SoC Software Project
Fixed-Point Conversion of DRM Software

Spring 2008
Agenda

- Objectives
- DRM Software Overview
- Conversion Process
- Challenges
- Recommendations
Objectives

- Optimize the freely available DRM source to run in real-time on an ARM processor.
- Utilize the co-processor to offload complex, time-consuming tasks from the main CPU to reduce CPU utilization to 25% or less.
Objectives

- **Given the DRM C++ code:**
  - To convert Floating Point variables (FLP) to Fixed Point (FXP) variables with correct precision.
  - To convert the data members (fields) of each class in the header files.
  - To convert local variables and parameter of the methods (functions) from Floating Point to fixed point.
  - BE SURE THAT WORK in ARMSD
Agenda

- Objectives
- DRM Software Overview
- Conversion Process
- Challenges
- Recommendations
Block Diagram of DRM Software

Sound card interface
   ReceiveData()

Resampling
   InputResample()

Frequency sync acquisition
   Frequency offset correction
      FreqSyncAcq()

Time sync acquisition
   Guard interval removal
      TimeSync()

OFDM demodulation
   OFDMDemodulation()

Sync using pilots:
   DRM frame sync,
   (sample) frequency offset tracking
      SyncUsingPil()

Channel estimation,
   Time sync tracking
      ReceiveData()

OFDM cell demapping
   OFDMCellDemapping()

Audio source decoder
   AudioSourceDecoder()

MSC demultiplexer
   MSCDemultiplexer()

MSC MLC decoder
   MSCMLCDecoder()

MSC symbol deinterleaver
   SymbDeinterleaver()

SDC MLC decoder
   SDCMLCDecoder()

FAC MLC decoder
   FACMLCDecoder()

Use SDC
   UtilizeSDCData()

Use FAC
   UtilizeFACData()
Module Overview: CDRMReceiver

- “Relevant” top level class
- Instantiates
  - Each “submodule” class
  - Data buffers

- Submodules execute sequentially
  - (when input buffer is sufficiently full)
Module Overview

- **Derived from base class CModul**
  - Input/Output data buffer pointers
    - pvecInputData
    - pvecOutputData
  - Virtual function declarations for Init, ProcessData

- **Modules must define the work they do:**
  - ProcessDataInternal
CModul Base Class Declaration

template<class TInput, class TOutput>
class CModul
{
  public:
    CModul();
    virtual ~CModul() {} 

    virtual void Init(CParameter& Parameter);
    virtual void Init(CParameter& Parameter, CBuffer<TOutput>& OutputBuffer);

  protected:
    CVectorEx<TInput>* pvecInputData;
    CVectorEx<TOutput>* pvecOutputData;

    /* Max block-size are used to determine the size of the required buffer */
    int iMaxOutputBlockSize;
    /* Actual read (or written) size of the data */
    int iInputBlockSize;
    int iOutputBlockSize;

    void Lock() {Mutex.Lock();}
    void Unlock() {Mutex.Unlock();}

  virtual void InitThreadSave(CParameter& Parameter);
  virtual void InitInternal(CParameter& Parameter) = 0;
  void ProcessDataThreadSave(CParameter& Parameter);
  virtual void ProcessDataInternal(CParameter& Parameter) = 0;

  private:
    CMutex Mutex;
};
Class Information

- Each module defines its own ProcessDataInternal function
- These functions, and each function that they call, must be converted from floating-point to fixed-point
Agenda

- Objectives
- DRM Software Overview
- Conversion Process
  - Stage 0
    - FXP Class
  - Stage 1
  - Stage 2
- Challenges
- Recommendations
FXP Conversion Process

- **Staged approach:**
  - **Stage 0: Overhead**
    - Develop the FXP class
    - Duplicate the Matlab functions in FXP
  - **Stage 1: Internal conversion of modules**
    - Temporary FXP-based buffers to store input/output
    - Internal float/double variables → FXP variables
    - Watch carefully the ranges of values!!
  - **Stage 2: Combine converted modules**
    - Replace input/output buffers with FXP equivalents
    - Eliminate the temporary FXP-base buffers
Conversion Stage 0: Fixed-Point Infrastructure

- Generate FXP Class
- Duplicate Matlab library in FXP
FXP Class

- To Brainstorm and Make test which could be the solution to convert to Fixed Point.
  - Change _REAL type (double) for _FXP (int or long). We need to changes Transmitter too.
  - Create a structure for FXP. What about the Mathematic Libraries?
  - So here came FXP.h library from Craig.
**FXP Class**

- C++ Fixed-point class containing a single 32-bit integer member
- Defines numerous constructors and casts to convert between all numeric data types used in DRM
- Defines all mathematical and comparison operators used in DRM
  - FXP to other types
  - Other types to FXP
  - FXP $\leftrightarrow$ Float/Double operations deliberately omitted
- Defines numerous other functions
  - Trig, exp, log functions are currently wrappers around float versions
**FXP Class**

- Intended to be configurable to support a different precision level (would it work?)

- **16-bit precision was chosen**
  - Unscientific analysis of data being passed between top-level modules
    - All floating point data at this level was between +/-32768, Most-significant digit > 0.1
    - 16-bit signed whole number representation left 16-bits for fractional representation
    - Resolution of 0.000015

- Provided a standard between blocks
Conversion Stage 1: Internal Variables

- All float/double-based variables within the module (and all subfunctions) are typed to FXP-based variables
- If input buffer is RHS argument, use the temporary FXP version instead
- Utilize the FXP-based Matlab functions generated in Stage 0
- Ensure variables always fit within range of FXP!!!
Conversion Stage 1: Temporary Buffers

- From MLC.cpp:

```cpp
CVectorEx<CFEquSig> fpvecInputData_buf;
CVectorEx<CFEquSig> * fpvecInputData = &fpvecInputData_buf;

fpvecInputData_buf.Init(pvecInputData->Size());
for(i=0;i<pvecInputData->Size();i++)
{
    (*fpvecInputData)[i].cSig = (*pvecInputData)[i].cSig;
    (*fpvecInputData)[i].rChan = (*pvecInputData)[i].rChan;
}
(*fpvecInputData).SetExData((*pvecInputData).GetExData());

* Temporary output buffer necessary only if used as RHS argument internally!
```
Conversion Stage 1: FXP Scaling

- Often encountered operations with numbers beyond the range of FXP

- Round-off of smaller values represented a large error percentage
Conversion Stage 1: FXP Scaling

Utilities.cpp:

```cpp
const CReal scaler = (CReal) SOUNDCRD_SAMPLE_RATE / 44100.0;
```

Numerator and denominator both exceed range of FXP

Can’t perform integer division as you would lose a critical fractional component.

Solution: Scale both numbers equally, perform operation as FXP (error < 0.3%)

Result:

```cpp
const CFReal scaler = (CFReal) (SOUNDCRD_SAMPLE_RATE>>1) / (44100>>1);
```
Conversion Stage 1: FXP Scaling

ChannelEstimation.cpp

```cpp
_REAL CChannelEstimation::GetDelay() const
{
    return rLenPDSEst * iFFTSizexn /
            (SOUNDCRD_SAMPLE_RATE * iNumIntpFreqPil * iScatPilFreqInt) *1000;
}
```

◆ Values

- rLenPDSEst: 0-80
- iFFTSizexn: 1024
- SOUNDCRD_SAMPLE_RATE: 48000
- iNumIntpFreqPil: 104
- iScatPilFreqInt: 2
Conversion Stage 1: FXP Scaling

- C++ converts each quantity to FXP as it operates from left to right
  - rLenPDSEst * iFFTSizeN can exceed FXP range by 2-4x
  - SOUNDCRD_SAMPLE_RATE exceeds FXP before multiplication by other constants
  - To accommodate 1000 factor, may be sacrificing precision
Conversion Stage 1: FXP Scaling

- **Range of** rLenPDSEst * iFFTSzizeN **necessitates scaling down by 4**
- **Must compensate in the denominator**
- **Know that** SOUNDCRD_SAMPLE_RATE **is a constant and is divisible by 4, 1000 and (4*1000)**

```cpp
_REAL CChannelEstimation::GetDelay() const
{
    return rLenPDSEst * iFFTSzizeN /
         (SOUNDCRD_SAMPLE_RATE * iNumIntpFreqPil * iScatPilFreqInt) *1000;
}

_FREAL CChannelEstimation::GetDelay() const
{
    return rLenPDSEst * (iFFTSzizeN>>2) /
         (SOUNDCRD_SAMPLE_RATE/4000 * iNumIntpFreqPil * iScatPilFreqInt);
}  ```