cat

Description

Specifies the Category number of the device.

Read

Gets the Category number of the device.

Example

```
int deviceCatNum;
deviceCatNum = dsm.device_cat;
```

Write

Read-Only.

device_name

Description

Specifies the Name of the device.

Read

Gets the Name of the device.

Example

```
String^ deviceName;
deviceName = dsm.device_name;
```

Write

Read-Only.

device_sn

Description

Specifies the Series Number of the device.

Read

Gets the Series Number of the device.

Example

```
int seriesNumber;
seriesNumber = dsm.device_sn;
```

Write

Read-Only.

device_subcat

Description

Specifies the Subcategory of the device.

Read

Gets the Subcategory of the device.

Example

```
int deviceSubcategory;
deviceSubcategory = dsm.device_subcat;
```

Write

Read-Only.

Dump Directory

Description

Specifies where to save memory dump files.

Read

Gets the current location of memory dump files.

Example

```
String^ dumpDirectory;
dumpDirectory = dsm.Dump_Directory;
```

Write

Sets new location for memory dump files. Specified as a string so must have double quotes.

Example

```
dsm.File_Directory = ".\MD";
```

File_Directory

Description

Specifies where to save and load waveform files.

Read

Gets the current location of waveform files.

Example

```
String^ fileDirectory;
fileDirectory = dsm.File_Directory;
```

Write

Sets new location for waveform files. Specified as a string so must have double quotes.

Example

```
dsm.File_Directory = ".\__WF";
```

firmware_alias

Description

Specifies the Alias of the firmware.

Read

Gets the Alias of the firmware.

Example

```
String^ fwAlias;
fwAlias = dsm.firmware_alias;
```

Write

Read-Only.

firmware_aux

Description

Specifies the Auxillary code of the firmware.

Read

Gets the Auxillary code of the firmware.

Example

```
int fwAuxCode;
fwAuxCode = dsm.firmware_aux;
```

Write

Read-Only.

firmware_cat

Description

Specifies the Category of the firmware.

Read

Gets the Category of the firmware.

Example

```
int fwCatNum;
fwCatNum = dsm.firmware_cat;
```

Write

Read-Only.

firmware_name

Description

Specifies the Name of the firmware.

Read

Gets the Name of the firmware.

Example

```
String^ fwName;
fwName = dsm.firmware_name;
```

Write

Read-Only.

firmware_subversion

Description

Specifies the Subversion of the firmware.

Read

Gets the Subversion of the firmware.

Example

```
int fwSubversion;
fwSubversion = dsm.firmware_subversion;
```

Write

Read-Only.

firmware_version

Description

Specifies the Version of the firmware.

Read

Gets the Version of the firmware.

Example

```
int fwVersion;
fwVersion = dsm.firmware;
```

Write

Read-Only.

loop_count

Description

Specifies the loop count of the chirp.

Read

Gets the current loop count.

Example

```
unsigned loopCount;
loopCount = dsm.loop_count;
```

Write

Sets the new loop count.

Example

```
dsm.loop_count = 0xA;
```

Notes

When this property is set to "0", the loop count is set to infinite so the DSM will be in Free Run mode. Be sure to set this property at the start of every chirp especially if you want the DSM to be operated in Free Run mode. Failure to set `loop_count` might result in unpredictable behavior by the DSM.

marker1

Description

Specifies when the marker signal goes high from low.

Read

Gets the current marker1 memory address.

Example

```
unsigned fwMarker1;
fwMarker1 = dsm.marker1;
```

Write

Sets a new marker1 memory address.

Example

```
dsm.marker1 = 0x10;
```

Notes

Be sure that the markers are turned on. You can set marker settings with the **`marker enable`** property.

marker2

Description

Specifies when the marker signal goes back low.

Read

Gets the current marker2 memory address.

Example

```
unsigned fwMarker2;
fwMarker2 = dsm.marker2;
```

Write

Sets a new marker2 memory address.

Example

```
dsm.marker2 = 0x5000;
```

Notes

Be sure that the markers are turned on. You can set marker settings with the **`marker enable`** property.

marker_enable

Description

Specifies marker signal status.

Read

Gets markers status.

Example

```
bool markerStatus;
markerStatus = dsm.marker_enable;
```

Write

Sets new markers status.

Example

```
dsm.marker_enable = true;
```

Notes

If you turn on the markers, be sure to set the marker properties (**marker1** and **marker2**).

memory_clock

Description

Toggles memory clock either low or high;

Read

Write-Only

Write

Toggles memory clock for "1" and "0" for low.

Example

```
dsm.memory_clock = 0;
```

Notes

You should only toggle the memory clock when the memory is stopped.

memory_clock_advance

Description

Specifies how many memory addresses to skip.

Read

Write-Only

Write

Advances memory by amount user specifies.

Example

```
dsm.memory_clock_advance = 6;
```

Notes

You should only advance the memory address when the memory is stopped.

memory_depth

Description

Specifies the Memory Depth which is the maximum number of memory addresses to make available in the device.

Read

Gets the current Memory Depth.

Example

```
unsigned memoryDepth;
memoryDepth = dsm.memory_depth;
```

Write

Sets a new Memory Depth.

Example

```
dsm.memory_depth = 0x40000;
```

Notes

Memory Depth is a hardware limited setting. There are only nine available settings for Memory Depth:

Decimal	Hex
16	10
64	40
256	100
1024	400
4096	1000
16384	4000
65536	10000
262144	40000
524288	80000

If the number of frequencies in your chirp is not equal to one of the nine above settings, you must set the Data Length using the **data length** property.

When you set the **memory_depth** property, the **page memory depth** property will also be changed. Changing one will change the other. Also, please note that the total Memory Depth available is dependent on the number of pages. If you have configured the DSM to 1 page, then the maximum Memory Depth available per

page will be 524288. If you have 2 pages enabled, then the maximum Memory Depth available per page is 262144. If you have 4 pages enabled, then the maximum Memory Depth will be 131072. Note that 131072 (Hex 20000) is not available as a Memory Depth setting. To get around this, you will have to set Memory Depth to 262144 and then set Data Length to 131072.

memory_dll

Description

Gets memory DLL status.

Read

Sees if memory DLL is on or off.

Example

```
bool memDLLStatus;
memDLLStatus = dsm.memory_dll;
```

Write

Sets new memory DLL status.

Example

```
dsm.memory_dll = true;
```

Notes

For a more detailed explanation of the memory DLL please see the DSM manual.

memory_dump

Description

Specifies if Memory Dump is enabled. If Memory Dump is enabled, the program will dump memory to a .dat file with the Waveform Name (method *get_WF_name*) into the Memory Dump location (property *Dump_Directory*).

Read

Gets the current status of Memory Dump. Specified in boolean.

Example

```
bool memoryDumpEnabled;
memoryDumpEnabled = dsm.memory_dump;
```

Write

Sets Memory Dump either on or off. Specified in boolean.

Example

```
dsm.memory_dump = true;
```

over_sampling_rate

Description

Specifies the oversampling factor.

Read

Gets the current oversampling factor.

Example

```
int oversamplingFactor;
oversamplingFactor = dsm.over_sampling_rate;
```

Write

Sets new oversampling factor.

Example

```
dsm.over_sample_rate = 2;
```

Notes

When this property is set to "1", the oversampling factor is 1. When set to "2", the oversampling factor is 2. When set to "3", the oversampling factor is 4. For more information regarding the oversampling factor, please see the DSM manual.

page

Description

Specifies the current page.

Read

Gets the current page.

Example

```
int currentPage;
currentPage = dsm.page;
```

Write

Sets to a new page.

Example

```
dsm.page = 1;
```

Notes

When there is only one page, only Page 0 is available. When there are two pages, Page 0 and 1 are available. When four pages are available, Page 0, 1, 2, and 3 are available. There is also a parser command that sets this property.

page_memory_depth

Description

Specifies the memory depth available for each page.

Read

Gets current memory depth available per page.

Example

```
unsigned memDepthPerPage;
memDepthPerPage = dsm.page_memory_depth;
```

Write

Read-Only

Notes

This property is the same as the **memory_depth** property. Changing one will change the other. Please read the **memory_depth** page for restrictions on the Memory Depth.

page_number

Description

Specifies how many memory pages there are.

Read

Gets how many memory pages there.

Example

```
int numOfPages;
numOfPages = dsm.page_number;
```

Write

Sets new number of memory pages.

Example

```
dsm.page_number = 4;
```

Notes

The value for this property should correspond with the hardware paging settings. To see how to configure the hardware paging settings please see the DSM Manual.

RESET_T1

Description

Specifies the T_{RESET1} time.

Read

Gets current T_{RESET1} time.

Example

```
unsigned resetT1;
resetT1 = dsm.RESET_T1;
```

Write

Sets new T_{RESET1} time.

Example

```
RESET_T1 = 0;
```

Notes

For a more detailed discussion of the T_{RESET2} time, please see the DSM Manual.

Please note that the reset length is included in the Data Length and Memory Depth. This means that if you had 100 frequencies to chirp and you had Memory Depth (or Data Length) set to 100 points and reset set to 10 points, then the DSM will output the reset for the first 10 points and then will only chirp the first 90 frequencies that you specified then start over again.

RESET_T2

Description

Specifies the T_{RESET2} time.

Read

Gets current T_{RESET2} time.

Example

```
unsigned resetT2;
resetT2 = dsm.RESET_T2;
```

Write

Sets new T_{RESET2} time.

Example

```
RESET_T2 = 0;
```

Notes

It is recommended that you set this property to at least 5 if you are going to use the phase reset by memory option. For a more detailed discussion of the T_{RESET2} time, please see the DSM Manual.

Please note that the reset length is included in the Data Length and Memory Depth. This means that if you had 100 frequencies to chirp and you had Memory Depth (or Data Length) set to 100 points and reset set to 10 points, then the DSM will output the reset for the first 10 points and then will only chirp the first 90 frequencies that you specified then start over again.

RESET_T3

Description

Specifies the T_{RESET3} time.

Read

Gets current T_{RESET3} time.

Example

```
unsigned resetT3;
resetT3 = dsm.RESET_T3;
```

Write

Sets new T_{RESET3} time.

Example

```
RESET_T3 = 3;
```

Notes

It is recommended that you set this property to at least 3 if you use the phase reset by memory option. For a more detailed discussion of the T_{RESET2} time, please see the DSM Manual.

Please note that the reset length is included in the Data Length and Memory Depth. This means that if you had 100 frequencies to chirp and you had Memory Depth (or Data Length) set to 100 points and reset set to 10 points, then the DSM will output the reset for the first 10 points and then will only chirp the first 90 frequencies that you specified then start over again.

slave_mode

Description

Specifies the Slave mode status.

Read

Gets current Slave mode status.

Example

```
bool slaveModeStatus;
slaveModeStatus = dsm.slave_mode;
```

Write

Sets new Slave mode status.

Example

```
dsm.slave_mode = false;
```

Notes

If you intend to be in Master or Slave mode, please remember to set the loop count with the **loop_count** property.

This property is also used to arm the Slave mode board. To arm the board, simply set the property to "true":

```
dsm.slave_mode = true;
```

status

Description

Specifies updated status of device.

Read

Gets the status of the device. Refer to table below for description of status configurations.

Example

```
unsigned deviceStatus;
deviceStatus = dsm.status;
```

Write

Read-Only

Configurations

Decimal	Hex Code	Binary Code	Status
1	0x1	00000001	Armed
2	0x2	00000010	Triggered
4	0x4	00000100	In Loop
8	0x8	00001000	Auto-Armed
16	0x10	00010000	Slave
32	0x20	00100000	Slave Wait
64	0x40	01000000	Infinite Loop
128	0x80	10000000	Data Length Enabled

Note that the device can be simutaneously be in multiple states. In binary code, each "1" bit represents a state. If there is more than one state, then the binary code will have multiple "1" bits. For example if the device was Armed, Triggered, In Loop and Auto-Armed, then the binary code would be "00001111". Status can be output in any format so it is best to output it in binary code.

synci_enable

Description

Specifies internal SYNCI status.

Read

Gets status of internal SYNCI.

Example

```
bool synciStatus;
synciStatus = dsm.synci_enable;
```

Write

Sets new internal SYNCI status.

Example

```
dsm.synci_enable = false;
```

Notes

For more information on the Internal SYNCI signal please see the DSM manual.

synco_enable

Description

Specifies SYNCO signal status.

Read

Gets current status of SYNCO signal.

Example

```
bool syncoStatus;
syncoStatus = dsm.synco_enable;
```

Write

Sets new SYNCO signal status.

Example

```
dsm.synco_enable = true;
```

Notes

The SYNCO signal should normally only be used to synchronize a Slave board. The SYNCO signal is always off when in Slave mode regardless of the **`synco_enable`** setting.

synco_T1

Description

Specifies the T_{SYNC1} time.

Read

Gets the current T_{SYNC1} time.

Example

```
unsigned tSync1;
tSync1 = dsm.synco_T1;
```

Write

Sets a new T_{SYNC1} time. Minimum value of "0x0" and maximum value of "0xFF" (255).

Example

```
dsm.synco_T1 = 0xAA;
```

Notes

This property is only available if SYNCO is enabled with the **synco_enable** property. For a more detailed discussion of the T_{SYNC1} time, please see the DSM Manual.

synco_T2

Description

Specifies the T_{SYNC2} time.

Read

Gets the current T_{SYNC2} time.

Example

```
unsigned tSync2;
tSync2 = dsm.synco_T2;
```

Write

Sets a new T_{SYNC2} time. Specified. Minimum value of "0x0" and maximum value of "0xFF" (255).

Example

```
dsm.synco_T2 = 0xBB;
```

Notes

This property is only available if SYNCO is enabled with the **synco_enable** property. For a more detailed discussion of the T_{SYNC1} time, please see the DSM Manual.

Helpful Tips -

This section will hopefully help you solve some of the more common problems that we have observed when making a program with the API.

General -

All of the following tips assume that you have created an intermediate class with individual member functions to interact with the API methods and properties. This practice is very much recommended.

- Bundle various parser commands, API method calls and property definitions in your intermediate functions for related DSM activities. For example to stop memory, you would use the **`stop`** method, then the **`flush`** method. For examples of bundling commands, please take a look at the example MyDSM class in the implementation file below. Some of the functions which use bundling include: download( ), startMem( ), stopMem( ), setToMaster( ), and setToSlave( ).
    - cldsm.cpp

- Be sure to have the general order of commands as listed on the Basic Structure page in the Getting Started section.

- It is recommended that you always stop the memory at the beginning of a new chirp definition.

- When downloading data into the DSM, create a delay between the time you start downloading and the time you start the memory. This will ensure that all of the data that you have stored on the computer will be downloaded correctly to the DSM. For longer Memory Depth chirps, this is especially important. The delay will be dependent on your system resources. You can experiment with the delay time by using the DSM GUI by downloading waveforms of various Memory Depth.

- As already stated on the **`data length`** properties page, when you want to perform a duty cycle chirp and the total number of frequencies is equal to one of the nine Memory Depth settings, you should turn off Data Length Enable and set the Data Length to the number of non-constant frequency points. If your total number of points is not equal to one of the nine Memory Depth settings, then you must either use the **`user define bulk`** or the **`user define file`** functions.

Multi-Board -

- Listed below are the recommended sequence of commands to put DSM into Master mode. These commands can be combined in a intermediate class member function.

```
dsm.slave_mode = false;
dsm.auto_armed = true;
dsm.synci_enable = true;
dsm.synco_enable = true;
```

- When you want to arm the Master DSM board to wait for the trigger signal, do not use the **restart** method (as you do in Free Run Mode). Instead use the **arm** method. The **arm** command will only arm the Master board, not the Slave board.

- Listed below are the recommended sequence of commands to put DSM into Slave mode.

```
dsm.slave_mode = true;
dsm.auto_armed = false;
dsm.synci_enable = false;
```

- To arm the Slave board to wait for the SYNCI signal, use the **slave mode** property, not the **arm** or **restart** methods.

- For either Master or Slave mode, be sure to specify the loop count to a non-zero, non-negative number. If you fail to do this then the board will use the last specified loop count setting (default is 0) in which case your board might be in Free Run mode.

- We recommend that you stop the memory before arming either the Master or Slave boards. This has the effect of stabilizing the hardware.

Debugging -

- You can have the GUI open when you are running your own program. The GUI can be a big help when you need to debug your program. Make sure that you use the USB Reconnect command before using it.

- You can use the GUI to check the status of the DSM. You can also stop the DSM memory with it and check the data in memory.

- All parser commands are sent to the DSM301.log file in your program's .exe folder.

- The easiest way to see whether Data Length is enabled is to set the Data Length to half of the Memory Depth and the markers to half of the Data Length. If the marker has a half duty cycle, then Data Length is on (marker runs to half of the Data Length). But if the marker has a quarter duty cycle, then the DSM is not running up to the Data Length (marker is a quarter of Memory Depth).

Examples -

Please choose an example in the menu to see an example. The Solution files for the two examples are zipped and can be found in the "Example" folder in your DSM folder that you specified during installation. The default location is **C:\Program Files\Euvis\DSM\Example**.

Please note that you must re-reference the DSM_CLR.dll API file for the examples. To do this, follow the instructions as detailed on the API Reference page.

For convenience, we have provided the header files and implementation files for the intermediate classes below. The souce code for each example can be found on the individual example pages.

dsmgroup.h
dsmgroup.cpp

cldsm.h
cldsm.cpp

General Waveforms -

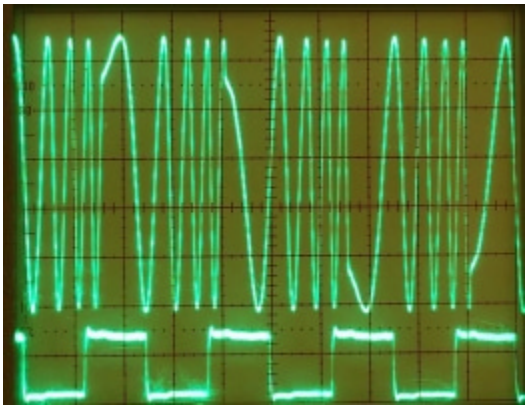
This example will take you through the major functions of the DSM. The source code is located in the "basics" folder in your "Examples" folder. Each example within the program's output is shown below. For the oscilloscope images, the top trace is of the actual waveform while the bottom trace is of the marker. For the spectrum images, the bandwidth is set from 0 to the Nyquist frequency of 1 GHz (clock is set to 2 GHz).

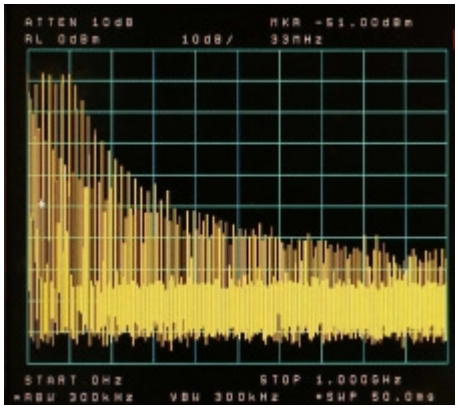
For convenience here is the source code:

basics.cpp

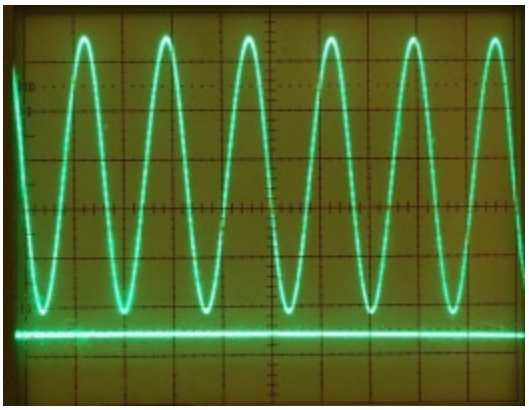
Example 1

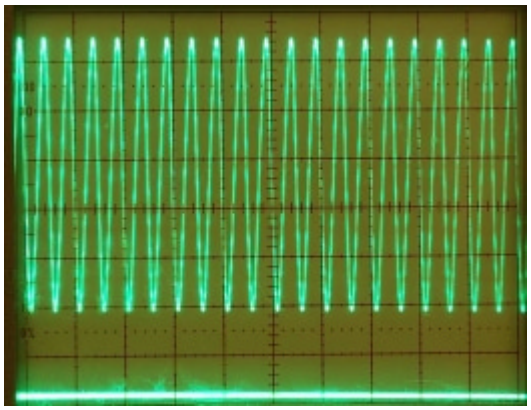
This example is just a simple 16-point waveform.





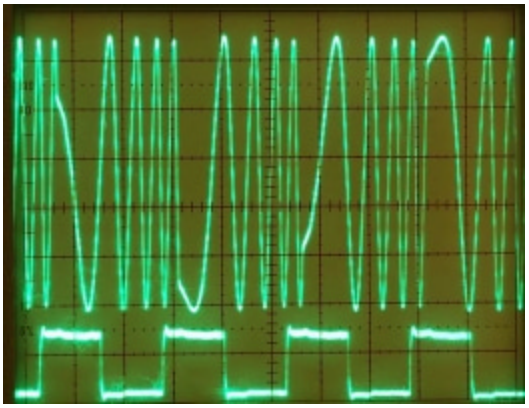
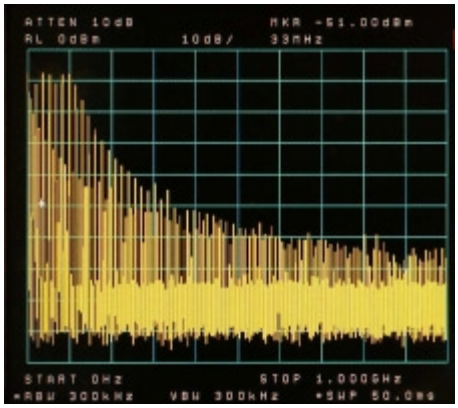
When pointwise chirping:





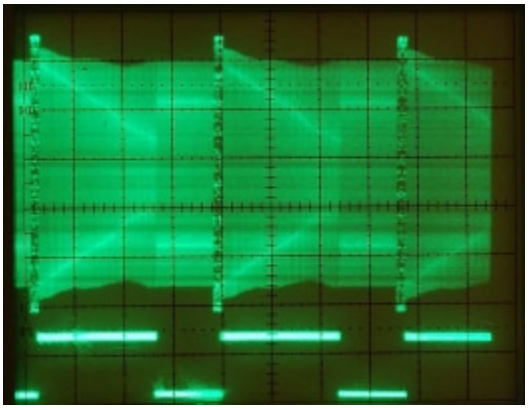
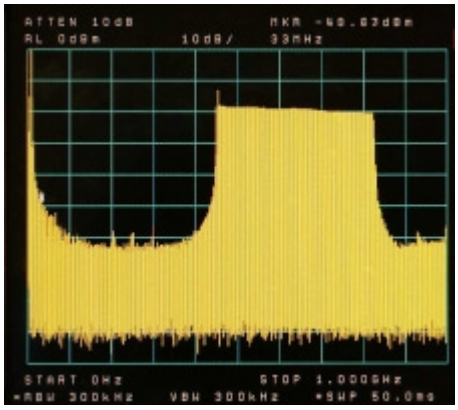
Example 2

This example is the the same as the above but this time uses the `user_define_bulk` method to define the waveform.

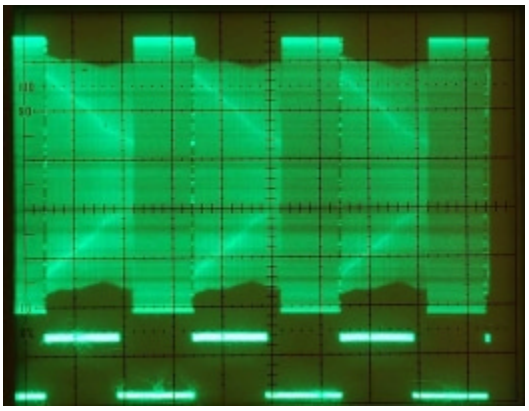
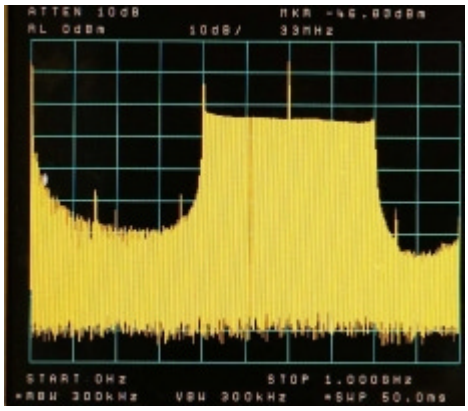
Example 3

This example demonstrates the use of the `user_define_bulk` method to create a duty cycle chirp. The marker is set to high when the DSM is chirping the non-constant frequencies and low when its outputs the constant frequencies.

Example 4

This example once again uses the `user_define_bulk` method to create a period based chirp. Please take a look at the code to see how it is implemented.

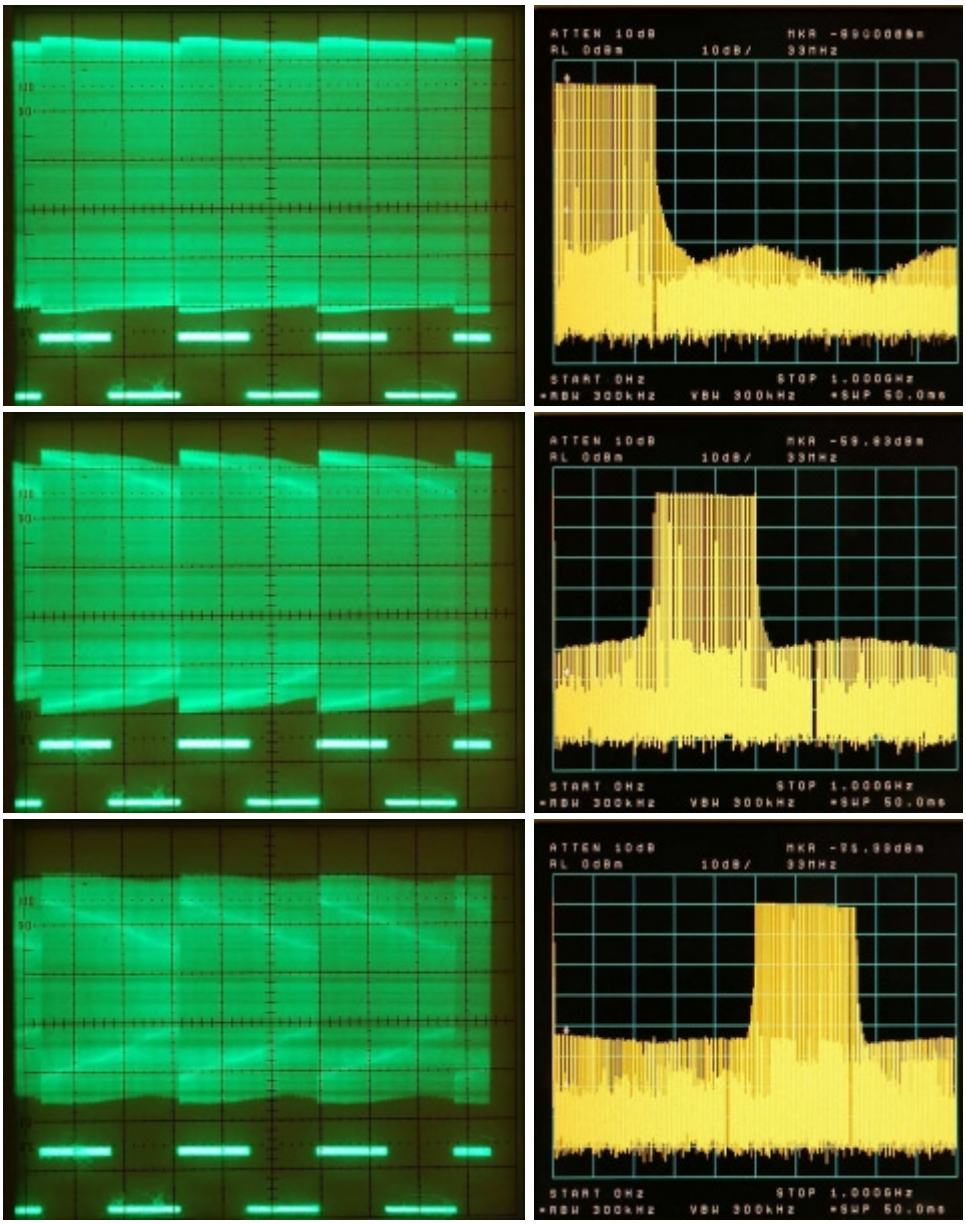



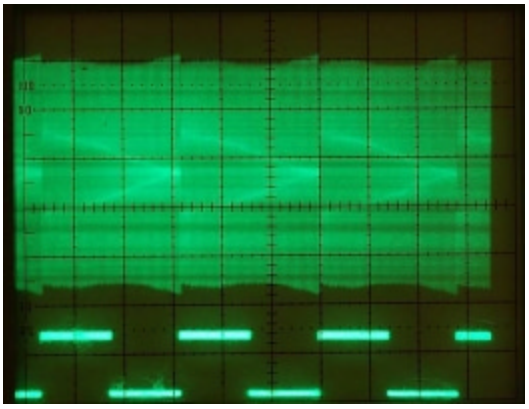
Example 5

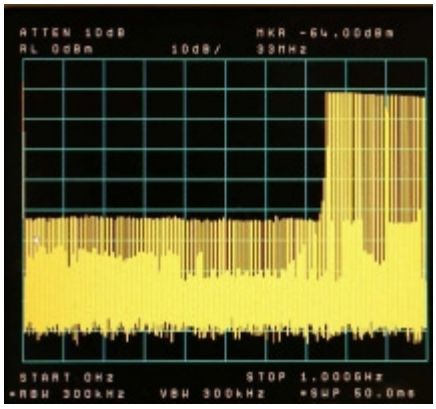
This example utilizes the `user_define_file` method to chirp frequencies that are stored in an external .uwf file.

Example 6

This example makes use of the different pages of the DSM. The DSM is set to 4 pages and the images show the output of each page. The images in order from top to bottom are: Page 0, 1, 2 and 3.





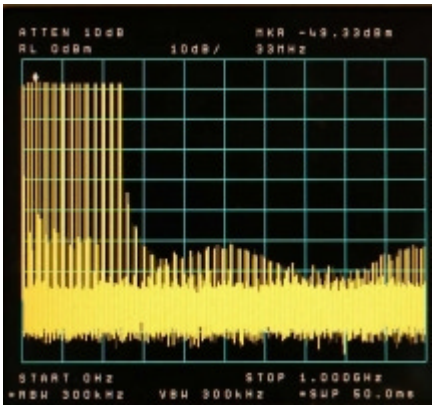


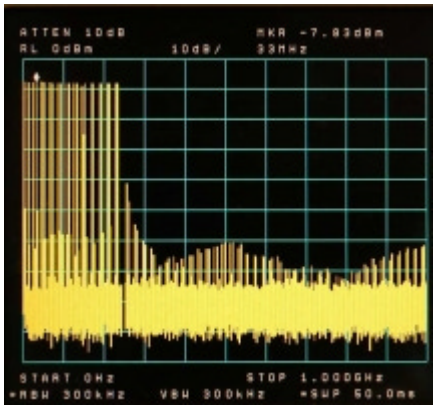
Example 7

This example is just a combination of the above waveforms separated into different pages. The output for each page should resemble the output images shown above.

Example 8

This example demonstrates the multi-board functions of the DSM. The DSM should only run when there is a trigger signal for the Master board and a SYNCI signal for the Slave board. The recommended trigger signal for Master mode is a 1 V peak to peak square wave with a DC offset of 0.5 V. For Slave mode, the only SYNCI signal should be the SYNCO signal from the Master board. The images below show the spectrum for the Master board on the left and the Slave board on the right.





New Features -

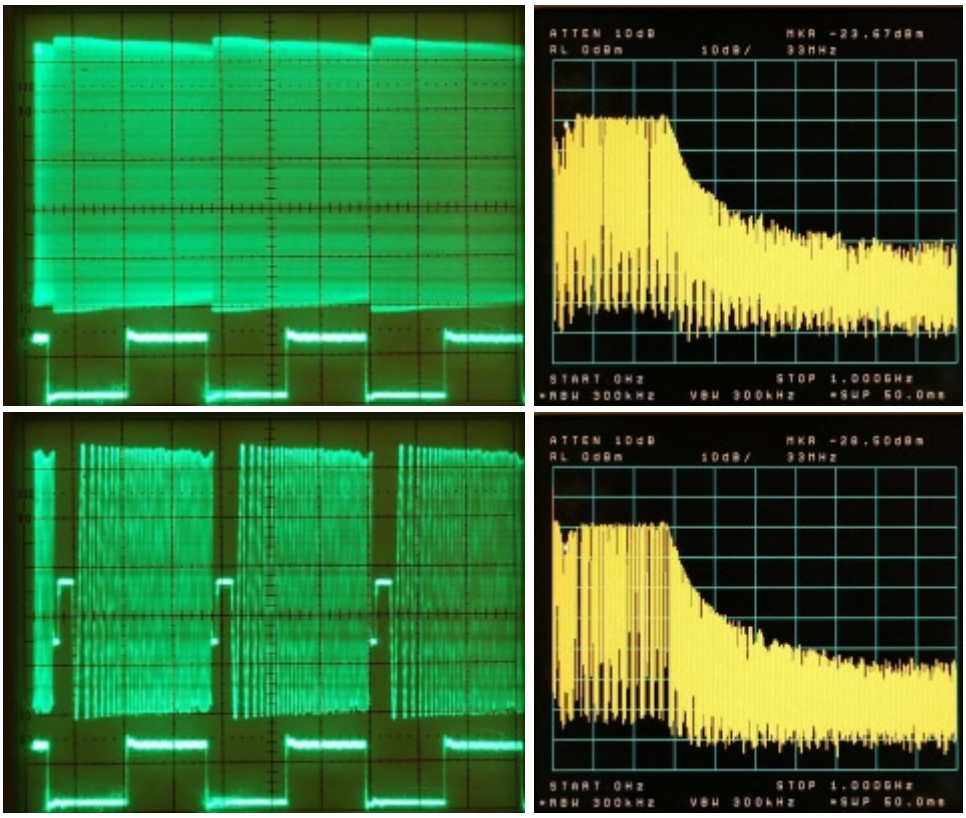
This example will demonstrate the new features of the API, mainly the DDS phase reset and reverse chirping capabilities.

For convenience here is the source code:

new.cpp

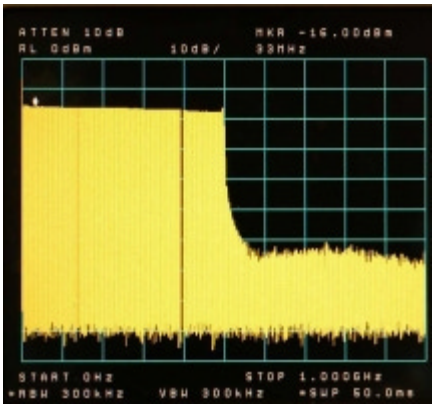
Example 1

This example demonstrates the new phase reset by memory feature. The top images are that of a waveform with no phase reset by memory. The waveform is unclear since the scope is unable to get a phase lock even with trigger by marker. The bottom images show the same waveform but this time with phase reset by memory turned on. Now, every time the DSM starts to chirp, the waveform will start at the same phase. The oscilloscope is now able to get a lock on to the waveform when triggered by marker.



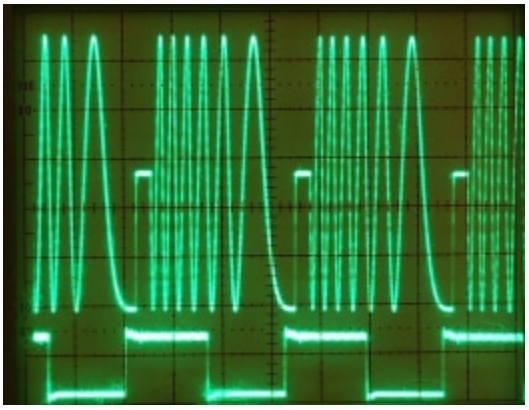
Example 2

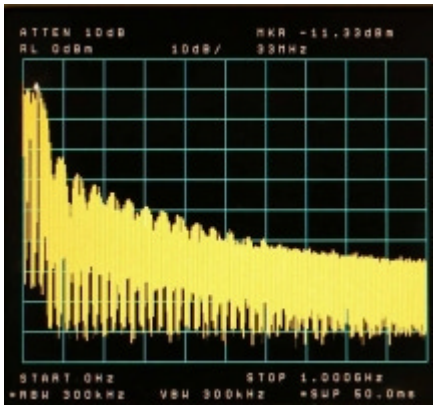
This example shows a waveform that uses the new overloaded `user_define_bulk` method.



Example 3

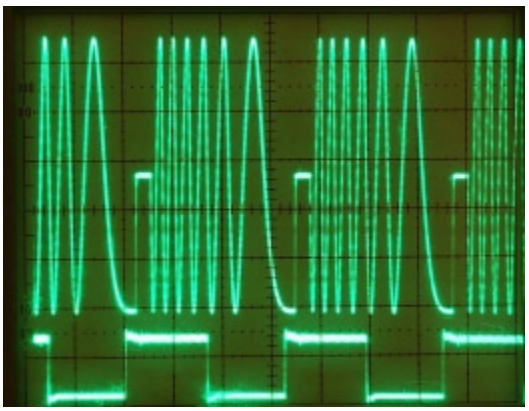
This example demonstrates the reverse chirping feature.

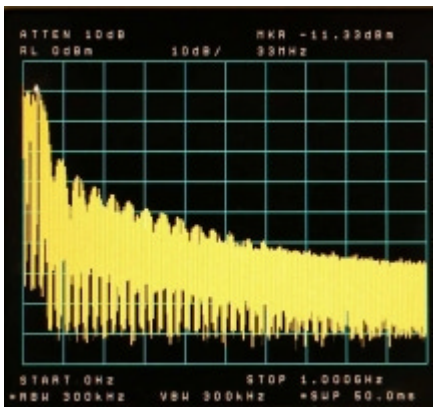




Example 4

This example does the same thing as Example 3 but uses the `user_define_bulk` method instead of the waveform built-in properties.





Appendix -

Jan 30, 2008

New API version 2.2 updates:

- Most parser commands have been omitted from manual but they still function. All commands have been replaced with either methods or properties. If you used parser commands previously please look in the Methods and Properties section to see the new commands.
- Two additional status bits have been added to the MSB. The two new status bits are "Data Length Enable" and "Infinite Loop". Please see the `status` property page for more information.
- There is no need to send the "r2" command for Memory Depth. To set Memory Depth you only need to set the `memory depth` property.
- New phase reset by memory option added.
- New waveform code "2" for reverse chirping.
- New overloaded `user define bulk` method. Please see the method page for more information.
- New examples in Examples section with more detailed code and utilizing an intermediate class to interact with the API methods and properties.

DSM A1

For users who have the DSM A1 board, there are some properties that are either unavailable to you or will require adjustments. The commands are detailed below:

- All multi-board commands are unavailable.
- The status command is unavailable.
- The memory DLL command is unavailable.
- The start and stop memory commands will behave a little bit differently.
- The default oversampling rate should be set to "2" since the clock divider on the A1 board is different.