Winrad - specifications for the external I/O DLL

by Alberto I2PHD

Winrad was initially born as a program taking its input from the sound card and without any provision for controlling external hardware. Soon it appeared that it would have been quite useful to be able to receive the input data also from other sources (notably the USB port), and to send out the information on the frequency set by the user. This would make possible to integrate Winrad with existing or planned external hardware, to better implement its final goal, i.e. to be the software part of a Software Defined Radio.

Of course it would have been quite impractical to code and to maintain a different version of Winrad for each and every different hardware to be supported, so the solution devised was that of placing an intermediate layer of code between Winrad itself and the hardware. Winrad "talks" in a standardized way to this layer of code, which in turn exists in various incarnations, each tailored to a specific hardware.

This document documents (in a still tentative way) the interface between Winrad and that layer of code, which has the form a Windows DLL, dynamically loaded by Winrad if found in its installation directory. Winrad at startup looks in its directory for a file of the form :

ExtIO_*.dll

where the asterisk means that the characters at that place are of no importance, and can be used as a mnemonic aid to indicate which hardware the DLL is meant for.

Should more than one such files be present in the directory, Winrad will ask the user, when initially loading, which one it must use. If found, Winrad dynamically loads the DLL, and calls its initialization entry point. In the rest of this document are described the mandatory and the optional entry points that the DLL must implement, together with the relevant parameters.

Calling conventions

DLLs can be written using different calling conventions. What Winrad expects is exemplified in this code snippet :

```
extern "C"
bool __stdcall __declspec(dllexport) InitHW(char *name, char *model, int& index)
{
   ..............
}
```

Of course the function name InitHW is just an example, as are the parameters and the return type. What is important to be noted here are the keywords **__stdcall** and **__declspec(dllexport)**, and the fact that name mangling is prevented by the use of the **extern "C"** clause.
If you use the Visual C++ compiler from Microsoft to compile your DLL, be sure to read the relevant note at the end of this document.

Synopsis

InitHW

```
bool __stdcall __declspec(dllexport) InitHW(char *name, char *model, int& type)
```

This entry is the first called by Winrad at startup time, and it is used both to tell to the DLL that it is time to initialize the hardware, and to get back a descriptive name and model (or Serial Number) of the HW, together with a type code.

Parameters :

name

descriptive name of the hardware. Preferably not longer than about 16 characters, as it will be used in a Winrad menu.

model

model code of the hardware, or its Serial Number. Keep also this field not too long, for the same reason of the previous one.

type

this is an index code that Winrad uses to identify the hardware type supported by the DLL.
Please use one the following values :

3

the hardware does its own digitization and the audio data are returned to Winrad via the callback device. Data must be in 16-bit (short) format, little endian.

4

The audio data are returned via the sound card managed by Winrad. The external hardware just controls the LO, and possibly a preselector, under DLL control.

5

the hardware does its own digitization and the audio data are returned to Winrad via the callback device. Data are in 24-bit integer format, little endian.

6

the hardware does its own digitization and the audio data are returned to Winrad via the callback device. Data are in 32-bit integer format, little endian.

7

the hardware does its own digitization and the audio data are returned to Winrad via the callback device. Data are in 32-bit float format, little endian.

Please ask me (i2phd@weaksignals.com) for the assignment of an index code for cases different from the above.

Return value :

true

everything went well, the HW did initialize, and the return parameters have been filled.

false

the HW did not initialize (error, or powered off, or other reasons).

OpenHW

```
bool __stdcall __declspec(dllexport) OpenHW(void)
```

This entry is called by Winrad each time the user specifies that Winrad should receive its audio data input through the hardware managed by this DLL, or, if still using the sound card for this, that the DLL must activate the control of the external hardware. It can be used by the DLL itself for delayed init tasks, like, e.g., the display of its own GUI, if the DLL has one.

It has no parameters.

Return value :

true
: everything went well.

false
: some error occurred, the external HW cannot be controlled by the DLL code.

StartHW

```
int __stdcall __declspec(dllexport) StartHW(long freq)
```

This entry is called by Winrad each time the user presses the Start button on the Winrad main screen, after having previously specified that the DLL is in control of the external hardware.

Parameters :

freq
: an integer specifying the frequency the HW should be set to, expressed in Hz.

Return value :

An integer specifying how many I/Q pairs are returned by the DLL each time the callback function is invoked (see later). This information is used of course only when the input data are not coming from the sound card, but through the callback device.
If the number is negative, that means that an error has occurred, Winrad interrupts the starting process and returns to the idle status.
The number of I/Q pairs must be at least 512, or an integer multiple of that value,

StopHW

```
void __stdcall __declspec(dllexport) StopHW(void)
```

This entry is called by Winrad each time the user presses the Stop button on the Winrad main screen. It can be used by the DLL for whatever task might be needed in such an occurrence. If the external HW does not provide the audio data, being, e.g., just a DDS or some other sort of an oscillator, typically this call is a No-op. The DLL could also use this call to hide its GUI, if any.
If otherwise the external HW sends the audio data via the USB port, or any other hardware port managed by the DLL, when this entry is called, the HW should be commanded by the DLL to stop sending data.

It has no parameters and no return value.

CloseHW

```
void __stdcall __declspec(dllexport) CloseHW(void)
```

This entry is called by Winrad when the User indicates that the control of the external HW is no longer needed or wanted. This is done in Winrad by choosing ShowOptions | Select Input then selecting either WAV file or Sound Card. The DLL can use this information to e.g. shut down its GUI interface, if any, and possibly to put the controlled HW in a idle status.

It has no parameters and no return value.

SetHWLO

```
int __stdcall __declspec(dllexport) SetHWLO(long LOfreq)
```

This entry point is used by Winrad to communicate and control the desired frequency of the external HW via the DLL. The frequency is expressed in units of Hz. The entry point is called at each change (done by any means) of the LO value in the Winrad main screen.

Parameters :

LOfreq
: a long integer specifying the frequency the HW LO should be set to, expressed in Hz.

Return values :

0
: The function did complete without errors.

< 0 (a negative number N)
: The specified frequency is lower than the minimum that the hardware is capable to generate. The absolute value of N indicates what is the minimum supported by the HW.

> 0 (a positive number N)

The specified frequency is greater than the maximum that the hardware is capable to generate. The value of N indicates what is the maximum supported by the HW.

GetHWLO

```
long __stdcall __declspec(dllexport) GetHWLO(void)
```

This entry point is meant to query the external hardware's set frequency via the DLL.. It is used by Winrad to handle a asynchronous status of 101 (see below the callback device), but not checked at startup for its presence.

The return value is the current LO frequency, expressed in units of Hz.

GetHWSR

```
long __stdcall __declspec(dllexport) GetHWSR(void)
```

This entry point is used to ask the external DLL which is the current value of the sampling rate. If the sampling rate is changed either by means of a hardware action or because the user specified a new sampling rate in the GUI of the DLL, Winrad must be informed by using the callback device (described below).

The return value is the value of the current sampling rate expressed in units of Hz.

GetTune

```
long __stdcall __declspec(dllexport) GetTune(void)
```

This entry point is meant to query the DLL about the Tune value that Winrad should set.. It is used by Winrad to handle a asynchronous status of 105 (see below the callback device), but not checked at startup for its presence.

The return value is the desired Tune frequency, expressed in units of Hz.

GetFilters

```
void __stdcall __declspec(dllexport) GetFilters(int& loCut, int& hiCut, int& pitch)
```

This entry point is meant to query the DLL about the new passband filter and the CW pitch to set.

Parameters :

loCut
: a reference to an int that will receive the new low cut frequency, in Hertz

hiCut
: a reference to an int that will receive the new high cut frequency, in Hertz

pitch
: a reference to an int that will receive the new CW pitch frequency, in Hertz

GetMode

```
char __stdcall __declspec(dllexport) GetMode(void)
```

This entry point is meant to query the DLL about the demodulation mode that Winrad should use.. It is used by Winrad to handle a asynchronous status of 106 (see below the callback device), but not checked at startup for its presence.

The return value is a single character indicating the desired demodulation mode, according to the following table

'A' = AM 'E' = ECSS 'F' = FM 'L' = LSB 'U' = USB 'C' = CW 'D' = DRM

ModeChanged

```
void __stdcall __declspec(dllexport) ModeChanged(char mode)
```

This entry point is meant to inform the DLL about the demodulation mode being currently set by Winrad.. It is invoked by Winrad each time the demodulation mode is changed, or initially set when the program starts. It is not checked at startup for its presence.

Parameter

mode

a single character indicating the current demodulation mode. For the meaning of the character, see the API GetMode()

GetStatus

```
int __stdcall __declspec(dllexport) GetStatus(void)
```

This entry point is meant to allow the DLL to return a status information to Winrad, upon request. Presently it is never called by Winrad, though its existence is checked when the DLL is loaded. So it must implemented, even if in a dummy way. It is meant for future expansions, for complex HW that implement e.g. a preselector or some other controls other than a simple LO frequency selection.

The return value is an integer that is application dependent.

ShowGUI

```
void __stdcall __declspec(dllexport) ShowGUI(void)
```

This entry point is used by Winrad to tell the DLL that the user did ask to see the GUI of the DLL itself, if it has one. The implementation of this call is optional

It has no return value.

HideGUI

```
void __stdcall __declspec(dllexport) HideGUI(void)
```

This entry point is used by Winrad to tell the DLL that it has to hide its GUI, if it has one. The implementation of this call is optional

It has no return value.

IFLimitsChanged

```
void __stdcall __declspec(dllexport) IFLimitsChanged(long low, long high)
```

This entry point is used by Winrad to communicate to the DLL that the user, through the Winrad GUI, has changed the span of frequencies visible in the Winrad spectrum/waterfall window. The information can be used by the DLL to change the LO value, so to implement a continuous tuning. For example, when the frequency tuned by the user is approaching the lower or higher limit (and this can be known via the TuneChanged API), the DLL could decide to alter the LO value so to bring the tuned frequency in the center of the window. Of course, Winrad must be informed of the change via the callback device, with status code 104. This will ensure that the tuned frequency value will not change.

Parameters:

low
: a long integer specifying the lower limit, expressed in Hz, of the visible spectrum/waterfall window

high
: a long integer specifying the upper limit, expressed in Hz, of the visible spectrum/waterfall window

It has no return value.

TuneChanged

```
void __stdcall __declspec(dllexport) TuneChanged(long freq)
```

This entry point is used by Winrad to communicate to the DLL that a change of the tuned frequency (done by any means) has taken place. This change can be used by a DLL that controls also a TX to know where to set the frequency of the transmitter part of the hardware. The implementation of this call is optional.

Parameters :

freq
: a long integer specifying the new frequency where Winrad is tuned to, expressed in Hz.

It has no return value.

SetCallback

```
void __stdcall __declspec(dllexport) SetCallback(void (* Callback)(int, int, float, short *))
```

This entry point is used by Winrad to communicate to the DLL the function address that it should invoke when a new buffer of audio data is ready, or when an asynchronous event must be communicated by the DLL. Of course the new buffer of audio data is only sent by DLLs that control HW that have their own internal digitizers and do not depend on the soundcard for input. In this case it's up to the DLL to decide which I/O port is used to read from the HW the digitized audio data stream. One example is the USB port. If you don't foresee the need of an asynchronous communication started from the DLL, simply do a return when Winrad calls this entry point.

The callback function in Winrad that the DLL is expected to call, is defined as follows :

```
void extIOCallback(int cnt, int status, float IQoffs, short IQdata[])
```

Parameters :

cnt

is the number of samples returned. As the data is complex (I/Q pairs), then there are two 16 bit values per sample. If negative, then the callback was called just to indicate a status change, no data returned. Presently Winrad does not use this value, but rather the return value of the StartHW() API, to allocate the buffers and process the audio data returned by the DLL. The cnt value is checked only for negative value, meaning a status change.

status

is a status indicator (see the call GetStatus). When the DLL detects a HW change, e.g. a power On or a power Off, it calls the callback function with a cnt parameter negative, indicating that no data is returned, but that the call is meant just to indicate a status change.
Currently the status parameter has just two implemented values (apart from those used by the SDR-14/SDR-IQ hardware) :

100

This status value indicates that a sampling frequency change has taken place, either by a hardware action, or by an interaction of the user with the DLL GUI.. When Winrad receives this status, it calls immediately after the GetHWSR() API to know the new sampling rate.

101

This status value indicates that a change of the LO frequency has taken place, either by a hardware action, or by an interaction of the user with the DLL GUI.. When Winrad receives this status, it calls immediately after the GetHWLO() API to know the new LO frequency.

102

This status value indicates that the DLL has temporarily blocked any change to the LO frequency. This may happen, e.g., when the DLL has started recording on a WAV file the incoming raw data. As the center frequency has been written into the WAV file header, changing it during the recording would be an error.

103

This status value indicates that changes to the LO frequency are again accepted by the DLL

104 ********** **CURRENTLY NOT IMPLEMENTED YET** **********

This status value indicates that a change of the LO frequency has taken place, and that Winrad should act so to keep the Tune frequency unchanged. When Winrad receives this status, it calls immediately after the GetHWLO() API to know the new LO frequency

105

This status value indicates that a change of the Tune frequency has taken place, either by a hardware action, or by an interaction of the user with the DLL GUI.. When Winrad receives this status, it calls immediately after the GetTune() API to know the new Tune frequency. The TuneChanged() API is not called when setting the new Tune frequency

106

This status value indicates that a change of the demodulation mode has taken place, either by a hardware action, or by an interaction of the user with the DLL GUI.. When Winrad receives this status, it calls immediately after the GetMode() API to know the new demodulation mode. The ModeChanged() API is not called when setting the new mode.

107

This status value indicates that the DLL is asking Winrad to behave as if the user had pressed the Start button. If Winrad is already started, this is equivalent to a no-op.

108

This status value indicates that the DLL is asking Winrad to behave as if the user had pressed the Stop button. If Winrad is already stopped, this is equivalent to a no-op.

109

This status value indicates that the DLL is asking Winrad to change the passband limits and/or the CW pitch. When Winrad receives this status, it calls immediately the GetFilters API .

Upon request from the DLL writer, the status flag could be managed also for other kinds of external hardware events.

IQoffs

If the external HW has the capability of determining and providing an offset value which would cancel or minimize the DC offsets of the two outputs, then the DLL should set this parameter to the specified value. Otherwise set it to zero.

IQdata

This is a pointer to an array of samples where the DLL is expected to place the digitized audio data in interleaved format (I-Q-I-Q-I-Q etc.) in little endian ordering. The number of bytes returned must be equal to IQpairs * 2 * N, where IQpairs is the return value of the StartHW() API, and N is the sizeof() of the type of data returned, as specified by the 'type' parameter of the InitHW() API.

RawDataReady

```
void __stdcall __declspec(dllexport) RawDataReady(long samprate, int *Ldata,
                                                  int *Rdata, int numsamples);
```

This entry point is used by Winrad to communicate to the DLL the raw audio data just acquired via whatever method (sound card, WAV file, external HW). This can be used by the DLL either to plot the data or to pre-process them, before Winrad has any chance to act on them. Beware that any processing done by this call adds to the buffer processing time of Winrad, and, if too long, could cause audio buffer overflow with audio glitches and interruptions. Keep the processing done inside this call to a minimum.

Parameters :

samprate
: is the current value of the sampling rate, expressed in Hz.

Ldata
: Is the pointer to a buffer of 32 bit integers, whose 24 low order bits contain the digitized values for the left channel. The 8 high order bits are set to zero. The number of samples in the buffer is given by the fourth parameter.

Rdata
: Is the pointer to a buffer of 32 bit integers, whose 24 low order bits contain the digitized values for the right channel. The 8 high order bits are set to zero. The number of samples in the buffer is given by the fourth parameter.

numsamples
: Is an integer that indicates how many samples are in each of the left and right buffers.

It has no return value.

==========

This concludes the synopsis of the APIs that a DLL must support to be interfaced with Winrad. Note that only V1.32 and above of Winrad is compliant with all of them. Previous versions are guaranteed to not support some of the above functionalities.

In the following pages you can find a skeleton for a very simple DLL, that you can customize to build your own.

Please report any inconsistencies you may have found in the present document, together with suggestions to make it clearer. Thanks.

Good luck

73 Alberto I2PHD

Main.h file

```
//---------------------------------------------------------------------------
#ifndef mainH
#define mainH
//---------------------------------------------------------------------------
#include <stdio.h>

    long   LOfreq;

#endif
```

Main.cpp file

```
//---------------------------------------------------------------------------
#include <windows.h>

#include "main.h"
//---------------------------------------------------------------------------

#pragma argsused

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
        return 1;
}
//---------------------------------------------------------------------------
extern "C"
bool __stdcall __declspec(dllexport) InitHW(char *name, char *model, int& type)
{
   static bool first = true;

   type = 4;    // 4 ==> data returned via the sound card

   if(first)
   {
     first = false;
     LOfreq = 7050000;     // just a default value

     .......... init here the hardware controlled by the DLL

      ......... init here the DLL graphical interface, if any ........

   }

   strcpy(name, "ACME HW");         // change with the name of your HW
   strcpy(model, " Mod. AC456");    // change with the model of your HW

   return true;
}
//---------------------------------------------------------------------------

extern "C"
bool __stdcall __declspec(dllexport) OpenHW(void)
{
   .... display here the DLL panel ,if any....
   .....if no graphical interface, delete the following statement
```

```
   ::SetWindowPos(F->handle, HWND_TOPMOST,
                  0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
// in the above statement, F->handle is the window handle of the panel displayed
// by the DLL, if such a panel exists

   return true;
}
//---------------------------------------------------------------------------
extern "C"
int __stdcall __declspec(dllexport) StartHW(long LOfreq)
{
   ……. Set here the frequency of the controlled hardware to LOfreq

   return 0;            // number of complex elements returned each
                        // invocation of the callback routine
}
//---------------------------------------------------------------------------
extern "C"
int __stdcall __declspec(dllexport) GetStatus(void)
{
   return 0;  // status not supported by this specific HW,
}
//---------------------------------------------------------------------------
extern "C"
void __stdcall __declspec(dllexport) StopHW(void)
{
   return;  // nothing to do with this specific HW
}
//---------------------------------------------------------------------------
extern "C"
void __stdcall __declspec(dllexport) CloseHW(void)
{

  ..... here you can shutdown your graphical interface, if any.....

}
//---------------------------------------------------------------------------
extern "C"
int __stdcall __declspec(dllexport) SetHWLO(long freq)
{
   LOfreq = freq;

   ..... set here the LO frequency in the controlled hardware
   return 0; // return 0 if the frequency is within the limits the HW can generate
}
//---------------------------------------------------------------------------
extern "C"
long __stdcall __declspec(dllexport) GetHWLO(void)
{
   return LOfreq;
}
//---------------------------------------------------------------------------
extern "C"
long __stdcall __declspec(dllexport) GetHWSR(void)
{
   ……. This DLL controls just an oscillator, not a digitizer…

   return 0;
}
//---------------------------------------------------------------------------
```

```
extern "C"
void __stdcall __declspec(dllexport) IFLimitsChanged(long low, long high)
{
      Do whatever you want with the information about the new limits of the
      spectrum/waterfall window

   return;
}
//--------------------------------------------------------------------------

extern "C"
void __stdcall __declspec(dllexport) TuneChanged(long freq)
{
   …. Do whatever you want with the information about the new frequency tuned by
      the user

   return;
}
//--------------------------------------------------------------------------

extern "C"
void __stdcall __declspec(dllexport) SetCallback(void (* Callback)(int, int, float,
short *))
{
   return;   // this HW does not return audio data through the callback device
             // nor it has the need to signal a new sampling rate.
}
//--------------------------------------------------------------------------

extern "C"
void __stdcall __declspec(dllexport) ShowGUI(void)
{
   ……. If the DLL has a GUI, now you have to make it visible

   return;
}
//--------------------------------------------------------------------------

extern "C"
void __stdcall __declspec(dllexport) HideGUI(void)
{
   ……. If the DLL has a GUI, now you have to hide it

   return;
}
//--------------------------------------------------------------------------


extern "C"
void __stdcall __declspec(dllexport) RawDataReady(long samprate, int *Ldata,
                                                  int *Rdata, int numsamples);
{
…….. we don’t know what to do with the raw audio data just passed from Winrad.
…….. We also could not implement this entry point.

   return;
}
//--------------------------------------------------------------------------
```

To summarize, the only mandatory entry points, those that *must* be implemented, and whose presence is checked by Winrad, are the following :

```
InitHW      OpenHW      StartHW     StopHW      CloseHW     SetHWLO     GetStatus
SetCallback
```

The remaining entry points :

```
GetHWLO     GetHWSR     ShowGUI     HideGUI     RawDataReady      TuneChanged
IFLimitsChanged         GetTune     GetMode     GetFilters        ModeChanged
```

are optional, and can be implemented only if needed by the DLL implementer.

Note that the above source code is meant to be compiled by the Borland C++ compiler. If your compiler is different, probably some tweaking will be needed. This is particularly true for the case of the Visual C++ compiler from Microsoft, which presents a small problem, easily circumvented.

The problem resides in the keyword `__stdcall`
The Borland compiler needs it, as the parameters and the stack are managed in accord with the conventions established by the use of that keyword. But the Visual C++ compiler, when it sees that keyword, decides of its own to "decorate" the entry point names… so e.g. the entry name `InitHW` becomes `_InitHW@12`
To prevent this, you must explicitly say to the VC compiler to stop doing that, and this can be obtained with the use of a .DEF file, to be added to the VC project for your DLL. The .DEF file must have this content :

```
LIBRARY     <myDLLname>.DLL

EXPORTS
    CloseHW                     = _CloseHW@0
    GetHWLO                     = _GetHWLO@0
    GetHWSR                     = _GetHWSR@0
    GetStatus                   = _GetStatus@0
    GetTune                     = _GetTune@0
    GetMode                     = _GetMode@0
    GetFilters                  = _GetFilters@12
    ModeChanged                 = _ModeChanged@1
    InitHW                      = _InitHW@12
    OpenHW                      = _OpenHW@0
    RawDataReady                = _RawDataReady@16
    SetCallback                 = _SetCallback@4
    SetHWLO                     = _SetHWLO@4
    ShowGUI                     = _ShowGUI@0
    HideGUI                     = _HideGUI@0
    StartHW                     = _StartHW@4
    StopHW                      = _StopHW@0
    TuneChanged                 = _TuneChanged@4
    IFLimitsChanged             = _IFLimitsChanged@8
```

Where of course <myDLLname> must be replaced with the name you give to your DLL The effect of this .DEF file is to define aliases for the "decorated" entry points, so that they can be found when Winrad dynamically loads the DLL.

==========