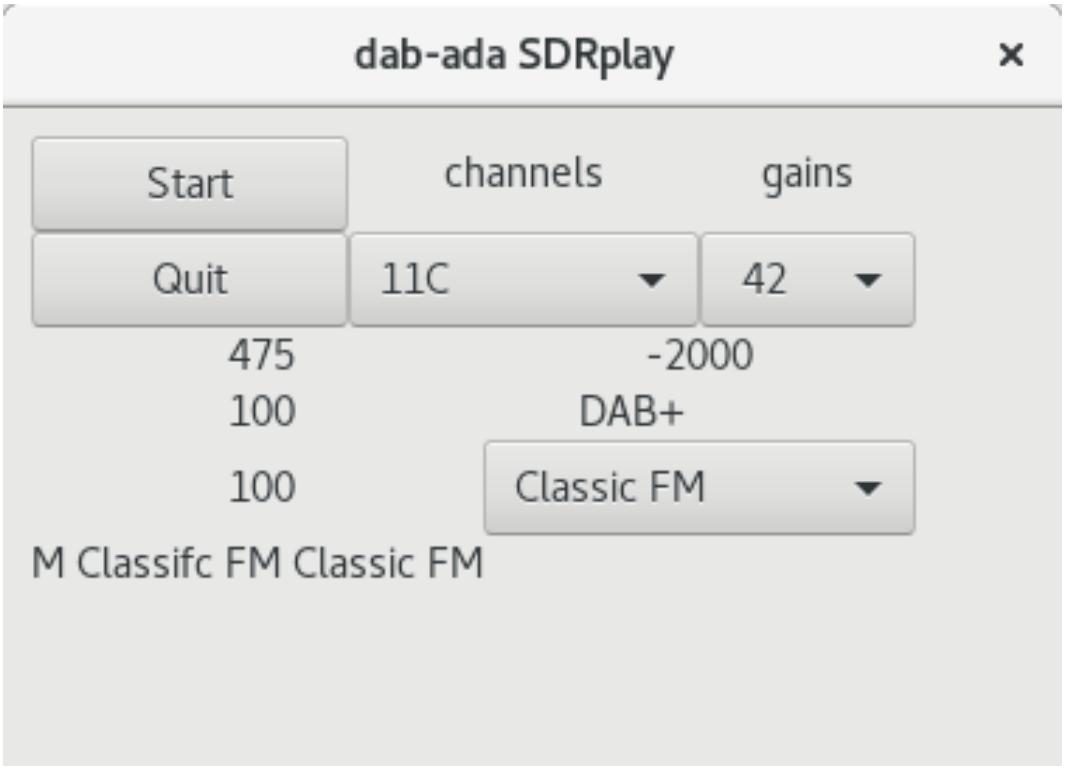
A simple DAB decoder in Ada*

Using Ada for a hobby project

Jan van Katwijk
Lazy Chair Computing
The Netherlands
J.vanKatwijk@gmail.com

July 25, 2017



1 Introduction

There seems to be a growing popularity and interest in DAB[1]. A couple of years ago, when I wrote the first version of the sdr-j-DAB software, the Netherlands "enjoyed" a single ensemble, with the data carried as "DAB'. A few years later, we receive over 4 ensembles in the Netherlands, with an abundant choice of programs, all encoded using the "DAB+" technology. It is said that with the arrival of DAB (here in the old Band III TV band) the "old" FM stations will be phased out, something already done in Norway. And indeed, right now there are more programs on the aforementioned ensembles than on the FM band (at least in my environment).

Being interested in radio technique and in programming, I started to develop a simple DAB decoding program. The program was written in C++, made use of the Qt libraries, and had as input device a so-called dabstick, an rt2832 based device for which there was a library developed in the osmocom project[2].

The DAB program turned out to be quite successful, although it continued to contain some nasty errors, causing the program sometimes to stop, for too long a period.

In 2015 a Raspberry PI 2 was acquired, and an effort was made to install the software on the Raspberry PI 2. Various modifications were needed: the original software overloaded one of the cores of the CPU and some load balancing was needed. While doing that, a silly - though killing - error was detected and repaired[1]

Since that time it happens on a regular basis that the DAB software on the Raspberry runs for 6 to 8 hours continuously, sending the sound to an IP port, allowing me to listen from another place through a simple client. Actually, the normal way for me of using the software is to run the DAB software on a remote Raspberry and listen from within my lazy chair in the living room.

An old interest in the Ada programming language (dating back to the eighties of the previous century) was encouraged by reading some articles on Ada 2005 and Ada 2012. Since Gnat is tightly coupled to the GCC system and available, I decided to recode the basics of the DAB decoder in Ada using Gnat.

One of the interesting parts would be the binding to the C libraries that are required for both handling the input from a device and the sound output.

The result is an operational DAB decoder, interfacing to RT2832 based DABsticks, to the SDRplay and the AIRspy, and allowing the selection of a channel and a program within that channel to prrovide sound.

In this report we discuss the structure and implementation of the Ada DAB decoder.

It is assumed that the reader has - at least - a basic knowledge of the Ada language and a basic understanding of the structure of a DAB datastream.

[1]It turned out that the reed-solomon error repair as applied in the DAB+ superframe was done *after* fetching the addresses in the segments from that segment in the DAB stream. The segment addresses in the superframe therefore were sometimes completely wrong, and error protection was not optimal.

2 Overall structure of the DAB decoder software

Strange as it may sound, the software for the DAB decoder is pretty straightforward. Obviously, one may assume that - through some interface - there is access to samples - sampled at the right speed of 2048000 I/Q samples per second - and a second interface is available through which sound is actually made audible.

Given that there is access to samples, the processing can be thought of as to be done in a few building blocks:

- the ofdm handling, i.e. reading samples, synchronizing the sample stream in time and frequency, analyzing the DAB frames and sending the decoded data blocks to either the *FIC* handler or the *MSC* handler.
- the first few blocks in each DAB frame are the FIC (Fast Information Channel) blocks, the FIC handler is continuously building up a kind of directory structure, containing descriptions of the data that is being sent to the MSC (Main Service Channel) handler.
- the MSC handling depends on the selection of the user: given that a user has selected a program, the MSC handling extracts the relevant data blocks from the stream of MSC data and decodes that data (sound, data or both). If no service is selected, the MSC handling does not do much.
- Although not directly coupled to the processing of the samplestream, the GUI can be considered to be an important component. While the ideas on GUI's differ from person to person, there is some commonality in requirements: in the end the user must be allowed to select a service. A fourth component to be distinguished is therefore the GUI and the GUI handling, with at least a possibility of selecting a service.

Finally, although hardly related to DAB handling, an issue is the control and handling of the input devices and the use of C functions in general. The Ada DAB decoder does support dabsticks, the SDRplay and the AIRspy. The libraries supporting there devices are written in C. Furthermore, the audio output is using the portaudio library[3], the DFT handling the fftw library[4] and the deconvolution also uses an existing C library[5] for the real work.

The compiler used was the Gnat compiler[6], which is available on the Linux box I am using, and the - minimal - interface is built using the GtkAda toolbox[7].

2.1 Main control

The *Mode*, the *Band* and the *Device* may be selected in the command line. As the program starts, these choices are made and cannot be changed during program execution. This simplifies the program considerably, since e.g. changing the Mode requires essentially a reset and a restart of most the program.

All device handlers are derived from a single Controlled type.

```
with header; use header;
with Ada. Finalization; use Ada. Finalization;
package device_handler is
        type device is
            new Ada. Finalization. Controlled with record null; end record;
        type device_P is access all device' Class;
        procedure Restart_Reader   (Object       : in out device;
                                    Success      : out Boolean);
        procedure Stop_Reader      (Object       : in out device);
        procedure Set_VFOFrequency (Object       : in out device;
                                    New_Frequency: Natural);
        procedure Set_Gain         (Object       : in out device;
                                    New_Gain     : Natural);
        procedure Get_Samples      (Object       : in out device;
                                    Out_V        : out complexArray;
                                    Amount       : out Natural);
        function Available_Samples (Object       : device) return Natural;
        function Valid_Device      (Object       : device) return Boolean;
end device_handler;
```

Selecting the handler for a specific device then reduces to matching the parameter passed and allocating the device handler.

```
 .....
                elsif parameter = "airspy" then
                  The_Device := new airspy_wrapper. airspy_device;
 ...
```

Selection of the Band does not have further influence on the other parts of the program. Once selected, the channels for the selected band may be made visible.

```
--the channel_handler, for the selected band
    Channel_Handler. Setup_Channels (Channel_Selector, The_Band);
```

The main components, implemented as generic packages, are instantiated in the main program

```
declare
--here we declare (instantiate) our components, they need
--  to be visible within the callbacks

   package my_mscHandler is new msc_handler (the_Mode);
   package my_ficHandler is new fic_handler (the_Mode);
   package my_ofdmHandler is
                 new ofdm_handler (The_Mode,
                                   The_Device.    Get_Samples,
                                   The_Device.    Available_Samples,
                                   my_mscHandler. process_mscBlock,
                                   my_ficHandler. process_ficBlock,
                                   my_ficHandler. Sync_Reached);
```

The *fic_handler* and *msc_handler* packages need the DAB mode as parameter. The package *ofdm_handler* is the main driver in the DAB software, it gets as parameters - next to the mode - functions to read the samples in (from *The_Device*) and functions to pass on the recognized data blocks (to *my_ficHandler* and *my_mscHandler*).

The GUI is - as mentioned before - implemented using the GtkAda toolkit.

One issue arises with this setup. The GUI functions, that are activated through callbacks when selecting items on the GUI, need access to the interfaces of the packages mentioned above.

As an example, selecting a program from the current ensemble requires access to *my_ficHandler* to obtain specific data on where to find and to decode the program, and then to *my_mscHandler*. The function *Programselector_clicked* is declared in the main program.

```
   procedure Programselector_clicked
                (Self : access Gtk_Combo_Box_Record' Class) is
      Program_Descriptor: audioData;
      El    : String   := Self. Get_Active_Text;
   begin
      if Deleting then
         return;
      end if;

      put ("program "); put (el); put_line ("selected");
      my_ficHandler. Data_for_Audioservice (El, Program_Descriptor);
....   -- code omitted
      if Program_Descriptor. length > 0 and then
                             Program_Descriptor. bitrate > 0 then
         my_mscHandler. set_audioData (Program_Descriptor);
      else
         put_line ("sorry, cannot find the audio right now");
      end if;
   end Programselector_clicked;
```

The procedure will also be invoked when deleting a program name from the GUI (which happens when a different channel is selected). To avoid confusion, a variable *Deleting* is set when selecting a channel, such that the invocation of the procedure quickly returns.

The procedure *Programselector_clicked* will call *my_ficHandler.Data_for_Audioservice* for asking for the relevant data of the program with the given name. It will then, if the data makes sense, pass it on to the package *my_mscHandler*, the implementation of the msc handler.

However, this implies that the functions implementing the call backs only can be created *after* the instantiation of the packages. Since the creation of the GUI is on library level, the scope of the GUI elements is wider than the scope of the callback functions, which is not appreciated by the language.

Fortunately, the Gnat compiler provides the opportunity of an *Unrestricted_Access* attribute, which is dangerous, but applied here in the binding of the callback functions to the GUI elements, which also is executed in the main program.

```
Win.                On_Destroy
                    (main_quit' Unrestricted_Access);
Channel_Selector.   On_Changed
                    (Channelselector_clicked' Unrestricted_Access);
programSelector.    On_Changed
                    (Programselector_clicked' Unrestricted_Access);
startButton.        On_Clicked
                    (start_clicked' Unrestricted_Access);
Gain_Selector.      On_Changed
                    (Gainselector_clicked' Unrestricted_Access);
quitButton.         On_Clicked
                    (button_quit'  Unrestricted_Access);
```

The obvious solution seemed to be to create the GUI only after having instantiated the packages. However, since many parts of the program send data to the GUI for display, it seems that the GUI should be created on library level.

2.2 Structure of the report

The structure of the remainder of this report, which is focussed on handling the datastream rather than the GUI, is as follows:

- in section 3 we discuss the ofdm handling.
- in section 4 we discuss the FIC, the handling, the structure and its implementation.
- in section 5 we discuss the MSC, its structure, the handling and its implementation.
- in section 6 we discuss the output to the soundcard, i.e. the interfacing to the portaudio library.
- in section 7 we discuss the interfacing to the external elements, i.e. devices, fftw library, etc.
- in section 8 we discuss the handling of the devices, with as example a description of the handling of the SDRplay.
- finally, in section 9 we briefly discuss some issues, solved and remaining.

3 The ofdm handling

3.1 Introduction

The ofdm handling - implemented in the package $ofdm\_handler$ - is a major component in this implementation. It basically handles the transform from the data from analog into the digital domain and drives the whole DAB processing. As such it is equipped

to recognize DAB frames, to extract timing and frequency errors in the incoming data stream and correct them, to identify the different data blocks, map them from the time to the frequency domain, extract the bits and send the result to the appropriate handler.

The real work is partitioned into two parts, one executed by the task $ofdm\_Worker$, the other one by the task $ofdm\_decoder$.

When porting the C++ version of the DAB decoder to the Raspberry, it turned out that execution of the ofdm handling functionality in a single thread caused an overload of one of the cores of the RPI processor. While it is still unclear whether the Ada version will run on an RPI or not, it was decided to mimic the (task) structure of the Ada version with that of the C++ version.

3.2 Time and frequency offsets and synchronization

3.2.1 Time synchronization

The time sychronization deals with identifying the first sample in the time domain data stream that marks a DAB frame. The DAB frames in the incoming samplestream provide mechanisms for synchronizing:

- each DAB frame starts with a so-call *null-period*, a period with a predefined length, defined by the DAB mode, where no carrier is transmitted. *Coarse* time synchronization, i.e. finding a reasonable point in the input stream where the *null-period* ends, by looking at the samples,

- the first data block of a DAB frame has - when converted to the frequency domain - a predefined structure, i.e. we know what data to expect in the frequency domain.

It is obvious that by looking at the average signal power we can estimate the start and the end of a *null-period*. Most likely we are a few samples off, but a rough estimate is certainly possible.

The *start* of the *null-period* can be determined by comparing the long term average signal value with the moving average of the last, say, 50 samples. This implies that we have read a sufficiently large amount of samples to "know" the long term average signal value.

```
for I in 0 .. 20 loop
   Get_Samples (Ofdm_Buffer, 0);
end loop;
```

The Tu_Sized_Buffer has T_u elements, where T_u is defined by the mode to be the length of a block in a DAB frame in the time domain. For mode 1 T_u is 2048 samples long.

The long term average signal value is maintained by the $Get\_Samples$ function, each sample read contributes. A similar - though shorter - loop is done for obtaining a decent $Current\_Strength$ value, after which we can look for a "dip".

```
Counter := 0;
while Current_Strength / 50.0 > 0.40 * Signal_Level loop
   declare
      Sample: complexArray (0 .. 0);
   begin
      Get_Samples (sample, Coarse_Corrector + Fine_Corrector);
      Env_Buffer (Syncbuffer_Index) := abs Sample (0);
      Current_Strength      :=
                        Current_Strength + abs Sample (0) -
                                Env_Buffer (Syncbuffer_Index - 50);
      Syncbuffer_Index              := Syncbuffer_Index + 1;
      Counter                       := Counter + 1;
      if Counter > 3 * T_s then -- hopeless
         goto notSynced;
      end if;
   end;
end loop;
```

It is clear that if we do not find a dip within a reasonable number of samples, we are either working with a channel without a decent DAB signal, or we are in the middle of a DAB frame. In both cases we might be far of the start of a DAB frame and have to restart[2].

Once we are in the *null-period*, the end is located using the same approach.
As soon as we have a reasonable estimate on the location of the start of the first datablock, we just collect T_u samples (T_u being the amount of "useful" data in the datablock, a value defined by the DAB mode), and have it processed by a procedure in the class *phaseSynchronizer*. The result is a positive number if - within reason - a start sample could be identified and a negative one if not. The positive number indicates the offset from the beginning of the data passed where the T_u part of the data of the block really begins.

In case of a negative number, we feel a little lost and do the coarse time synchronization again.

```
--  Read in Tu samples and look for the startIndex
       Get_Samples (Reference_Vector, Coarse_Corrector + Fine_Corrector);
       Start_Index   :=  my_Phasehandler. Find_Index (Reference_Vector, 3);
       if Start_Index < 0 then              -- no sync, try again
          goto notSynced;
        end if;
```

Once we are time-synchronized, we fill block 0, take the DFT and use that as reference for decoding the next block.

```
    Reference_Vector (0 .. Tu - Start_Index - 1) :=
                          Reference_Vector (Start_Index .. Tu - 1);
    if Start_Index > 0 then
       Get_Samples (Reference_Vector (Tu - Start_Index .. Tu - 1),
```

[2]In the Qt-DAB program this is used to estimate (guess) whether we have a DAB signal or not

```
                            Coarse_Corrector + Fine_Corrector);
    end if;

--      we  now have the contents of block 0 and
--      set the reference vector for the frame
    my_fft. do_FFT (Reference_Vector);
```

3.2.2 Frequency synchronization

The frequency synchronization deals with identifying the frequency offset of the incoming data stream and correcting this offset. It is good to realize that the frequency offset may be larger than the frequency difference between successive carriers in blocks in the frequency domain. The *coarse* offset will be expressed in the number of carriers, while the frequency offset to the nearest carrier in the frequency domain is called the *fine* offset and expressed just in Hz. The total offset is then the sum of these two.

It is pretty obvious that as when there is a course offset unequal to 0, a carrier i is found where carrier j should be, and further decoding is useless. The coarse offset should be *0*. The fine frequency offset, i.e. the offset from the nearest, correct, carrier is less dangerous, though should be compensated for as much as possible.

Other than e.g. a DRM signal, the data blocks of a DAB frame do not contain carriers with specific characteristics that might help in the frequency synchronization. In the past we experimented with different approaches to determine the coarse offset.

- It is known that the transmitted power of all carriers in the first DAB datablock is the same, the first approach was to "balance", i.e. to find the carrier in the "middle" of the DFT of the first datablock. In theory, the energy in the carriers to the left and to the right is the same. The obvious drawback is that in case of stronger fading of the frequencies in one end of the spectrum would lead to an error in the detected "middle".

- Knowing the contents of the carriers in the DFT of the first datablock of a DAB stream, it should be possible to do some form of correlation with the incoming first datablock. Note however that correlation based on the *energy* contained in the carriers hardly works: all carriers (should) have the same energy content.

  In the Ada version, we address the correlation by looking at the sum of the *phase differences* between predefined carriers as they should be and as they are. Since the phase difference between carriers as they should be show in the range near the carrier zero a distinct pattern, this works well.

The computation of the coarse offset takes place - as mentioned - using the known pattern of phase offsets between given successive carriers. By computing the sum $\sum abs(R_x \times conj(R_{x+k})$ for predefined values of x and k, we decide for which value of x the sum is least, and assume that that x value indicates the position of carrier 0.

```
   function Compute_Offset (Block_0_Buffer: Tu_Sized_Buffer)
                                                 return Integer is
```

```
      Search_Range   : constant Integer := 36;
      Res_Vector     : Tu_Sized_Buffer renames Block_0_Buffer;
      M_min          : Float            := 1000.0;
      Index          : Integer          := Tu;
   begin
-- we look at a special pattern consisting
-- of zeros in the row of args between successive carriers.
      for I in Tu - Search_Range / 2 .. Tu + Search_Range / 2 loop
         declare
            A1: Float := abs (abs  (arg (Res_Vector ((I + 1) mod Tu) *
                           conj (Res_Vector ((I + 2) Mod Tu))) / M_PI) - 1.0);
            A2: Float := abs arg (Res_Vector ((I + 1) mod Tu) *
                                  conj (Res_Vector ((I + 3) Mod Tu)));
            .....   -- similar code here deleted
            B4: Float := abs (arg (Res_Vector ((I + 16 + 5) mod Tu) *
                                   conj (Res_Vector ((I + 16 + 6) mod Tu))));
            Sum: Float := A1 + A2 + A3 + A4 + A5 + B1 + B2 + B3 + B4;
         begin
            if Sum < M_min then
               M_min := Sum;
               Index := I;
            end if;
         end;
      end loop;
      return Index - Tu;
   end Compute_Offset;
```

The range over which this computation is performed is limited, we assume that the frequency offset is less than 18 times the frequency difference between successive carriers[3].

Once the coarse frequency offset is detected and corrected it might be expected that the data can be decoded correctly, and as soon as there is a signal that indeed ensemble and program data is found, we might assume the coarse frequency is correct. Unless we know that synchronization is lost, there is no need for recomputing the coarse offset (computation as such is a cpu intensive operation) it for each DAB frame.

```
--here we look only at computing a coarse offset when needed
--first check
   if not Correction_Flag then
      Correction_Flag := not Sync_Reached;
   end if;
   if Correction_Flag then
      declare
         Correction_Value : Integer :=
                              Compute_Offset (Reference_Vector);
      begin
         if Correction_Value = 0
                  and then Previous_1 = 0
```

[3]Although it is known that the offset measured at cheap DABsticks can be as large as 30 Khz the frequency range of DAB transmissions

```
                and then Previous_1 = Previous_2 then
            Correction_Flag := false;
         elsif Correction_Value /= 100 then
            Coarse_Corrector := Coarse_Corrector +
                                    Correction_Value *
                                    Carrier_Diff;
            if abs Coarse_Corrector > kHz (35) then
               Coarse_Corrector := 0;
            end if;
            Previous_2 := Previous_1;
            Previous_1 := Correction_Value;
         end if;
      end;
   end if;
```

It is obvious that compensating a coarse offset with the correct value leads to a measured offet 0. We assume therefore that when there are three successive zeros for the measured offset with a given correction value, we are done.

Whether we need to compute the coarse offset is indicated by the setting of the variable *Correction_Flag*. Note that in the processing of each frame we ask the FIC handler whether or not we are still synchronized.
The fine frequency offset, however, depends not only on tuning errors, but also on reception conditions, so we continuously need to compute that offset and correct it.
The approach taken here to compute it is a classic one and done with the time domain samplestream. Each data block carried in the DAB frame has a *cyclic prefix*, where the first T_g samples in the block are a copy of the samples $T\_u..T\_s$ in that block.

The fine offset then can be computed by looking at the phase difference between the $sample[i]$ and the $sample[T\_u + i]$. In practice the average is taken of all pairs in the blocks of the DAB frame.

In the end, we compute the frequency offset by
$arg(\sum_i \sum_j block_i[j] * conj(block_i[T\_u + j]))/(2 * \pi) * distance$
where *distance* is the difference - in Hz - between successive carriers.
This newly computed frequency offset is integrated with the previous offsets. This integration may lead to a value larger than half of the *distance*, in wich case the coarse offset is adapted. This all is done after having recognized a full frame, i.e. being ready to handle the next one.

```
    Fine_Corrector   :=  Fine_Corrector +
                        integer (0.1 * arg (Phase_Error) / M_PI *
                                      float (Carrier_diff) / 2.0);

--   OK,  here we are at the end of the frame
--   we assume everything went well and we just skip T_null samples
--   after which we expect the next frame to be visible
     Get_Samples (Null_Buffer, Coarse_Corrector + Fine_Corrector);
    Syncbuffer_Index   := 0;

--   Here we just check the validity of the fineCorrector
```

```
    if Fine_Corrector > Carrier_diff / 2 then
       Coarse_Corrector := Coarse_Corrector +  Carrier_Diff;
       Fine_Corrector   := Fine_Corrector - Carrier_Diff;
    elsif Fine_Corrector < - Carrier_Diff / 2 then
       Coarse_Corrector := Coarse_Corrector - Carrier_Diff;
       Fine_Corrector   := Fine_Corrector + Carrier_Diff;
    end if;
<<ReadyForNewFrame>>
    goto SyncOnPhase;
```

Once we are time-synchronized, there is really no need to look for the *null-period* when processing the next DAB frame. We therefore just skip the amount of samples in the *null-period* and start synchronizing with the first data block. I.e. at the end of recognizing a DAB frame, we merely collect T_null samples and go back looking for a Block 0.

Note that when we are wrong, the synchronization with the first data block will fail, and processing will continue at the label *notSynced*.

3.3 Structure of the ofdm handler

When knowing how to compute the synchronization, the structure of the ofdm handler - implemented in the task $Ofdm\_Worker$ in the package, can be sketched as follows

```
<<Initing>>
-- we just read a lot of samples to get a decent value
-- for the average signal value in Signal_level
......           -- code omitteh
--
--  When we are really out of sync, we will be here
<<notSynced>>
-- we read in 50 values to compute a reasonable value
-- for the moving average of the last 50 samples
......

--  We now have initial values for currentStrength (i.e. the sum
--  over the last 50 samples) and sLevel, the long term average.

<<SyncOnNull>>
--  here we start looking for the null period, i.e. a dip
..... -- code omitted
--
--  It seems we just successfully passed the start of a null period,
--  now start looking for the end of the null period.
--  This "end" should be there within T_null samples,
--  otherwise, just give up and start all over again
<<SyncOnEndNull>>
..... -- code omitted

--  The end of the null period is identified, it ended probably about 40
```

```
--  or 50 samples earlier
<<SyncOnPhase>>

--  We now have to find the exact first sample of the non-null period.
--  We use a correlation that will find the first sample after the
--  cyclic prefix.
--  When in "sync", i.e. pretty sure that we know were we are,
--  we skip the "dip" identification and come here right away.
--  Read in Tu samples and look for the startIndex
..... -- code omitted
    Get_Samples (Reference_Vector, Coarse_Corrector + Fine_Corrector);
    Start_Index   :=  my_Phasehandler.
                                Find_Index (Reference_Vector, 3);
    if Start_Index < 0 then -- no sync, try again
       goto notSynced;
    end if;

.....            -- code omitted
--  we  now have the contents of block 0 and
--  set the reference vector for the frame
    my_fft. do_FFT (Reference_Vector);
--
--  and we update the coarse synchronization if needed
..... -- code omitted
--
--  Here we really start processing the data
    Phase_Error   := (0.0, 0.0);
--
    Ofdm_Decoder. Block_0 (Reference_Vector);

    for Symbolcount in 2 .. L_Mode loop
       Get_Samples (Ofdm_Buffer, Coarse_Corrector + Fine_Corrector);
       for I in 0 .. Tg - 1 loop
            Phase_Error := Phase_Error +
                           conj (Ofdm_Buffer (I)) * Ofdm_Buffer (Tu + I);
       end loop;
       Ofdm_Decoder. Put (Symbolcount, Ofdm_Buffer (Tg .. Ts - 1));
    end loop;

<<NewOffset>>
--  we integrate the newly found frequency error with the
--  existing frequency offset
    Fine_Corrector :=  Fine_Corrector +
                              integer (0.1 * arg (Phase_Error) / M_PI *
                                     float (Carrier_diff) / 2.0);

--  OK,  here we are at the end of the frame
--  we assume everything went well and we just skip T_null samples
--  after which we expect the next frame to be visible
    Get_Samples (Null_Buffer, Coarse_Corrector + Fine_Corrector);
    Syncbuffer_Index := 0;
```

```
--  Here we just check the validity of the fineCorrector

    if Fine_Corrector > Carrier_diff / 2 then
       Coarse_Corrector := Coarse_Corrector +  Carrier_Diff;
       Fine_Corrector   := Fine_Corrector - Carrier_Diff;
    elsif Fine_Corrector < - Carrier_Diff / 2 then
       Coarse_Corrector := Coarse_Corrector - Carrier_Diff;
       Fine_Corrector   := Fine_Corrector + Carrier_Diff;
    end if;
<<ReadyForNewFrame>>
    goto SyncOnPhase;
```

3.3.1 Time synchronization with Block 0

The value of the carriers in Block 0 as they should be, are predefined. These values are helpful in locating the first sample in the time domain data stream that belongs to Block 0.

As we saw earlier, we just read in T_u samples and hand them over to the $Find\_Index$ function of the package $my\_Phasehandler$. This functions returns a positive number if a - reasonable - estimate of the offset of the first sample in the passed data is found, a negative number otherwise. A threshold value (3 was selected) for determining whether the computed value is reasonable, is passed together with the data.

```
    Get_Samples (Reference_Vector, Coarse_Corrector + Fine_Corrector);
    Start_Index   :=  my_Phasehandler.
                                    Find_Index (Reference_Vector, 3);
    if Start_Index < 0 then -- no sync, try again
       goto notSynced;
    end if;
```

The function $Find\_Index$ itself, member of the package generic package $phase\_handler$, computes - through correlation- an estimate of the offset of the first sample in the block. It takes the DFT of the incoming data, then first multiplies this - carrier by carrier - with the conjunct of the value as it should be, and finally takes the inverse DFT of the result. The maximum of this result then indicates the requested value.

We demand that the maximum found is at least *threshold* times (here 3) as large as the average value found in the output of the inverse DFT.

```
   function Find_Index (inputBuffer : bufferType;
                        threshold   : integer) return integer is
       Res_Vector  : complexArray   := inputBuffer;
       Max_Index   : Integer        := -1;
       Sum         : Float          := 0.0;
       Max         : Float          := 0.0;
       Avg         : Float          := 0.0;
    begin
       Forward_fft. do_FFT (Res_Vector);

--   back into the frequency domain, now correlate
```

```
      for I in Res_Vector' range loop
         Res_Vector (I) := Res_Vector (I) *
                              complexTypes. Conjugate (Ref_Table (I));
      end loop;

--   and, again, back into the time domain
      Backward_fft. do_FFT (Res_Vector);
--   normalize and compute the average signal value ...
      for I in Res_Vector' range loop
         Res_Vector (I) := (Res_Vector (I). Re / float (Res_Vector'Length),
                            Res_Vector (I). Im / float (Res_Vector'Length));
         Sum            := Sum + abs Res_Vector (i);
      end loop;
--
      Max     := -10000.0;
      for I in Res_Vector' range loop
         if abs (Res_Vector (I)) > Max then
            Max_Index  := I;
            Max        := abs Res_Vector (I);
         end if;
      end loop;

      Avg     := Sum / float (Res_Vector' length);
--  that gives us a basis for defining the threshold
      if Max < Avg * float (Threshold) then
         return -1;
      else
         return Max_Index;
      end if;
   end Find_Index;
```

3.4 Block 0 and block decoding

Decoding the data blocks, i.e. taking the DFT, perform some de-interleaving and extract the soft bits - is delegated to a separate task, the task $Ofdm\_Decoder$. As mentioned earlier, handling ofdm is quite resourcefull and with this partitioning we follow the partitioning applied in the C++ version.

Ofdm decoding in DAB is based on the $phasedifference$ of a carrier in Block x compared to the carrier on the same position in Block x - 1. The carriers in the first datablock, Block 0[4], act as reference for the carriers in the second block, the carriers in the second block are the references for the carriers in the third block, etc, etc.

The carriers in block 0 have predefined values, these values are used to correlate with the data in the incoming data stream to locate the first sample of the first block[5].

[4]Note that we are inconsistent in that we were talking about the first data block, and after Block 0 block 2 will follow.

[5]Note that the predefined values for the carriers in Block 0 are used in the time synchronization. They are not used as reference values for the decoding of the first data block

3.4.1 Data decoding

At first sight it may seem tempting to use the predefined values of the carriers of Block 0 as reference for the decoding. However, one must take into account that there may be a large phase difference between the predefined carriers and the carriers in the actual Block 0, so that is not advisable.

Data decoding itself is pretty straightforward and delegated to a procedure $process\_Token$. The procedure takes a Tu_sized_Buffer and puts the decoded data into an array, local to the embodying task.

```
procedure process_Token (Buffer  : in out Tu_Sized_Buffer) is
```

For each data block - in the time domain - that arrives, we first have to compute the DFT

```
my_fft. do_FFT (Work_Vector);
```

After which we can compute the phase differences of the (relevant) carriers, e.g. for Mode 1, the T_u size is 2048, while only the 1536 carriers "in the middle" carry useful information. For each DAB mode the number of carriers is defined. The carriers are interleaved in the transmitted datablock, we do the de-interleaving on the fly, for which purpose a function Map_in in the class $freqInterleaver$, addressed as $my\_freqInterleaver$ is available.

```
--  Note that "mapIn" maps to -carriers / 2 .. carriers / 2
--  we did not set the fft output to low .. high,
--  so the order - in processing - remains high .. low
    for I in 0 .. Carriers - 1 loop
       declare
          Index   : Integer := my_freqInterleaver. Map_In (i);
          R1      : complexTypes. complex;
       begin
          if Index < 0 then
             Index := Index + Tu;
          end if;
--
--  this is the data value, we keep the "old" value as reference
--  value for the next block
          R1 := Work_Vector (Index) * conj (Reference_Vector (Index));
          Reference_Vector (Index) := Work_Vector (Index);
```

The mapping from the resulting complex numbers to the resulting bits is straightforward: just look at the quadrant the value is in, extract the real and imaginary component, scale the value in the range -127 .. 127 and pass on the resulting soft bits.

```
--  Recall:  with this viterbi decoder
--  we have 127 = max pos, -127 = max neg, so we scale
          Ibits (I)            := short_Integer (R1. Re / abs R1 * 127.0);
          Ibits (Carriers + i) := short_Integer (R1. Im / abs R1 * 127.0);
       end;
    end loop;
```

Once the decoding is done, all that remains is passing the data on to either the FIC handler or the MSC handler

```
accept Put (Blkno : Natural; Data : Tu_Sized_Buffer) do
   Help  := Data;
   Block := Blkno;
end Put;
process_Token (Help);
if Block <= 4 then
   process_ficBlock (Ibits, Block);
else
   process_mscBlock (Ibits, Block);
end if;
```

3.5 Handling the input

While functions to access the handler for the associated device are passed as parameter, a local interface function is used to get input to where we want it. The function, *Get_Samples*, takes as parameter (i) the buffer in which the samples are to be written and (ii) the correction term for the frequency.

The function, when called, will check whether or not the task should be running. If not, it will raise an exception that will be handled by the task (the task is the sole client of the function), and causes the task to terminate.

```
if not Running then
   raise Exit_ofdmProcessing;
end if;
```

The function will read - as soon as possible - the requested amount of samples from the buffer in the device handler. The samples read need to be corrected for the frequency offset. For this purpose, the package $ofdm\_handler$ contains a - pretty large - vector $OscillatorTable$, that is initialized with 2048000 complex I/Q values.

```
for i in OscillatorTable' Range loop
   OscillatorTable (i) :=
          (Math. cos (float (i) * 2.0 * M_PI / float (Input_Rate)),
           Math. sin (float (i) * 2.0 * M_PI / float (Input_Rate)));
end loop;
```

Correction then is by multiplying incoming samples with elements from this table, decrementing[6] the phase with each step.

```
for I in Out_V' Range loop
   Current_Phase := (Current_Phase - Phase_Ind) mod Input_Rate;
   Out_V (I)             := Out_V (I) *
                    OscillatorTable (Current_Phase);
   Signal_Level  := 0.00001 * abs Out_V (I) +
                            (1.0 - 0.00001) * Signal_Level;
end loop;
```

[6]We decrement the phase since we want to downconvert

The same loop where the frequency correction takes place is used to update the average value of the signal, *Signal_Level* contains a moving average of the last 100000 samples.

4 The FIC handler

4.1 Introduction

As mentioned, the ofdm handler is the main driver. It is instantiated with - next to *The_Mode*, the function from the msc handler and the functions from the device handler - two functions from the fic handler, *process_ficBlock* and *Sync_Reached*.

The first few data blocks of a DAB frame are sent to the FIC handler. The package *fic_handler*, implementing the fic handler, is itself a generic one, with the DAB mode as parameter.

```
generic
   the_Mode : dabMode;
package fic_handler is
   procedure Process_Ficblock (Data   : shortArray;
                               Blkno  : Integer);
   procedure   Stop;
   procedure   Reset;
   procedure   Restart;
   procedure   Data_for_Audioservice (Name_of_Program: String;
                                      Data           : out audioData);
   function    Sync_Reached return Boolean;
end fic_handler;
```

It provides - next to the procedure and function that were passed as parameter to the ofdm handler - 4 other procedures, that are called from the GUI handling to control the behaviour.

- *Restart*, *Reset* and *Stop* are for control.
- *Data_for_AudioService* is called - from within the GUI handling - whenever the user selects a program. The data telling where to find the audio in the MSC stream will be given back.

Since data is coming in from the ofdm handler, and commands are coming in from the GUI, the functionality is implemented as a hidden task with an entry for each of the above mentioned procedures.

The procedure *Process_Ficblock* is the data interface, it is called by the ofdm handler whenever there is work to be done.

```
    procedure Process_Ficblock (Data   : shortArray;
                                Blkno  : Integer) is
    begin
```

```
      if Blkno = 2 then
         Buffer_Index   := 0;
         Fic_Number     := 0;
      end if;

      if 2 <= Blkno and then Blkno <= 4 then
         for I in 0 .. bitsperBlock - 1 loop
            Ofdm_Inputdata . ofdm_data (Buffer_Index) := Data (I);
            Buffer_Index                              := Buffer_Index + 1;
            if Buffer_Index >= 2304 then
               Fic_Processor. Handle (0, ofdm_data);
               Buffer_Index            := 0;
            end if;
         end loop;
      else
          put ("we should not be here at all with ");
          Put_Line (Integer' Image (Blkno));
      end if;
   end Process_FicBlock;
```

The procedure merely collects the data, and as soon as 2304 bits are in the data is passed on to the real processor by putting that data into a buffer. Note that the ofdm handler will never call the *process_Ficblock* function with a block other than one of block 2, 3, or 4, so the test is superfluous.

Since a DAB data block takes app 2500 samples, and the samplerate is 2048000, collecting data for a DAB block takes app 1 msec, a sufficient amount of time to allow passing the data through a rendez-vous without additional buffering.

The "other" procedures in the interface are merely abstractions for the entry calls.

```
   accept Data_for_Audioservice (Program_Name : String;
                                 Data         : out audioData) do
      fib_handler. Data_for_Audioservice (Program_Name, Data);
   end Data_for_Audioservice;
```

4.2 The task *Fic_Processor*

The main ingredient of the task *Fic_Processor* is the part that handles the incoming data. Depuncturing is done within the body of the accept statement, handling the result - i.e. the deconvolution and the prbs handling does not need to be in synchronization with the caller.

```
   accept Handle (Fic_Number : Natural; Data : Ficblock) do
      depuncture_FICblock (Data, Viterbi_Data);
   end Handle;
   Deconvolver. deconvolve (Viterbi_Data, Bitbuffer_out);
--
--  if everything worked as planned, we now have a
--  768 bit vector containing three FIB's
```

```
--  first step: prbs handling
    for I in BitBuffer_out' Range loop
       Bitbuffer_out (I) :=
                         Bitbuffer_out (I) xor Prbs_Vector (I);
    end loop;
```

Once $Bitbuffer$ is filled, it contains data for three FIB's, each with its own CRC protection.

```
    for I in 0 .. 2 loop
       -- code omitted
       declare
          My_Fib_Buffer : Fib_Buffer renames
                            Bitbuffer_out (I * 256 .. (I + 1) * 256 - 1);
       begin
--     check the slice of the bitBuffer
          if Check_ficCRC (My_Fib_Buffer) then
             fib_handler. process_FIB (My_Fib_Buffer);
          else
             Missed_Blocks := Missed_Blocks + 1;
          end if;
       end;
    end loop;
```

4.3 Depuncturing

Depuncturing is done in the procedure $depuncture\_FICblock$. The specification of the puncturing that was applied at the sender site is well described

- First we have 21 blocks with punctured according to table PI_16 each 128 bit block contains 4 subblocks of 32 bits on which the given puncturing is applied. Table PI_16 refers to a table in a list of tables, the values of which can be accessed using the function get_PCode.

```
Data_Out := (Others => 0);
for I in 1 .. 21 loop
   for K in 0 .. 3 loop
      for L in 0 .. 31 loop
         if get_PCode (16, short (L)) /= 0 then
            Data_Out (Fillpointer) := Data_In (Inputcounter);
            Inputcounter           := Inputcounter + 1;
         end if;
         Fillpointer  := Fillpointer + 1;
      end loop;
   end loop;
end loop;
```

- Second, we have 3 blocks with puncturing according to table PI_15 each 128 bit block contains 4 subblocks of 32 bits on which the given puncturing is applied (code similar as above),

- we have a final block of 24 bits with puncturing according to table PI_X. This block constitues the 6 * 4 bits of the register itself.

The tables, used for depuncturing are in the code. A table entry "0" means that on the sender's site a bit was deleted.

The soft bits as given by the ofdm handler are values in the range -127 .. 127. A '0' is therefore the representation of a 'do not know'.

Deconvolution is done by an instance of the deconvolver, which will be discussed later.

4.4 Handling PRBS

On the sender side, the data is mixed with a pseudo random binary sequence (PRBS) by XORring the data with the PRBS vector, on the receiver side, this is also to be done. Since the PRBS vector is known to have a length of 768 bits, the vector can be - and is - build on package instantiation.

Handling PRBS then is - almost - trivial

```
for I in BitBuffer_out' Range loop
   Bitbuffer_out (I) := Bitbuffer_out (i) xor Prbs_Vector (i);
end loop;
```

4.5 Checking CRC

For checking the CRC, it must be noted that the CRC value is in the last 16 bits. The CRC polynome is given by *CRC_Polynome*

```
CRC_Polynome    : constant byteArray (0 .. 14) :=
              (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0);  -- MSB .. LSB

  function Check_ficCRC (Vector : Fib_Buffer) return Boolean is
     Buffer:      byteArray (0 .. 15)     := (others => 1);
     Sum:         Integer         := 0;
     Temp:        fib_buffer      := vector;
  begin
     for I in Temp' Last - 16 + 1 .. Temp' Last loop
        Temp (I) := Temp (I) xor 1;
     end loop;

     for I in Temp' range loop
        if (buffer (0) xor temp (I)) = 1 then
           for F in 0 .. 15 - 1 loop
              Buffer (f) := CRC_Polynome (F) xor buffer (F + 1);
           end loop;
           Buffer (15) := 1;
        else
           Buffer (0 .. 14) := Buffer (1 .. 15);
           Buffer (15) := 0;
```

```
      end if;
   end loop;

   for I in Buffer' range loop
      Sum       := Sum + Integer (Buffer (i));
   end loop;
   return Sum = 0;
end Check_ficCRC;
```

4.6 Creating the PRBS

As mentioned, the PRBS vector is a constant one during the execution of the program. The vector - length 768 - is pre-computed on elaboration of the package and its computation follows the definition from the DAB standard.

```
declare
   Shift_Register: byteArray (0 .. 8) := (others => 1);
begin
   for I in 0 .. 768 - 1 loop
      Prbs_Vector (i) := Shift_Register (8) xor Shift_Register (4);
      for J in reverse 1 .. 8 loop
         Shift_Register (J) := Shift_Register (J - 1);
      end loop;
      Shift_Register (0) := Prbs_Vector (I);
   end loop;
   Buffer_Index := 0;
   Fic_Number := 0;
end;
```

4.7 Creating a FIB structure

The function of the FIB is to maintain a description of the data in the MSC, such that - when a program or service is selected - it is clear where to locate the data for that program or service in the stream and what is needed to process that data.

Since the upperlimit of the number of programs and services is known - and not very large - a choice was a rather static data structure: three vectors of length 64 containing resp.

- the $ficList$, an array containing field describing - if used the channel organization;
- the $listofServices$, an array containing typical service information and a service Label, i.e. a name and a 32 bits service identifier.
- the *components*, an array containing descriptions of - if used - the service components.

The components in these arrays contain references to each other in the form of indices.

Filling these arrays with data coming from the FIC's is - from a programming point of view - a pretty trivial (and often boring) job.

The structure of the fib handler is pretty trivial. As said before, FIBs are sehments of 256 bits. When handed over to the fib handler, the CRC is already checked, so the FIB handler starts unpacking the FIGs.

```
procedure process_FIB (p  : fib_buffer) is
   FIGtype        : uint16_t;
   processedBytes : short_Integer         := 0;
   bitOffset      : short_Integer         := 0;
begin
   while processedBytes < 30 loop
      FIGtype   := get_Bits (p, bitOffset, 3);
      case FIGtype is
         when 0   => process_FIG0 (p, bitOffset);
         when ... -- code omitted
      end case;
      processedBytes   := processedBytes +
                             short_Integer (get_Bits (p, bitOffset + 3, 5)) + 1;
      bitOffset        := processedBytes * 8;
   end loop;
end process_FIB;
```

The FIB is built up from a row of FIGs. The *type* of the FIG is recorded in the first few bits. To access bits, a function *get_Bits* is available with three parameters

- the array where the FIB is stored,
- the offset of the first bit,
- and the amount of bits to extract.

The FIG further contains an indication of the length (in bytes), making it easy to dispatch the subsequent FIGs in a single loop.

The FIGs themselves are grouped in types, so next to the type there are subtypes, expressed in *extensions*. The structure chosen is therefore to have the *process_FIGxx* functions dispatch the extension and let a dedicated function handle the extension.

```
procedure process_FIG0 (p      : fib_buffer;
                        offset : short_Integer) is
   extension : short_Integer :=
            short_Integer (get_Bits (p, offset + 8 + 3, 5));
begin
   case extension is
      when 0 => FIG0Extension0 (p, offset);
      when 1 => FIG0Extension1 (p, offset);
      when 2 => FIG0Extension2 (p, offset);
      .... -- code omitted
--     when 22 => FIG0Extension22 (p, offset);
      when others =>
                null;
```

```
      end case;
   end process_FIG0;
```

A single FIG of a given type may contain a sequence of extensions of the same type. The structure chosen to handle extensions is therefore as given below

```
   procedure FIG0Extension1 (d      : fib_buffer;
                             offset : short_Integer) is
....  -- declarations omitted
   begin
      while bitOffset / 8 < Length - 1 loop
         bitOffset := bitOffset / 8;
         bitOffset := HandleFIG0Extension1 (d, 8 * bitOffset, PD_bit);
      end loop;
   end FIG0Extension1;
```

Not all FIGs are needed, in this implementation we merely look for those FIGs giving information on the programs and the associates services and channels.

As said above, extracting data from the various FIGs is basically boring and - from a programmers point of view - a simple operation, not to be discussed further.

5 The MSC handler

5.1 Introduction

The MSC handler is the "server" for handling the incoming MSC datastream. As mentioned before, the ofdm handler passes on a vector with (soft)bits for each block in the MSC part of the DABframe, the msc handler then does the (control of the) processing of these data blocks.

The incoming data is stored in a so-called CIF vector, a vector of 55296 bits. The start address and length of segments to be processed when a service is relative to the start of this CIF vector.

Since the DAB mode is known and does not change during the execution of the DAB program, the MSC handler needs to be instantiated only once and then "knows" all relevant parameters related to the mode, such as the number of (soft)bits in the incoming data bloks and the amount of blocks per CIF.

To keep things simple, the msc handler merely collects the data, fills the CIF vector, sets parameter values if asked to do so, it selects the data and delegates the further processing to the *Dab_Handler*, a separate module.

To allow concurrent processing of the ofdm handler and the msc handler - and its associates - the msc processor is implemented as a (hidden) task, and can be executed on another CPU core (if available).

As with the FIC handler, there is an issue that two separate "processors" may (want to) access the msc handler. The GUI may want to instruct the msc handler to set or

change parameters when a different program is selected, while the ofdm handler just continues to push its collected MSC data into the MSC handler.

As with the FIC handler, we choose here for a task implemention of the *msc_processor*, with entries as given below

```
task msc_processor is
   entry reset;
   entry stop;
   entry process_mscBlock (Element : buffer_Element);
   entry set_audioData    (Data    : audioData);
end msc_processor;
```

To handle the potential blocking of the ofdm handler whenever the *msc_processor* (or one of its delegates running in the task) is busy, an active buffer is placed between the ofdm handler and the *msc_processor* task[7]

Requests from the $Ofdm\_Decoder$ to process a vector with data are put into a buffer

```
package mscBuffer is new Generic_Buffer (buffer_element);
the_mscBuffer : mscBuffer. Buffer (L (The_Mode));
```

This buffer is embedded in a *helper* task, implemented such that the ofdm handler can deliver its data without being blocked.

```
task body helper is
   Element : buffer_Element;
begin
   loop
      the_mscBuffer. Get (Element);
      msc_processor. process_mscBlock (Element);
   end loop;
end;
```

5.2 The *msc_processor* task

The implementation of the task body *msc_processor* then is easy

- As long as there is no "command" to do something, processing incoming data blocks is trivial: they are ignored. If, however, there is "work to be done", indicated by $Work\_to\_be\_done$ then the blocks are stored in the CIF vector; as soon a CIF vector is completely filled, a function in the *dab_handler* is called with the slice of the CIF vector containing the data for the requested service.

```
accept Process_mscBlock (Element : buffer_Element) do
   if Work_To_Be_done then
      Current_Block := (Element. blkno - 5) mod Blocks_per_CIF;
      Cif_Vector (Current_Block * Bits_per_Block ..
                  (Current_Block  + 1) * Bits_per_Block - 1) :=
                                 Element. Data;
```

[7]Note that we have chosen to execute all audio handling within the thread of this task. To prevent blocking of the $Ofdm\_Decoder$ this intermediate buffer is created.

```
   else
      return;
   end if;
end process_mscBlock;
if Current_Block >= Blocks_per_CIF - 1 then  --  a full CIF
   Cif_Count := (Cif_Count + 1) mod 4;
   The_DabProcessor.
          Process (Cif_Vector (Integer (Start_Address) * CUSize ..
                       Integer (Start_Address + Length) * CUSize - 1));
end if;
```

- When the GUI calls for setting parameters for a selected a program (service), the data describing the relevant parameters for that service are set as output parameter.

  The *msc_processor* will set the appropriate parameters and create a (new) instance of the *Dab_Processor* in the package *Dab_Handler* with sufficient parameters to enable the *DAB_Processor* to do the decoding.

```
accept Set_AudioData (Data: audioData) do
   The_Data         := Data;
end Set_AudioData;
Work_To_Be_Done     := true;
Start_Address       := The_Data. startAddr;
Length              := The_Data. Length;
if The_Data. ASCTy = 8#077# then
   Dabmodus         := DAB_PLUS;
else
   Dabmodus         := DAB;
end if;

if The_Dabprocessor /= null then
   Free_dabProcessor (the_dabProcessor);
end if;
The_Dabprocessor       := new Dab_Handler. Dabprocessor
                                  (Dabmodus,
                                   Integer (The_Data. Length) * CUSize,
                                   The_Data. bitRate,
                                   The_Data. uepFlag,
                                   The_Data. protLevel,
                                   Audio_Handler);
```

  This approach allows choosing between different *DabProcessor* instances, depending whether an audio or data service is selected (in the Ada implementation we limit ourselves for the time being to audio services).

  In this case, parameters are, apart from the Dabmodus (not to be confused with DabMode) that tells whether we are dealing with DAB or DAB+, the address of the data in the CIF vector and the parameters needed for handling the protection and deconvolution. The handler for the PCM samples, i.e. the sound output, which

was initialized during the elaboration of the package body of the *msc_handler*, is passed on as parameter as well.

- A "Reset" is simply a reset of some of the parameters, and
- a "stop" - only called on at program termination - will terminate the *dabProcessor*, free the associated resources and raise an exception to terminate the task.

```
accept stop;
   if The_Dabprocessor /= null then
      Free_dabProcessor (the_dabProcessor);
   end if;
   raise endMSC;
end;
```

5.3 The dab handler

The dab handler controls the deconvolution of the selected data in the CIF vector. It applies the de-interleaving to the data, after which it controls depuncturing and deconvolution and it sends the result to either the MP2 handler - in case a classic "DAB" program is selected - or the MP4 handler - in case a "DAB+" program is selected.

The *DabProcessor* is implemented as a controlled type.

In the initialization for instances of that type one of the MP2 or MP4 handler is instantiated as well as the "processor" that handles the deconvolution.

- the variable *The_AudioProcessor* is set to either the MP2 or MP4 processor;
- the variable *The_ProtectionProcessor* is set to either the *uepProcessor* or the *eepProcessor*, depending on the parameter passed.

```
if uepFlag = 0 then
   The_ProtectionProcessor :=
                      new uepProcessor (bitRate, protLevel);
else
   The_ProtectionProcessor :=
                      new eepProcessor (bitRate, protLevel);
end if;

if dabModus = DAB then
   The_AudioProcessor :=
                    new mp2_handler. mp2Processor (bitRate, audio);
else
   The_AudioProcessor :=
                    new mp4_handler. mp4Processor (bitRate, audio);
end if;
```

Processing the data is then straightforward. The procedure *Process* is called with an array with soft bits, and interleaving is done in-line, the *The_ProtectionProcessor* is called to do the depuncturing and the deconvolution and convert the softbits into hard bits, after which the in-line energy dispersal takes place.

5.3.1 The in-line de-interleaving

De-interleaving is done in-line. We know the length of the fragments, and we know the de-interleaving table, so it is easy to create a two dimensional array for performing the de-interleaving.

```
interleaveData     : shortBlock (0 .. 15, 0 .. fragmentSize);
countforInterleaver: int16_t;
interleaverIndex   : int16_t;
```

The maximum depth of the interleaving is 15. The $interleaverIndex$ moves in the range 0 .. 15 and indicates the row in the $interleaverData$ array where the new incoming data is to be put.

We maintain a counter that indicates whether or not the interleaving buffer is sufficiently filled, so we can output the de-interleaved data. The delays used in the interleaving (de-interleaving) are maintained in a constant array $interleaveDelays$.

De-interleaving then is essentially straightforward, put the new data in a row in the two dimenstional structure $Interleavedata$, and take the right element from the columns. Virtual "shifting" the data in the rows of the structure is by $interleaverIndex$, the index to the row of the newest data.

```
for I in Integer range 0 .. Object. fragmentSize - 1 loop
   declare
      Index          : Integer := Integer (I mod 16);
      currentRow     : int16_t :=
             int16_t ((Object. interleaverIndex +
                                interleaveMap (Index)) mod 16);
   begin
      tempX (I) := Object. Interleavedata (currentRow, I);
      Object. interleaveData (Object. InterleaverIndex, I) :=
                                             data (data' First + I);
   end;
end loop;
```

The output of the de-interleaver is handed over for depuncturing and deconvolution, functionality implemented in $The\_Protectionprocessor$.

```
--just wait until the interleaver is "filled"
if countforInterleaver < 15 then
   countforInterleaver := countforInterleaver + 1;
else
   The_ProtectionProcessor. deconvolve (tempX, outV);
```

The output of the deconvolution is in the array $outV$, after applying the energy dispersal it is (better: should be) the basis to create a nice sound.

The energy dispersal is implemented as a simple "shift and xor" operation, done in-line for performance reasons.

```
    declare
       shiftRegister: byteArray (0 .. 8) :=  (others => 1);
    begin
       for I in outV' Range loop
          declare
             B: uint8_t := shiftRegister (8) xor shiftRegister (4);
          begin
             for J in reverse 1 .. 8 loop
                shiftRegister (J) := shiftRegister (J - 1);
             end loop;
             shiftRegister (0)   := b;
             outV (I)            := outV (I) xor b;
          end;
       end loop;
    end;
```

The result, the elements in the vector *outV*, is input to the audio decoder, and since we already selected a "processor", we merely pass on the data

```
The_AudioProcessor. Add_to_Frame (outV, 24 * bitRate);
```

5.4 Handling DAB: The MP2 processor

Although the classical version of DAB, using MP2 as coding mechanism, almost completely disappeared, a DAB software system is not complete without support for MP2.

For handling MP2 we completely rely on an external library, the KJMP library from Martin Fiedler[8], our software translates the frames as delivered by the dab handler to frames, acceptable to the KJMP2 library.

The MP2 and MP4 handler are both derived from an *Audio_Processor*, in the package *audio_handler*, and reimplementing the common interface function *Add_to_Frame*.

The *Audio_Processor* is instantiated each time a program is selected in the GUI, the controlled type provides the mechanism to instantiate and delete objects of the type.

```
package audio_handler is
    type Audio_Processor (bitRate:    short_Integer;
                          pcmHandler: audiopackage. audioSink_P) is
                    new Ada. Finalization. Controlled with
       record
          null;
       end record;
    type Audio_Processor_P is access all Audio_Processor' Class;

    procedure Add_to_Frame (Object  : in out Audio_Processor;
                            Data    : byteArray;
                            Nbits   : short_Integer);
end audio_handler;
```

The interface to the KJMP2 library contains a few C functions that are renamed into Ada ones.

- *kjmp2_init*, to be called each time once to initialize each kjmp2 decoder instance.

```
procedure kjmp2_init    (mp2: System. Address);
```

- *kjmp2_get_sample_rate*, returns the sample rate of a MP2 stream. The parameter frame points to at least the first three bytes of a frame from the stream. The return value is the sample rate of the stream in Hz, (or zero if the stream isn't valid).

```
function kjmp2_get_sample_rate (frame: System. Address) return Integer;
```

- *kjmp2_decode_frame*, to decode one frame of audio. The parameter mp2 is s pointer to a context record that has been initialized with *kjmp2_init*. The parameter frame is a pointer to the frame to decode. It *must* be a complete frame, because no error checking is done! and the parameter pcm is a pointer to the output PCM data. Note that the return value will always be 1152.

```
function kjmp2_decode_frame (mp2:     System. Address;
                             frame:   System. Address;
                             pcm:     System. Address) return Integer;
```

The processing is quite straightforward, if we are "in sync", we just keep adding bits (using a local function *Add_bit_to_MP2*) until the buffer is filled, and we call the *kjmp2_decode_frame* function to do the decoding. The output is converted to floats and the data is handed over to the audio handler.

```
if Object. MP2Header_OK = 2 then
   Add_bit_to_MP2 (Object. MP2frame. all,
                   Data (i),
                   Object. MP2bitCount);
   Object. MP2bitCount   := Object. MP2bitCount + 1;
   if Object. MP2bitCount >= lf then
      count := kjmp2_decode_frame (Object. context' Address,
                                   Object. MP2frame. all' Address,
                                   outBuffer' Address);

      if count <= 0 then
         return; -- something wrong
      end if;
      for i in 0 .. KJMP2_SAMPLES_PER_FRAME - 1 loop
         sample_Buf (i) := (Float (outBuffer (2 * i    )) / 32768.0,
                            Float (outBuffer (2 * i + 1)) / 32768.0);
      end loop;
      Object. pcmHandler. putSamples (sample_Buf,
                                      uint64_t (Object. baudRate));
      Object. MP2Header_OK     := 0;
      Object. MP2HeaderCount   := 0;
      Object. MP2bitCount      := 0;
   end if;
```

If, on the other hand, we are not (yet) in sync, we try to locate the start of a frame in two steps. First by looking at the input until we meet a sequence of 12 successive "1" bits.

```
elsif Object. MP2Header_OK = 0 then -- no sync yet
   if Data (i) = 01 then -- all bits should be a "1"
      Object. MP2HeaderCount  := Object. MP2HeaderCount + 1;
      if Object. MP2HeaderCount = 12 then -- we have 12 '1' bits in a row
         Object. MP2bitCount := 0;
         for j in 0 .. 12 - 1 loop
            Add_Bit_to_MP2 (Object. MP2frame. all,
                                    1, Object. MP2bitCount);
            Object. MP2bitCount   := Object. MP2bitCount + 1;
         end loop;
         Object. MP2Header_OK  := 1; -- next state
      end if;
   else
       Object. MP2HeaderCount   := 0;
   end if;
```

If we have the 12 "1" bits in a row, we read bits to extract the baudrate.

```
elsif Object. MP2Header_OK = 1 then
   Add_Bit_to_MP2 (Object. MP2frame. all,
                   Data (i),
                   Object. MP2bitCount);
   Object. MP2bitCount := Object. MP2bitCount + 1;
   if Object. MP2bitCount = 24 then   -- relevant part header
      Object. baudRate :=
            kjmp2_get_sample_rate (Object. MP2frame. all' Address);

      if Object. baudRate /= 48000
                  and then Object. baudRate /= 24000 then
         Object. MP2Header_OK     := 0;
         Object. MP2HeaderCount   := 0;
         Object. MP2bitCount      := 0;
         return; -- failure
      end if;
      Object. MP2Header_OK  := 2;
   end if;
end if;
```

If valid we are just in sync, and we can process a frame.

5.5 Handling DAB+: The MP4 processor

5.5.1 Overview

The structure of the data in DAB+ is slightly more complex than the structure of plain DAB data. The data is carried in so-called *superframe*'s, where a superframe consists of 5 data segments in subsequent DAB frames.

The first part of such a superframe can be recognized by showing a distinct pattern in the first few bytes, the $firecode$.

Recognizing superframes is therefore by shifting in 5 subsequent data segments into a potential superframe, then verifying that the first segment matches the firecode. If not, then the segments in the potential superframe are shifted, and a new one is added.

If a match is found, we might have a superframe and start processing it.

First step in processing a superframe is applying a Reed-Solomon algorithm to correct up to 5 errors in the superframe. If the errors could not be corrected, we just give up, if the errors (if any) could be corrected we start interpreting the data and extracting the Audio Units.

5.5.2 Adding the data

The input to the function *Add_to_Frame* is an array with the data in bits, still one bit per byte.

```
    procedure Add_to_Frame (Object   : in out mp4Processor;
                            Data     : byteArray;
                            Nbits    : int16_t);
```

So, the first step is to collect them into bytes

```
    for i in 0 .. Nbytes - 1 loop
       temp    := 0;
       for j in 0 .. 8 - 1 loop
          temp := Shift_Left (temp, 1) or (Data (8 * i + j) and 8#01#);
       end loop;
       Object. RSin_Data (Block_FillIndex * nbytes + i) := temp;
    end loop;
--
    Object. Blocks_InBuffer := Object. Blocks_InBuffer + 1;
    Object. Block_FillIndex := (Object. Block_FillIndex + 1) mod 5;
```

The bits are packed in bytes - it is known that the amount of bits can be divided by 8 - and the resulting block is added to the (potential) superframe.

```
    fcVector := Object. RSin_Data (Block_FillIndex * nbytes ..
                         Block_FillIndex * nbytes + fcVector' Length - 1);
    if not firecode_Checker. check (fcVector) then
--  we were wrong, a virtual shift to left in block sizes
       Object. Blocks_InBuffer  := 4;
       Object. FrameErrors      := Object. FrameErrors + 1;
       return;
    end if;
```

If the firecode is OK, which is reported from the call to $firecode\_Checker.check$, we can try to process the superframe. *Failure* to process the superframe, indicated by a $false$ value of the result, is believed to show an error in detecting the segments forming the superframe, in that case just delete one segment, we shift in a next one and retry to detect a superframe.

```
processSuperFrame (Object,
                   Object. RSin_Data. all,
                   short_Integer (Block_FillIndex * nbytes),
                   result);

if not result then   -- we will try it again, next time
   Object. Blocks_InBuffer   := 4;
   Object. Frameerrors       := Object. Frameerrors + 1;
   return;
end if;
```

5.5.3 Processing the superframe

Processing a superframe involves two actions:

- applying a Reed-Solomon algorithm to correct errors - if any -, and
- extracting the Audio Units (and passing on the data).

Applying Reed-Solomon The superframe consists of $RSDims * 120$ unsigned bytes, where $RSDims$ is computed from the $bitRate$, which is passed as parameter at the instatiation of an object of the type.

```
Object. Superframe_size   := Integer (110 * (bitRate / 8));
Object. RSDims            := Integer (bitRate / 8);
Object. RSin_Data         := new byteArray (0 .. RSDims * 120 - 1);
Object. RSout_Data        := new byteArray (0 .. RSDims * 110 - 1);
```

For applying Reed-Solomon decoding we have to put these unsigned bytes in a two dimensional array, columnwise, apply Reed-Solomon decoding on the rows, and extract the results, again columnwise.

Rather we extract the rows as *they should appear in the matrix*, apply Reed-Solomon decoding and putting them back.

```
for J in 0 .. RSDims - 1 loop
   for K in rsIn' Range loop
      rsIn (K) := frameBytes ((Integer (base) + J +
                               K * RSDims) mod (RSDims * 120));
   end loop;

   the_rsDecoder. decode_rs (rsIn, 135, rsOut, Errors_in_RS);
   ...   -- code omitted
   for K in rsOut' Range loop
      Object. RSout_Data (J + (K - rsOut' First) * RSDims) :=
                                                  rsOut (K);
   end loop;
end loop;
```

The Reed-Solomon decoding takes a vector of 120 bytes. The decoding is based on a 255,10 scheme, so 135 zeros are added, which explains the number 135 that appears in the parameterlist.

Note that the length of the length of the vectors that enter the Reed-Solomon decoding is 120 bytes, while, the vectors returned ar 110 bytes, the 10 bytes difference are the parity bytes with which at most 5 erroneous bytes can be repaired.

The last parameter in the call to *the_rsDecoder.decode_rs* is an output parameter, telling the number of corrected errors or, in case error correction turned out to be impossible, a negative number.

In case the errors could not be corrected, processing of this superframe is given up. If all errors are corrected, it is time to extract the Audio Units.

Extracting the Audio Units The number of Audio Units (AU) in a superframe (up to 7), and their locations in the superframe, are encoded in the first few bytes of the superframe. Since there should not be any errors left, we can interpret this data, find out the number of units, and find the offsets of these units in the incoming data.

The header of a superframe contains a number of bits, to be extracted and used to decode the number of AU's and their offsets in the superframe.

```
case 2 * dacRate + sbrFlag is
   when 0    =>
      num_aus        := 4;
      au_start (0)   := 8;
      au_start (1)   := uint16_t (RSout_Data (3)) * 16 +
                        uint16_t (Shift_Right (RSout_Data (4), 4));
      au_start (2)   := uint16_t (
                          (RSout_Data (4)) and 16#0f#) * 256 +
                         uint16_t (RSout_Data (5));
      au_start (3)   := uint16_t (RSout_Data (6)) * 16 +
                        uint16_t (Shift_Right (RSout_Data (7),  4));
      au_start (4)   := uint16_t (110 *  (bitRate / 8));
   when 1 =>
      ....  -- code omitted
end case
```

The *num_aus* variable tells the number of AU's, the *au_start* vector contains the offsets of the start of the segments in the superframe.

The subsequent AU's are extracted, a CRC check performed and the data is handed over to an AAC decoder.

```
for i in 0 ..  num_aus - 1 loop
   declare
      aac_frame_length  : uint16_t;
      theAU             : ByteArray (0 .. 1920) := (others => 0);
   begin
      Object. au_count   := Object. au_count + 1;
      aac_frame_length   := au_start (i + 1) - au_start (i) - 2;

      if dabPlus_crc (RSout_Data, au_start (i),
                                  aac_frame_length) then
         theAU (0 .. Integer (aac_frame_length - 1)) :=
                       RSout_Data (Integer (au_start (i)) ..
```

```
                        Integer (au_start (i)) +
                             Integer (aac_frame_length) - 1);

             if (Shift_Right (theAU (0), 5) and 07) = 4 then
                pad_handler. Process_PAD (theAU);
             end if;

--we add a few zero bytes to allow look ahead of the aac decoder
             for J in aac_frame_length .. aac_frame_length + 10 loop
                theAU (Integer (J)) := 0;
             end loop;

             faad_decoder. mp42pcm (short_Integer (dacRate),
                                    short_Integer (sbrFlag),
                                    short_Integer (mpegSurround),
                                    short_Integer (aacChannelMode),
                                    theAU,
                                    aac_frame_length,
                                    Samples_Out,
                                    Object. pcmHandler);
          else
             Object. au_errors := Object. au_errors + 1;
          end if;
       end;
    end loop;
```

Note that, since the errors are corrected in the superframe, it is not expected for the CRC check to fail. However, we are a little defensive and add some "sanity checks".

Note further that the AU may contain data to be extracted for pad handling

```
if (Shift_Right (theAU (0), 5) and 07) = 4 then
   pad_handler. Process_PAD (theAU);
end if;
```

The PAD handling is not further discussed in this report.
Note finally that the size of the array to contain the AU is far beyond the expected length of 960.

5.5.4 Handling the AAC

The AAC decoder is implemented in an external C library, the faad[9] library, which is interfaced to with a class $faad\_decoder$.

The approach taken is to have just a "clever" procedure $mp42pcm$ that gets the data, a procedure that "knows" whether or not to initialize the real faad library. The aac coding of DAB is slightly special, it is a 960 coding rather than the more common 1024 coding.

```
asc (0)  := Interfaces. C. unsigned_char (
                           (Shift_left (2#00010#, 3) or
```

```
                              Shift_Right (core_sr_index, 1)) and 16#FF#);
    asc (1)  := Interfaces. C. unsigned_char (
                             (Shift_Left (core_sr_index and 01, 7) or
                        Shift_Left (uint16_t (core_ch_config), 3) or 2#100#) and 16#FF#);
--
--      Note that NeAACDecInit2 has two "out parameters", we just
--      pass their addresses to the C function
            init_result    := NeAACDecInit2 (aacHandle,
                                             asc,
                                             2,
                                             sample_rate,
                                             channels);
```

Once the library is initialized, we can delegate the decoding

```
outBuffer    :=  NeAACDecDecode (aacHandle,
                                 hInfo' Address,
                                 buffer' Address,
                                 Interfaces. C. long (bufferLength));
```

The call to the function return a C array, beforehand we do not know the constraints, so, to make it into an Ada variable we apply a conversion. Note that the amount of samples is part of the values returned in the $hInfo$ structure.

```
declare
   type C_shortArray is array (integer range <>) of
                                             Interfaces. C. short;
   subtype localOutput is C_shortArray (0 .. samples - 1);
   package arrayConverter is
            new System. Address_To_Access_Conversions (localOutput);
   theBuffer : arrayConverter. Object_Pointer :=
                            arrayConverter. To_Pointer (outBuffer);
```

The package $Address\_To\_Access\_Conversions$ provide the means to convert a pointer to a C array to a decent Ada access value to an array with a given subtype. Instantiating the package with the subtype provides a decent access type to an array, initialized to the C array.

Obviously, once we have a decent access to the buffer values, we can put them into buffer to be handled by the audio handler.

5.6 Depuncturing

The so-called *uep_processor* and *eep_processor* handle the depuncturing and they delegate the deconvolution of the selected elements in the MSC data stream to a deconvolution handler.

The interface to the two "processors" is the same, and derived from an (almost empty) parent.

```
package body protection_handler is
   type protectionProcessor (bitRate    : short_Integer;
                             protLevel  : short_Integer)
 is new Ada. Finalization. Controlled with record null; end record;
   type protectionProcessor_P is access all protectionProcessor'Class;
   procedure deconvolve (Object    : in out protectionProcessor;
                         inBuffer  : shortArray;
                         outBuffer : out byteArray) is
   begin
      null;
   end deconvolve;
end protection_handler;
```

The basic idea is to give as input parameter an array with the "soft bits", and get back an array with the "hard bits".

Both the *eepProcessor* and the *uepProcessor* are instantiated when needed and deleted when not needed anymore, so they are implemented as derived from a controlled type.

5.6.1 The eep processor

On instantiation of the *eepProcessor*, two parameters are given, the *bitRate* and the *protectionLevel*. The protection level determines which tables are to be used in the depuncturing. The tables are all listed in a constant array P_code_array, made available in the package *prottables*, and their elements are accessible through get_PCode.

The bitrate determines the size of the deconvolved output. One input segment should result in 24 msec of output, the number of output bytes from the deconvolution therefore is $24 \times bitRate/8$.

```
case Object. protLevel mod 8 is
   when 1   =>
      Object. L1          := 6 * Object. bitRate / 8 - 3;
      Object. L2          := 3;
      Object. PI1_Index   := 24;
      Object. PI2_Index   := 23;
   when 2  =>
      ....   -- code omitted
```

The variables $L1$ and $L2$ contain the length of the segments to be depunctured. The depuncturing tables themselves are the tables with index $PI1\_Index$ resp. $PI2\_Index$.

The tables itself are 32 values long, so the depuncturing will be repeated after each 32 values. Note that - due to the choice made in the ofdm handler with decoding values - that 0 is the "no idea" value.

```
for i in 0 .. Object. L1 - 1 loop
   for j in short_Integer Range 0 .. 128 - 1 loop
      if get_PCode (Object. PI1_Index, j mod 32) = 1 then
         Object. viterbiBlock (viterbiCounter) :=
                                   inBuffer (first + inputCounter);
```

```
            inputCounter     := inputCounter + 1;
         end if;
         viterbiCounter := viterbiCounter + 1;
      end loop;
   end loop;
```

It is known that the final block is 24 bits, constituing the 6 * 4 bits of the deconvolution register. This way the vector *viterbiBlock* is filled and, when complete, handed over to the deconvolver to do the translation to hard bits.

```
-- we have a final block of 24 bits  with puncturing according to PI_X
-- This block constitues the 6 * 4 bits of the register itself.
   for i in 0 .. 24 - 1  loop
      if PI_X (i) = 1 then
         Object. viterbiBlock (viterbiCounter) :=
                                        inBuffer (first + inputCounter);
         inputCounter    := inputCounter + 1;
      end if;
      viterbiCounter     := viterbiCounter + 1;
   end loop;
   Object. viterbi. deconvolve (Object. viterbiBlock. all, outBuffer);
```

The resulting "hard" bits are - one bit per byte - stored in $outBuffer$.

5.6.2 The uep processor

The structure of the uep processor is - quite obvious - more or less similar to that of the eep processor.

There are now, however, 4 tables to deal with. The right protection profile (i.e. the lengths of the segments and which depuncturing table to apply , is determined by the bitrate and the protection level, values that are set when instantiating the uep processor. These values lead to an index in the *protectionTable*, the values of which are used.

```
   function findIndex (bitRate     : short_Integer;
                       protLevel   : short_Integer)
                                            return short_Integer is
   begin
      for i in profileTable' Range loop
         if profileTable (i). bitRate = bitRate and then
                          profileTable (i). protLevel = protLevel then
            return i;
         end if;
      end loop;
      return -1;
   end findIndex;
```

The index in the $profileTable$ then is found by

```
Index        := findIndex (Object. bitRate, Object. protLevel);
```

and the tables used in the depuncturing

```
Object. L1   := profileTable (index). L1;
Object. L2   := profileTable (index). L2;
Object. L3   := profileTable (index). L3;
Object. L4   := profileTable (index). L4;

Object. PI1_Index    := profileTable (index). PI1;
Object. PI2_Index    := profileTable (index). PI2;
Object. PI3_Index    := profileTable (index). PI3;
if profileTable (index). PI4 /= 0 then
   Object. PI4_Index := profileTable (index). PI4;
else
   Object. PI4_Index := -1;
```

The depuncturing itself follows the same scheme as we have seen with the eep processor.

```
Object. viterbiBlock. all    := (Others => 0);
for I in 0 .. Object. L1 - 1 loop
   for J in short_Integer Range 0 .. 128 - 1 loop
      if get_PCode (Object. PI1_Index, J mod 32) = 1 then
         Object. viterbiBlock (viterbiCounter) :=
                                        inBuffer (first + inputCounter);
         inputCounter := inputCounter + 1;
      end if;
      viterbiCounter := viterbiCounter + 1;
   end loop;
end loop;
```

and, again as with the eep processor, the actual deconvolution is delegated to

```
Object. viterbi. deconvolve (Object. viterbiBlock. all, outBuffer);
```

6 Handling the output

6.1 The Interface

The KJMP2 library in case of DAB, and the faad[9] libary in case of DAB+, generate PCM samples which can be made audible. For controlling the soundcard we use the portaudio library[3], which provides a portability layer between the soundcard and the program.

The interface to the AAC hander and the MP2 handler is simple, a simple a *putSamples* function is available.

```
   pcmHandler. putSamples (outBuf, uint64_t (sample_rate));
```

The functions in the audio handler interface read

```
procedure portAudio_start       (Object:  in out audioSink;
                                 result:  out boolean);
procedure portAudio_stop        (Object:  in out audioSink);
```

```
procedure putSamples            (Object:  in out audioSink;
                                          data:    header. complexArray;
                                          sampleRate: header. uint64_t);
procedure selectDefaultDevice   (Object:  in out audioSink;
                                          res:     out boolean);
procedure selectDevice          (Object:  in out audioSink;
                                          res:     out boolean;
                                          Device_Index: PaDeviceIndex);
```

The rate of the output of the Ada-DAB program is 48000, in some cases the generated PCM samples have a different sample rate. The actual rate of the data is therefore part of the parameters of *putSamples*, and in some cases rate conversion is required.

Since in all cases - in the current implementation - the default device is selected, the procedure *selectDevice* is not used outside the audio handler.

The audio handler package provides a controlled type *audiosink*, on its initialization it will call

```
Error        := Pa_Initialize;
if Error /= paNoError then
   Object. Has_Error := true;
   return;
end if;

Object. Numof_Devices        := Pa_GetDeviceCount;
Object. Is_Initialized       := true;
Object. Is_Running           := false;
```

The portaudio library will be initialized, and it is known how many devices there are (in- and output devices).

The initialization of the audio handling - in this implementation - is in the msc handler and consists of selecting a device and starting the device.

```
Audio_Handler. selectDefaultDevice (res);
if res then
   put_line ("setting default device succeeded");
end if;
Audio_Handler. portAudio_start (res);
```

Note that - as mentioned earlier - we keep it simple here and support only the default output device.

6.2 The callback function

The portaudio library uses a callback function to acquire data from the user to be sent to the selected soundcard.

The portaudio library is in 'C', the callback function in Ada. Note that the same type of callback function is used for input and output. Since input from the soundcard is not handled here, there are some parameters unused.

While the actual processing is simple, just copying data from one buffer to another,

```
    pa_ringBuffer.
       getDataFromBuffer (My_Environment. buffer,
                          My_Buffer. all, Amount);
```

we have to do some conversions to get the Ada and C world in harmony.

```
    function paCallback (Input       : System. Address;
                         Output      : System. Address;
                         Frame_Count : Interfaces. C. unsigned_long;
                         TimeInfo    : access PaStreamCallbackTimeInfo;
                         StatusFlags : PaStreamCallbackFlags;
                         UserData    : System. Address)
                                       return PaStreamCallbackResult is
       subtype localBufferType is
              pa_ringBuffer. buffer_data (0 .. integer (Frame_Count) - 1);
       package environmentConverter is
              new System. Address_To_Access_Conversions (audiosink);
       package arrayConverter is
              new System. Address_To_Access_Conversions (localBufferType);
--
--     my_environment is the record of type audiosink, passed on through the
--     Pa_OpenStream function
       My_Environment: environmentConverter. Object_Pointer :=
                               environmentConverter. To_Pointer (userData);
--
--     mybuffer is actually a C array, provided for by the underlying
--     portaudio library, here named as My_Buffer
       My_Buffer : arrayConverter. Object_Pointer :=
                         arrayConverter. To_Pointer (output);
       Amount    : Integer;
    begin
       if My_Environment. Callback_Returnvalue /= paContinue then
          return My_Environment. Callback_Returnvalue;
       end if;

       pa_ringBuffer.
             getDataFromBuffer (My_Environment. buffer,
                                My_Buffer. all, Amount);
       if Amount <  Integer (Frame_Count) then
          My_Buffer. all (Amount .. My_Buffer. all' Last) :=
                                      (Others =>  (0.0, 0.0));
       end if;
       return My_Environment. Callback_Returnvalue;
    end paCallBack;
```

First of all, the callback function has a C array as parameter for storing the output (i.e. the PCM samples). The parameter *Frame_Count* indicates the number of frames that can be handled, so we can create a subtype to be applied in an instantiation of the package *Address_To_Access_Conversions*.

```
       subtype localBufferType is
                pa_ringBuffer. buffer_data (0 .. integer (Frame_Count) - 1);
```

```
package arrayConverter is
 new System. Address_To_Access_Conversions (localBufferType);
```

With this instance, we can create an unchecked access to the parameter "output" as though it were real Ada.

```
My_Buffer : arrayConverter. Object_Pointer :=
                          arrayConverter. To_Pointer (output);
```

Then there is the issue of scope and visibility. The callback function is called from a different environment, and does not "see" any of the variables declared in the package. This means that the buffer where the data is coming from is not directly visible. It is made visible though through a "context" parameter, passed on registering the callback function, and inside the callback it is known as $My\_Environment$.

The soundcard data is fetched from the buffer with $getDataFromBuffer$ and put into $My\_Buffer.all$. It might happen that at some point in time the portaudio library processes the data faster than it is delivered, in which case the buffer is filled up with zero values.

6.3 Handling rate conversions

The external interface to the audio handler is the *putSamples* procedure. The procedure is merely a dispatcher on the samplerate.

```
procedure putSamples    (Object     : in out audiosink;
                         data       : complexArray;
                         sampleRate : uint64_t) is
begin
   case sampleRate is
      when 16000        => putSamples_16 (Object, data);
      when 24000        => putSamples_24 (Object, data);
      when 32000        => putSamples_32 (Object, data);
      when 48000        => putSamples_48 (Object, data);
      when Others       => null;                --just ignore the stuff
   end case;
end putSamples;
```

The samplerates that are possible are 16000, 24000, 32000 and 48000. For 48000, there is no need for conversion. While 16000 and 24000 are easy to handle: just add some zero values in the data and apply filtering, 32000 is slightly more complicated

For the latter conversion we apply two steps:

- in the first step we up-convert 32000 to 96000, and
- in the second step we decimate to 48000.

The filters used are simple lowpass FIR filters of degree 5.

```
f_16    : fir_filters. lowPass_filter (5, 16000, 48000);
f_24    : fir_filters. lowPass_filter (5, 24000, 48000);
f_32    : fir_filters. lowPass_filter (5, 32000, 96000);
```

Upconversion implies adding zeros and filtering, for the subsequent down conversion we know that the bandwidth of the signal is still smaller than 32000 (the upconversion does not add bandwidth to the signal), so for downconversion we can keep it simple and just take each second sample.

```
procedure putSamples_32     (Object : in out audiosink;
                             data   : complexArray) is
   buffer_1 : complexArray (0 .. 3 * data' Length - 1);
   buffer_2 : buffer_data (0 .. buffer_1' Length / 2 - 1);
begin
   for i in Data' Range loop
      declare
         index : Integer := Integer (i - Data' first);
      begin
         buffer_1 (3 * index)        := f_32. Pass (data (i));
         buffer_1 (3 * index + 1)    := f_32. Pass ((0.0, 0.0));
         buffer_1 (3 * index + 2)    := f_32. Pass ((0.0, 0.0));
      end;
   end loop;

   for i in buffer_2' Range loop -- we know it is 0 .. X
      buffer_2 (i) :=
                   (Interfaces. C. C_float (buffer_1 (2 * i). Re),
                    Interfaces. C. C_float (buffer_1 (2 * i). Im));
   end loop;
   pa_ringBuffer. putDataIntoBuffer (Object. buffer, buffer_2);
end putSamples_32;
```

6.4 Handling devices

The audiohandler is a simple layer interfacing to the portaudio library. The portaudio library provides functions for enquiring available devices, these are used.

Our function *selectDevice* does what the name suggests: a device, a reference to which is passed as parameter, is selected. In the current implementation the function is only called for a default device in the underlying portaudio library.

```
procedure selectDefaultDevice   (Object  : in out audiosink;
                                 res     : out boolean) is
   Device_index : PaDeviceIndex := Pa_GetDefaultOutputDevice;
begin
   selectDevice (Object, res, Device_Index);
end selectDefaultDevice;
```

where $Pa\_GetDefaultOutputDevice$ provides a handle - as the name suggests - for the default output device on the system.

The main elements of *selectDevice* are setting parameters, such as the $Device\_Index$ (which is a value obtained from portaudio), the number of channels and the format of the samples.

```
    Object. Output_Parameters. device        := Device_Index;
    Object. Output_Parameters. channelCount  := 2;
    Object. Output_Parameters. sampleFormat  := paFloat32;
    if Object. Latency = LOW_LATENCY then
       Selected_Latency :=
            Pa_GetDeviceInfo (Device_Index). defaultLowOutputLatency;
    else     -- latency = HIGH_LATENCY
       Selected_Latency :=
            Pa_GetDeviceInfo (Device_Index). defaultHighOutputLatency;
    end if;

    Device_Buffersize :=
        2 * integer (float (Selected_Latency) * float (Object. cardRate));

    Error   := Pa_OpenStream (Object. Ostream,
                              null,
                              Object. Output_Parameters' access,
                              Long_Float (48000),
                              Interfaces.C.unsigned_long (Device_BufferSize),
                              0,
                              paCallback' access,
                              Object' Address);
    if error /= paNoError then
       res     := false;
       return;
    end if;
```

On opening the stream, the address of the callback function ($paCallback'access$) and a reference to the "context" is passed on as parameter, in this case $Object'Address$.

6.5 Starting and Stopping

The audio package provides functions for starting and stopping.

```
    procedure  portAudio_start (Object : in out audiosink;
                                result : out boolean) is
       Error:    PaError;
    begin
--    default:
       result   := false;
       .... -- checking code omitted

--     It took a while, but is seems we can start
       Object. Callback_Returnvalue := paContinue;
       Error := Pa_StartStream (Object. Ostream. all);
       if Error = paNoError then
          Object. Is_Running  := true;
          result              := true;
       end if;
    end portAudio_start;
```

The procedure *portAudio_start* does some consistency checks and then calls upon a function *Pa_StartStream*, a function of the underlying portaudio library. Communication with the callback function is (a.o) through the *Callback_Returnvalue* variable. As long as this is set to *paContinue*, the callback function will continue to run.

```
procedure portAudio_stop (Object : in out audioSink) is
   Error  : PaError;
begin
   if not Object. Is_Initialized or else not Object. Is_Running then
      return;
   end if;

   Object. Callback_Returnvalue := paAbort;
   Error        := Pa_StopStream (Object. Ostream. all);
   while Pa_IsStreamStopped (Object. Ostream. all) /= 1 loop
      Pa_Sleep (1);
   end loop;
end portAudio_stop;
```

The procedure *portAudio_stop* sets the return value for the callback function to *paAbort*, and it calls upon *Pa_StopStream* to try to stop the audiostream. It then just waits until the stream is stopped.

7 Handling C functions and libraries

7.1 The deconvolution

One thing that remains is the actual deconvolution, applied in both the FIC handling and the MSC handling.

In the very first version of the DAB software (in C++), a litteral translation of the standard algorithm for deconvolution was made, which turned out to work (which was positive), however it consumed well over 40 % of the CPU time used (which was negative).

Applying the spiral code library[5] speeded the software up, and was used consequently. The API to the spiral code library basically provides 5 imported functions.

```
function Create_Viterbi (Wordlength : Interfaces. C. int)
                                            return system. address;
procedure Init_Viterbi (Handle      : system. address;
                        startState  : Interfaces. C. int);
procedure Delete_Viterbi   (Handle : system. Address);
procedure Update_Viterbi_Blk (Handle    : system. Address;
                              Symbols   : dataVectype;
                              nbits     : Interfaces. C. int);
procedure Chainback_Viterbi (Handle     : system. Address;
                             Output     : out outBuffer;
                             Wordlength : Interfaces. C. int;
                             EndState   : Interfaces. C. int);
```

The interface contains three functions to control the existence of a deconvolver, and two functions that are actually used in the deconvolution.

Deconvolution is then done in two steps. In the first step we pay the price for having the ofdm decoder return values in the range -127 .. 127. The spiral code implementation expects values in the range 0 .. 255.

```
Init_Viterbi (Object. handler, 0);
for I in 0 .. (Object. wordLength + (K - 1)) * rate - 1 loop
   Temp_Value := Integer (-input (Integer (I))) + 127;
   if Temp_Value < 0 then
      Temp_Value := 0;
   elsif Temp_Value > 255 then
      Temp_Value := 255;
   end if;
   Object. symbols (I) := Interfaces. C. int (Temp_Value);
end loop;
```

The real work is then done by the functions $Update\_Viterbi\_Blk$ and $Chainback\_Viterbi$, two C functions, after which we try to get out the data.

```
Update_Viterbi_Blk (Object. Handler,
                    Object. Symbols. all,
                    Interfaces. C. int (Object. Wordlength + (K - 1)));
Chainback_Viterbi (Object. Handler,
                   Data,
                   Interfaces. C. int (Object. Wordlength), 0);

for I in 0 .. uint16_t (Object. Wordlength) - 1 loop
   output (Integer (I)) :=
               Getbit (data (Integer (Shift_Right (i, 3))),
                                      Integer (i and 8#07#));
end loop;
```

7.2 Handling the fftw library

Per DAB frame we need - in Mode 1 - mode than 76 DFT's to be executed on blocks with a length of 2048. While there are more than 10 DAB frames per second, it is obvious that a DFT function with a good performance is required.

The software uses the FFTW library[4], which is written in C and therefore some interfacing is needed.

Ada (Gnat) allows setting the aligment of arrays of certain types. By setting the alignment constraint for arrays, we can ensure that the arrays passed on for a DFT operation are well aligned.

We can create a "plan" for a DFT by creating an array and calling the function

```
function fftwf_plan_dft_1d (size     : Interfaces. C. int;
                            inVec    : System. Address;
                            outVec   : System. Address;
                            kind     : Interfaces. C. int;
```

```
                              approach : Interfaces. C. Int)
                                         return System. Address;

    pragma Import (C, fftwf_plan_dft_1d, "fftwf_plan_dft_1d");

    thePlan := fftwf_plan_dft_1d (Interfaces. C. int (fftSize),
                                  fft_Vector_w' Address,
                                  fft_Vector_w' Address,
                                  (if direction = FFTW_FORWARD
                                         then -1 else 1),
                                  FFTW_ESTIMATE
                                  );
```

We just declare the Ada view of the C function $fftwf\_execute$

```
procedure fftwf_execute (plan :  System. Address);
pragma Import (C, fftwf_execute, "fftwf_execute");
```

and our local procedure do_FFT then is

```
procedure do_FFT (v : in out fftVector) is
begin
   fft_Vector_w := v;
   fftwf_execute (thePlan);
   v            := fft_Vector_w;
end do_FFT;
```

8 Handling Devices

8.1 The interface

As mentioned before, the software for the supported devices merely consists of an interface handler between the C library for that device and our software. The device handler is selected in the command line and is implemented as a controlled type, the instance of which is created dynamically.

The functions in the interface were already mentioned:

```
package device_handler is
    type device is
        new Ada. Finalization. Controlled with record null; end record;
    type device_P is access all device' Class;
    procedure Restart_Reader   (Object       : in out device;
                                Success      : out Boolean);
    procedure Stop_Reader      (Object       : in out device);
    procedure Set_VFOFrequency (Object       : in out device;
                                New_Frequency: Natural);
    procedure Set_Gain         (Object       : in out device;
                                New_Gain     : Natural);
    procedure Get_Samples      (Object       : in out device;
                                Out_V        : out complexArray;
```

```
                                Amount       : out Natural);
    function Available_Samples (Object       : device) return Natural;
    function Valid_Device      (Object       : device) return Boolean;
end device_handler;
```

8.2 The sdrplay-wrapper

For each of the supported devices we create a derived type with code for wrapping the C code library.

```
    package sdrplay_wrapper is
        use header. complexTypes;
        type sdrplay_device is new device with private;
        type sdrplay_device_p is access all sdrplay_device;

        overriding
        procedure Restart_Reader   (Object       : in out sdrplay_device;
                                    Success      : out Boolean);
        procedure Stop_Reader      (Object       : in out sdrplay_device);
        procedure Set_VFOFrequency (Object       : in out sdrplay_device;
                                    New_Frequency: Natural);
-- some function declarations omitted
        function Valid_Device      (Object       : sdrplay_device)
                                                        return Boolean;
    private
        procedure Initialize (Object: in out sdrplay_device);
        procedure Finalize   (Object: in out sdrplay_device);

        package sdrplay_Buffer is new ringBuffer (complex);
        use sdrplay_Buffer;
        type sdrplay_device is new device with
        record
           The_Buffer        : sdrplay_Buffer. ringBuffer_data (16 * 32768);
           Running           : Boolean                         := False;
           Current_Gain      : Integer                         := 40;
           VFO_Frequency     : Integer                         := 227000000;
           err               : Interfaces. C. int;
           Library_Version   : Interfaces. C. c_float;
           isValid           : Boolean;
        end record;
```

The structure of the different wrappers is more or less the same, somehow a frequency and a samplesrate have to be set, and a callback has to be registered. While for the DABstick software one has to provide an execution thread from which the callback function is called, for the SDRplay the C library handles control of the callback function.

For the SDRplay, the access type to the callback function is

```
    type Sdrplay_Callback_Type is access
                 procedure (xi             : access Interfaces. C. short;
                            xq             : access Interfaces. C. short;
```

```
                            firstSampleNum : Interfaces. C. int;
                            grChanged      : Interfaces. C. int;
                            rfChanged      : Interfaces. C. int;
                            fsChanged      : Interfaces. C. int;
                            numSamples     : Interfaces. C. unsigned;
                            reset          : Interfaces. C. unsigned;
                            userData       : system. Address);
    pragma Convention (C, Sdrplay_Callback_Type);
```

For us, the $x1$ and xq are the important parameters, on calling the callback function, they refer to arrays with the I resp. Q values read.

As with the audio library, the callback function has to be provided with some "context", so, on initialization of the underlying C library, an access value to the instance of the type is passed.

The value passed in initialization is passed to the callback as the parameter *userdata* of type *System.Address*, and needs to be converted to an access value to the instance of the type for which we instantiate a package *Address_to_Access_Conversions*.

```
package environmentConverter is
        new System. Address_to_Access_Conversions (sdrplay_device);
```

Similarly, we need to convert the access values xi and xq to values that can be dealt with in Ada.

```
type shortArray is Array (0 .. Integer (numSamples) - 1) of int16_t;
package arrayConverter is
     new System. Address_To_Access_Conversions (shortArray);
```

With this, the implementation of the callback is pretty straightforward

```
procedure Sdrplay_Callback (xi             : access Interfaces.C. short;
                            xq             : access Interfaces.C. short;
                            firstSampleNum : Interfaces. C. int;
                            grChanged      : Interfaces. C. int;
                            rfChanged      : Interfaces. C. int;
                            fsChanged      : Interfaces. C. int;
                            numSamples     : Interfaces. C. unsigned;
                            reset          : Interfaces. C. unsigned;
                            userData       : system. Address) is
--   xi_buffer and xq_buffer are actually  C arrays,
--   provided for by the underlying sdrplay library
--   We convert them to Ada-like arrays by an "arrayConverter"
      xi_buffer      : arrayConverter. Object_Pointer :=
                               arrayConverter. To_Pointer (xi. all' Address);
      xq_buffer      : arrayConverter. Object_Pointer :=
                               arrayConverter. To_Pointer (xq. all' Address);
      localEnv       : environmentConverter. Object_Pointer :=
                       environmentConverter. To_Pointer (userData);
      collect_Buffer : sdrplay_Buffer. buffer_data (0 .. Integer (numSamples) - 1);
    begin
      for I in collect_Buffer' Range loop
```

```
         collect_Buffer (I) := (Float (xi_buffer (I)) / 2048.0,
                                Float (xq_buffer (I)) / 2048.0);
      end loop;
      localEnv. The_Buffer. putDataIntoBuffer (collect_Buffer);
   end Sdrplay_Callback;
```

The main task of the callback function is to put the incoming data into a buffer as fast as possible, and that is what the function does.

8.2.1 Starting and stopping

Starting the data transfer is - from a user's perspective - by calling the function *Restart_Reader*. That function calls *mir_sdr_StreamInit*, the C function that does the work.

```
   procedure Restart_Reader (Object  : in out sdrplay_Device;
                             Success : out Boolean) is
      sps         : Interfaces. C. int;
      gRdBSystem  : Interfaces. C. int := 0;
      agcMode     : Interfaces. C. int := 0;
      err         : Interfaces. C. int;
      localGain   : Interfaces. C. int := 20;
   begin
--  some code deleted

      err   := mir_sdr_StreamInit (Interfaces. C. int (localGain),
                                   Interfaces. C. double (inputRate)  / mHz_1,
                                   Interfaces. C. double (Object. VFO_Frequency) / mHz_1,
                                   mir_sdr_BW_1_536,
                                   mir_sdr_IF_Zero,
                                   0,
                                   gRdBSystem,
                                   agcMode,
                                   sps,
                                   Sdrplay_Callback' Access,
                                   Sdrplay_Gain_Callback' Access,
                                   Object' Address);
      if err /= 0 then
         put ("probleem init"); put_line (Integer' Image (Integer (err)));
         raise HardwareError;
      end if;
-- some code deleted
      Object. Running  := true;
      Success  := true;
   end Restart_Reader;
```

Stopping the transfer is by calling the function *Stop_Reader*, which basically encapsulates the library function *mir_sdr_StreamUnit*.

```
   procedure Stop_Reader (Object : in out sdrplay_device) is
   begin
```

```
   if not Object. Running then
      return;        -- do not bother
   end if;

   mir_sdr_StreamUninit;
   Object. Running     := false;
end Stop_Reader;
```

8.2.2 Get_Samples

While the callback function puts the data *into* a ringbuffer, the procedure *Get_Samples* takes them *out* this buffer and hands them over to the caller.

Note that in the implementation of the ofdm handler, a call to *Get_Samples* is *only* done when it is certain that the amount of requested samples is available. Since the callback function translates the incoming samples to complex values, the function is pretty simple, just fetch the samples from the buffer

```
procedure Get_Samples (Object : in out sdrplay_Device;
                       Out_V  : out complexArray;
                       Amount : out Natural) is
begin
   Object. The_Buffer.
           getDataFromBuffer (sdrplay_Buffer. buffer_data (Out_V), amount);
end Get_Samples;
```

8.2.3 Initializing the device

Initialization is in the procedure *Initialize*. The function checks on the API

```
err  := mir_sdr_ApiVersion (Object. Library_Version);
if err /= 0 then
   put_line ("Error in querying library");
elsif Object. Library_Version < 2.05 then
   put ("Library version too old");
else
   put ("Library version ");
   put_line (Float' Image (Float (Object. Library_Version)));
end if;
```

Then it looks for the availability of a device.

```
mir_sdr_GetDevices (devDesc, numofDevs, 4);
```

In this Ada implementation we only use device "0".

```
mir_sdr_SetDeviceIdx (0);
```

9 Results and issues

The resulting Ada program runs well on my laptop. It consists of app 40 Ada packages, with app 6500 lines of code (including the texts on copyright and GPL) in the package bodies, about 4000 lines of code (including the text on copyright and GPL) in the packages, and about 800 lines of C code.

The structure of the programs in Ada and C differs in the GUI handling, in the application of tasks and in a number of details.

Since we have chosen to reduce complexity of the Ada version by having the selection of a a device, the Mode and the Band through the command line, the ofdm handler, the FIC and the MSC handler could be simple generic packages in Ada. In the Qt-DAB version, on the other hand, all three items can be selected - and modified - through the GUI, which - in the implementation form as chosen - requires dynamic allocation of the handlers.

The task structure in the Ada version resembles to a certain extent the structure in the Qt-DAB version. For the ofdm handling we use two tasks, in both the Qt-DAB and the Ada version. In the Qt-DAB version, however, the FIC handling is done within the thread of the $ofdmDecoder$ thread, while the FIC handling in the Ada version is done in its own task. The rationale being that that is the easiest solution to handle mutual exclusion in the FIB/FIC processing. Mutual exclusion is an issue since while data from the ofdm handling is being processed, queries may come from the GUI.

In the Qt-DAB version the msc handling is done within the thread of the $ofdmDecoder$, while the subsequent DAB handling is done in its own thread. In the Ada version the msc handling is done in its own task, again to ease the implementation of mutual exclusion, while the subsequent DAB processing is done from within this task.

Furthermore, to decouple the ofdm handling and the msc handling, an intermediate buffer task is used in the Ada version.

9.1 Unmentioned parts of the implementation

The implementation of three - important - parts of the program are not discussed:

- the checker for the firecode in the $mp4\_handler$,
- the $reed\_solomon$ and the $galois$ packages for the implementation of the Reed-Solomon decoding,
- the ringbuffer.

9.1.1 The firecode checker

The first one, the package $firecode\_checker$ is a translitteration of the C module with the same name, which is itself a translitteration of the module from the GNU radio code[10].

9.1.2 The Reed-Solomon Error recovery

The Ada version of the reed-Solomon decoder, in package *reed_solomon.adb*, is a translitteration of the C++ version used for Qt-DAB, which in turn is a (partial) rewrite of the Reed-Solomon decoder from Phil Karn[11] from 2002. While the implementation is pretty standard (i.e. computing the syndromes using Horner, computing Lambda with Berlekamp-Massey, applying a Chien search, and computing and repairing the errors), a discussion would involve a lengthy discussion, we refer to a paper[12].

9.1.3 The Ring Buffer

The third one, the implementation of the package $ringbuffer$ is a translation/translitteration of the ringbuffer of the portaudio[3] library. The package is generic, with as parameter the element type, and it implements a lockfree single producer/consumer buffer.

9.2 GUI handling

A significant difference between the Ada and the C++ version(s) deals with the GUI. The C++ implementation uses the Qt framework for a.o creating a GUI, the Ada implementation uses GtkAda.

While the layout of the GUI in the C++ implementation was done with the designer tool from the Qt toolbox - and therefore basically uninteresting from a programming point of view, the layout of the GUI in the Ada implementation is created by explicitly coding the relationship between the GUI elements. The relationship is expressed by explicitly attaching GUI elements to other GUI elements.

```
Win.Add (Grid);
Gtk_New (startButton, "Start");
Grid. Attach (startButton, 0, 0, 1, 2);

Gtk_New (quitButton, "Quit");
Grid. Attach_Next_To (quitButton, startButton, POS_BOTTOM, 1, 2);

Gtk_New (Channel_Selector);
Grid. Attach_Next_To (Channel_Selector, quitButton, POS_RIGHT, 2, 1);
... -- more code to come
```

Qt provides a strong signal-connect mechanism with which it is not only possible to pass on signals from the GUI to the gui handler, it provides possibilities of signaling from other tasks to the GUI. Important when we want to display (computed) entities, such as the SNR, the quality of the FIC signals etc etc. of the GUI.

In GtkAda it is certainly possible to handle signals generated by e.g. a mouse handling on the screen in the gui handling.

```
--  called after pressing the start button
   procedure start_clicked (Self : access Gtk_Button_Record' Class) is
      Result : Boolean;
   begin
```

```
      if Running then
         return;
      end if;

      My_Device. Restart_Reader (result);
      if not Result then
         put_line ("device did not start");
      else
         Running := true;
         programSelector. Remove_All;
         my_ficHandler. reset;
         my_ficHandler. restart;
         my_ofdmHandler. reset;
      end if;
   end start_clicked;
```

It is, however, - at least for me - not clear whether or not GtkAda provides a mechanism for handling signals from other tasks in the GUI.

In order to be able to show messages from the various tasks on the GUI, a queuing mechanism, based on a protected object (actually, 2) was created. The messages are values to be displayed on the GUI. While the first 4 signals relate to numbers to be displayed, the 5-th one is a message to add a character to the text line in the GUI.

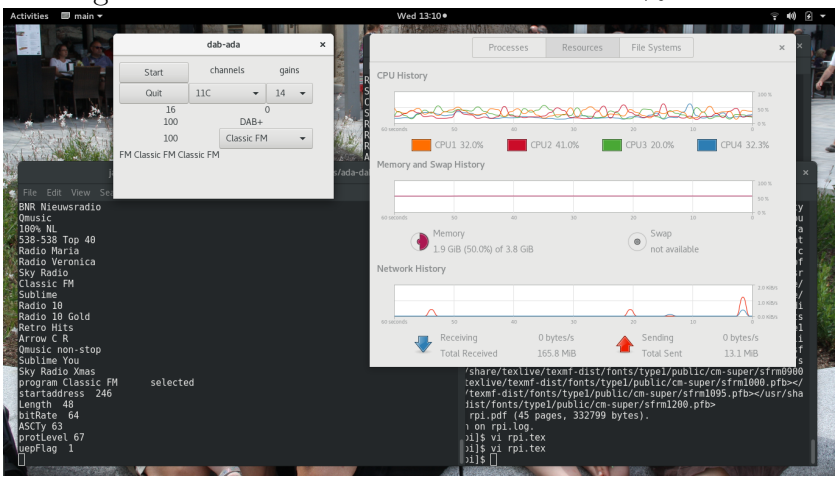
```
package simple_messages is
   type SIGNAL is (FIC_RESULTS,
                   MSC_RESULTS,
                   FINE_CORRECTOR_SIGNAL,
                   COARSE_CORRECTOR_SIGNAL,
                   TEXT_SIGNAL);
   type message is
      record
         key: SIGNAL;
         val: Integer;
      end record;
   package message_handler is new Generic_Buffer (message);
   use message_handler;
   message_queue: message_handler. Buffer (50);
end simple_messages;
```

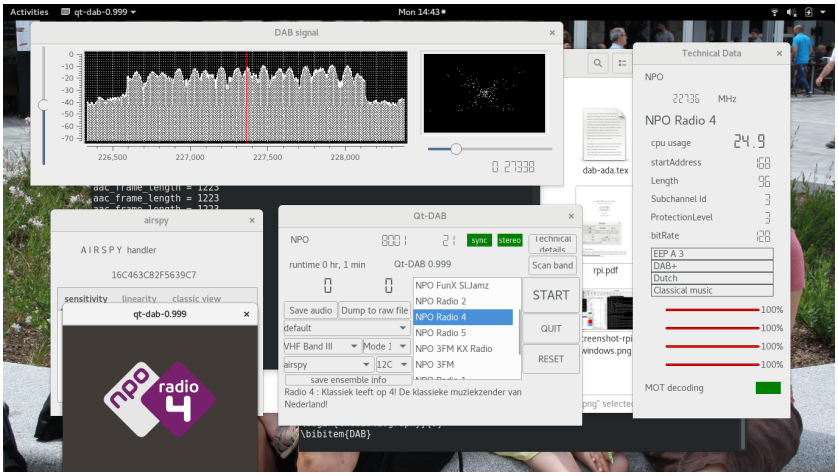
At the receiving size, a function *Dispatch* (member of *gui*) is available to dispatch the messages. The function is basically polling: it is called twice a second.

9.3 CPU load

The CPU load of the Ada program is - as expected - higher, due to the large(r) amount of run time tests being executed, than the load of executing the Qt-DAB program. However, the load is fully acceptable. Both programs are run on a laptop with an Intel Core I5-2450M cpu running at 2.5G (a rather old one). While the average load when running the Qt-DAB program - with spectrum being displayed - is around 25 %, the load when

running the Ada version is between 30 and 35 %.





References

[1] Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers. ETSI EN 300 401.

[2] see "http://sdr.osmocom.org/trac/wiki/rtl-sdr".

[3] , see "www.portaudio.com".

[4] see "www.fftw.org".

[5] see "www.spiral.net/codegenerator.html".

[6] see "www.adacore.com".

[7] see "https://github.com/AdaCore/gtkada".

[8] see "http://keyj.emphy.de/", private communication.

[9] see "www.audiocoding.com/faad2.html" and "http://drm.sourceforge.net/wiki/index.

[10] see "www.gnuradio.com".

[11] C files found on the internet.

[12] Reed-Solomon error correction, C.K.P Clarke, BBC R & D White Paper, WHP 031, July 2002.