Software Implementation of a Digital Radio Mondiale (DRM) Receiver, Part I (Framework)

Volker Fischer
Institute for Communication Technology
Darmstadt University of Technology
v.fischer@nt.tu-darmstadt.de

Abstract— **The bandwidth of a DRM pass-band signal is smaller than 20 kHz and the number of carriers used in the OFDM-modulation is relatively small (Max 460). This characteristic motivates a real-time software implementation of a DRM-receiver on a conventional personal computer (PC) using the soundcard as the input and output device. In this paper, the basic concept and the framework of an actual implementation is described.**

I. Introduction

THE DRM-receiver consists of several modules (OFDM-demodulation, channel decoder, demultiplexer, etc.) and multiple data streams [1]. Each module is defined to work on a block-wise processing of data. Since some modules need more or less data to perform a certain task, an intelligent data transfer between the modules is needed. Also the merging and splitting of data streams in the multiplexer has to be taken care of. To separate the specific implementation of a module from the standard task of organizing the data transfer between the modules, an object-oriented implementation was chosen. Furthermore, this type of implementation increases the clearness and provides an easier maintenance of the resulting code.

The resulting framework consists of a class `CBuffer`, which manages the data transfer between different modules, and a class `CModule`, from which all modules are derived.

The basic concept of the framework is described in Sect. II. In Sect. III the class `CModule` is described. The cyclic-buffer class `CBuffer` is specified in Sect. IV.

II. Basic concept

Since we have a block-wise processing of quasi continuous realtime input- and output-data-streams, we have to decide whether the input or the output stream defines the timing. In this implementation we use an input-driven processing sequence. When a new data block from the sound-card is available, the data is processed and fed to the output device (e.g. sound-card or a network stream).

The main loop of the receiver calls the "processing" - routine of the base-class `CModule` of all modules successively. This routine checks the intermediate cyclic input-buffer (derived from `CBuffer`) whether enough data is available or not. When data is available, the processing-routine of the derived class is called. This routine produces an output which is stored in the intermediate cyclic-buffer of the following module. Thereby, the sequence of the modules in the main loop is irrelevant. The resulting structure is illustrated in Figure 1.

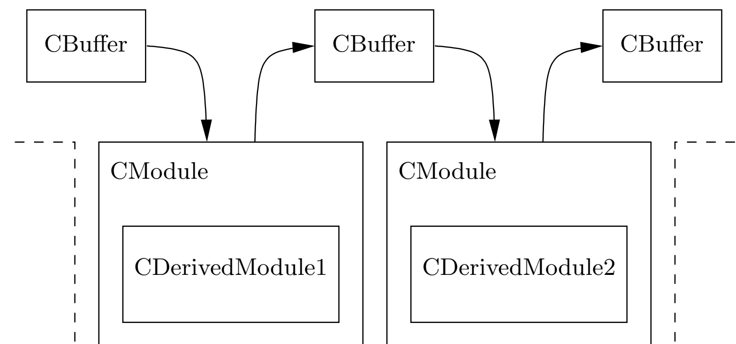

Fig. 1. Illustration of dependency between CModule, CBuffer and derived modules.

III. Class CModule

The class `CModule` basically relieves the implementation of a derived module from managing the data transfer between different modules. For a single-input, single-output module, `CModule` defines two internal buffers and two variables, signalling the size of the buffers (one for input and one for output data blocks). `CModule` is responsible for allocating and deleting these resources. These buffers contain exactly the amount of data needed for one module-cycle. The module is responsible for setting the right buffer-sizes, which is done in a special initialization-routine. In case that the block sizes vary with time, a maximum block-size must be defined. The current size is then adjusted in the processing routine of the module.

The task for the `CModule` is to check the cyclic-input-buffer (`CBuffer`) whether enough data for processing is available. If so, it has to copy one block of data from the cyclic-buffer to the internal memory block and afterwards call the processing routine. When the data is processed, `CModule` copies the output-data from the internal buffer in the cyclic-buffer of the following module.

For multiple-input blocks, an internal buffer for all input streams has to be provided by the `CModule` class. In that case, the processing routine is called when all internal buffers are filled.

IV. Class CBuffer

The class `CBuffer` is an implementation of a cyclic-buffer. The advantage of using a cyclic-buffer is to use different sizes of input- and output-blocks, which is necessary for the different DRM-modules.

This implementation emulates a cyclic-buffer by using a linear buffer and managing the wrap-around of data-blocks. It defines two pointers which store the position of the beginning and end of the useful data-block in the buffer. When new data is read or written, the positions of the pointers are adapted. An ambiguous situation occurs when both pointers point to the same place. In this case, the buffer could be empty or total full. To obviate this ambiguity, a flag is introduced to indicate the state of the buffer after reading or writing a block of data. With this design, Data-blocks of any size can be written until the total buffer-size is reached. In the same manner, data can be read until the buffer is empty.

To be type-independent, this class is a template-class. The data type of the memory must be defined, when the class object is declared.

References

[1] European Telecommunications Standards Institute: Digital Radio Mondiale (DRM), *System Specification ETSI TS 101980*, 2001.