Spectrum challenge: Contestant database interface

This document describes the interface provided to contestant radios. The interfaces described here list the C functions. Bindings for National Instruments LabVIEW are also provided. The LabVIEW blocks have the same names, return values and parameters as the functions listed here.

The first step is creating a context, which is used for internal state tracking. This is done as follows:

spectrum* ctx = spectrum_init(char debug);

The function's only parameter should be 0 in normal operation, and the return value is a pointer to the context, is should simply be passed to the other functions and should be finally destroyed using the 'spectrum_delete' function.

The second step a contestant radio should perform is connecting to the central server. This is done using the following function:

spectrum_eror_t spectrum_connect(spectrum* ctx, char* hostname, uint16_t port, char* contestantId, char* contestantPassword);

The parameters of the function are as follows:

Hostname	The hostname to connect to.
Port	The port to use when connecting.
ContestantId	The username of the contestant.
ContestantPassword	The password of the contestant.

All required values above are provided at the start of the challenge. A testing library will be made available so you can test your system in advance.

The function will block, and will give one of the following return values:

ERROR_OK	The client is successfully connected to the management system.
ERROR_INVALID	One or more of the parameters were invalid (eg. NULL)
ERROR_AUTH	The user name or password is incorrect.
ERROR_TIMEOUT	The connection attempt timed out. Verify your network setup.
ERROR_CONNECT_REFUSED	The server refused the connection.
ERROR_OTHER	Something else went wrong.

After connecting, the radio should wait for the start of a stage. A match contains three stages:

1. Radio initialization: During this stage the radio should initialized. RF transmissions are not allowed, and also the primary user also will be silent. The system will continue to the next stage when all radios are ready. Your radio should be ready within 60s.
2. Observation phase: During this phase the primary user will be delivered packets and will start transmitting. During this stage the secondary user can profile the primary user. RF

transmissions are allowed, and potential interference to the PU is no taken into account for the final score. This stage lasts 10 minutes (600 seconds). Note that if all participating radios signal that they are ready for the next stage before the 10 minutes are up the system will continue to the next state.

3. Data transfer phase: All radios will be provided with packets to transmit. Of course, further sensing may be done during this stage. This stage will also last 10 minutes (600 seconds). The final score is only based on performance during this stage.

This is done with the function waitForState:

spectrum_eror_t spectrum_waitForState(spectrum* ctx, uint32_t wantedState, uint32_t timeoutMs);

The parameters are self describing:

wantedState	The state the function should wait for. For example, if your radio is instantly ready to go, you could directly wait for the start of stage 3, ignoring 1 and 2. Settings this to 0 simply waits for the next stage change.
timeoutMs	How many milliseconds to wait. A negative value means infinite.

The function will block, and will give one of the following return values:

Positive value	The server has changed to the current stage.
ERROR_TIMEOUT	The function timed out. The server is still in the current stage.
ERROR_INVALID	One or more of the parameters were invalid.
ERROR_OTHER	Something went wrong

This is a blocking function. Giving a timeout of zero will make the function non-blocking. If you want to exploit the full duration of a stage without polling you can call this function from another thread and send a signal when it returns. All functions are thread-safe, but re-entrant calling should be avoided (although the design is such that it should work, it is not tested deeply).

Now we get to the most important part of the interface: requesting and delivering packets.

The transmitting side of the radio should call the 'spectrum_getPacket' function to request a packet:

spectrum_eror_t spectrum_getPacket(spectrum* ctx, uint8_t* buffer, uint32_t bufferLength, uint32_t timeoutMs);

The parameters of the function are as follows:

buffer	The buffer where the packet will be placed.
bufferLength	The length of the buffer. If the packet is longer than this buffer, no packet will be returned.
timeoutMs	How many milliseconds to wait. A negative value means infinite.

The return value gives either the packet length, or an error:

Positive value	The length of the packet put into the buffer.
ERROR_TIMEOUT	The function timed out.
ERROR_BUF	The buffer is too small for the packet.
ERROR_INVALID	One or more of the parameters were invalid.

This function can block if no packet is available. This should normally not happen during the communication phase. Giving a timeout of zero will make the function non-blocking.

When the receiver has a packet, it should call the function:
spectrum_eror_t spectrum_putPacket(spectrum* ctx, uint8_t* buffer, uint32_t bufferLength);

The parameters of the function are as follows:

buffer	The buffer where the packet is stored.
bufferLength	The length of the packet.

The return value gives either the packet length, or an error:

ERROR_OK	The packet was delivered
ERROR_BUF	The buffer is null or otherwise invalid.
ERROR_INVALID	One or more of the parameters, other than buffer, were invalid.

This function will never block.

The final part of the interface is obtaining the throughput of the active radio links. This is done with the function 'spectrum_getThroughput':

double spectrum_getThroughput(spectrum* ctx, uint8_t radioNumber, double durationMs);

The radioNumber parameter specifies which radio's throughput to request. In a simple primary user versus secondary user scenario, the primary user will be 0, while the secondary user will be 1. To create a generic program, you can request your own radioNumber using 'spectrum_getRadioNumber'. Indeed, you can measure your own radio's performance. Requesting the throughput of a non-existing radio returns 0bps.

DurationMs is used to specify the period over which to average the throughput. The minimum value is 10ms. Shorter times will be treated as 10ms. If the duration is negative or longer than the current game duration, the function will give the average throughput over the entire game.

The return value is the throughput in bps. It can only be greater or equal to zero.

In the same way, one can also request the throughput provided to a radio. This means the number of bytes per second requested by the transmitter. This function works the same as the previous on:

double spectrum_getProvidedThroughput(spectrum* ctx, uint8_t radioNumber, double durationMs);

The main functions have been described above. Now we will handle some auxiliary functions:

- ***void spectrum_delete(spectrum* ctx);***
  You can delete the spectrum object using this function.
- ***void spectrum_errorToText(spectrum* ctx, spectrum_eror_t error, char* output, uint32_t len);***
  This function converts the error value into a human readable string. The string is null terminated.
- ***void spectrum_getStatusMessage(spectrum* ctx, spectrum_eror_t error, char* output, uint32_t len);***
  This function returns a status message. This is used for the LabVIEW GUI. The string is null terminated.
- **spectrum_eror_t spectrum_getRadioNumber*****(spectrum* ctx);***
  You can use this function to get your radio number. A negative value indicates an error.

An example program will be provided which implements a transmitter and a receiver. Instead of transferring the data over the air, a simple UDP stream is used. Of course templates are also provided for LabVIEW and GNURadio.