```
{
    Disp("\n There is an error in the SEQ_INIT result\n");
    return;
}

//u32 uFrameBufAddr = oMfc.m_uStreamBufEndAddr[uProcessIdx]+STREAM_WR_SIZE;

MFC_GetDecSrcFormat(&picX, &picY, &frameRate);
oMfc.m_uPicX[uProcessIdx] = picX;
oMfc.m_uPicY[uProcessIdx] = picY;
Disp("%d x %d @%.2f Hz\n", picX, picY, frameRate);
Assert(picX > 0);
Assert(picY > 0);

oMfc.m_bDecRotEn[uProcessIdx] = bDecRotEn;
oMfc.m_uRotFrameIdx[uProcessIdx] = 0;
oMfc.m_uMp4DeblockFrameIdx[uProcessIdx] = 0;
oMfc.m_uFrameIndex[uProcessIdx] = 0;

// H.263 Annex J deblock is in-loop filter, otherwise deblock is out-loop filter
MFC_IsDecH263AnnexJOn(uProcessIdx, &oMfc.m_bAnnexJOn[uProcessIdx]);

oMfc.m_uFrameDelayCount[uProcessIdx] = mfcInp32(RET_DEC_SEQ_FRAME_DELAY);
//DbgMfc("Delay Frame num=%d\n", oMfc.m_uFrameDelayCount[uProcessIdx]);

MFC_GetDecRefFrameNum(uProcessIdx, &uNumOfRefReconFrame);
//DbgMfc("num of RefFrame = %d\n", uNumOfRefReconFrame);

uFrameBufNumTemp    =    (oMfc.m_bDecRotEn[uProcessIdx])    ?    uNumOfRefReconFrame+2    :
```

```
uNumOfRefReconFrame;
    uFrameBufNum = (oMfc.m_bMp4DecDeblkMode[uProcessIdx] && !oMfc.m_bAnnexJOn[uProcessIdx]) ?
uFrameBufNumTemp+2 : uFrameBufNumTemp;

    uStride = (picX%16 == 0) ? picX : (picX+15)/16*16;
    uHeight = (picY%16 == 0) ? picY : (picY+15)/16*16;
    MFC_InitDecFrameBuffer(uProcessIdx, uFrameBufNum, uStride, uHeight, uFrameBufStAddr);

    MFC_IssueCmdOfSetFrameBuffer(uProcessIdx, uNumOfRefReconFrame, uStride);
}
```

多格式视频编解码器的解码处理初始化代码：

```
void MFC_InitProcessForDecoding(
    u32 uProcessIdx, MFC_CODEC_MODE eCodecMode, u32 uStreamBufStAddr, u32 uStreamBufSize,
    u32 uFrameBufStAddr, bool bDecRotEn, bool bMp4DeblkEn, bool bH264ReorderEn)
{
    u32 uStreamBufSizeCeilingToKbMultiple;
    u32 uMp4DecDeblkMode;
    u32 uH264DecReorderMode;
    bool stat;
    float frameRate;
    u32 picX, picY;
    u32 uNumOfRefReconFrame;
    u32 uStride;
    u32 uHeight;
    u32 uFrameBufNumTemp;
    u32 uFrameBufNum;

    Assert(uProcessIdx < MAX_PROCESS_NUM);
    Assert(eCodecMode == MP4_DEC || eCodecMode == AVC_DEC || eCodecMode == VC1_DEC);
```

```
    oMfc.m_eCodecMode[uProcessIdx] = eCodecMode;
    uStreamBufSizeCeilingToKbMultiple = (uStreamBufSize+1023)/1024*1024;

    oMfc.m_uStreamBufStAddr[uProcessIdx] = uStreamBufStAddr;
    oMfc.m_uStreamBufEndAddr[uProcessIdx] = uStreamBufStAddr +
uStreamBufSizeCeilingToKbMultiple;
    oMfc.m_uStreamBufByteSize[uProcessIdx] = uStreamBufSizeCeilingToKbMultiple;

    oMfc.m_uBitRdPtr[uProcessIdx] =
        (uProcessIdx == 0) ? BITS_RD_PTR0 :
        (uProcessIdx == 1) ? BITS_RD_PTR1 :
        (uProcessIdx == 2) ? BITS_RD_PTR2 :
        (uProcessIdx == 3) ? BITS_RD_PTR3 :
        (uProcessIdx == 4) ? BITS_RD_PTR4 :
        (uProcessIdx == 5) ? BITS_RD_PTR5 :
        (uProcessIdx == 6) ? BITS_RD_PTR6 : BITS_RD_PTR7;
    oMfc.m_uBitWrPtr[uProcessIdx] =
        (uProcessIdx == 0) ? BITS_WR_PTR0 :
        (uProcessIdx == 1) ? BITS_WR_PTR1 :
        (uProcessIdx == 2) ? BITS_WR_PTR2 :
        (uProcessIdx == 3) ? BITS_WR_PTR3 :
        (uProcessIdx == 4) ? BITS_WR_PTR4 :
        (uProcessIdx == 5) ? BITS_WR_PTR5 :
        (uProcessIdx == 6) ? BITS_WR_PTR6 : BITS_WR_PTR7;

    mfcOutp32(oMfc.m_uBitRdPtr[uProcessIdx], oMfc.m_uStreamBufStAddr[uProcessIdx]);
    mfcOutp32(oMfc.m_uBitWrPtr[uProcessIdx],
oMfc.m_uStreamBufStAddr[uProcessIdx]+uStreamBufSize);
```

```
    mfcOutp32(DEC_SEQ_BIT_BUF_ADDR, oMfc.m_uStreamBufStAddr[uProcessIdx]);
    mfcOutp32(DEC_SEQ_BIT_BUF_SIZE,  oMfc.m_uStreamBufByteSize[uProcessIdx]/1024);     // KB
unit

    oMfc.m_bMp4DecDeblkMode[uProcessIdx] = (eCodecMode == MP4_DEC) ? bMp4DeblkEn : false;
    uMp4DecDeblkMode   =   (oMfc.m_bMp4DecDeblkMode[uProcessIdx])   ?   MP4_DBK_ENABLE   :
MP4_DBK_DISABLE;
    uH264DecReorderMode = (bH264ReorderEn) ? REORDER_ENABLE : REORDER_DISABLE;
    mfcOutp32(DEC_SEQ_OPTION, uMp4DecDeblkMode|uH264DecReorderMode);

    oMfc.m_bIsNoMoreStream[uProcessIdx] = false;

    MFC_IssueCmd(uProcessIdx, SEQ_INIT);

    stat = MFC_IsCmdFinished();

    if(stat == false)
    {
        Disp("\n There is an error in the SEQ_INIT result\n");
        return;
    }

    MFC_GetDecSrcFormat(&picX, &picY, &frameRate);
    oMfc.m_uPicX[uProcessIdx] = picX;
    oMfc.m_uPicY[uProcessIdx] = picY;
    Disp("%d x %d @%.2f Hz\n", picX, picY, frameRate);
    Assert(picX > 0);
    Assert(picY > 0);
```

```
    oMfc.m_bDecRotEn[uProcessIdx] = bDecRotEn;
    oMfc.m_uRotFrameIdx[uProcessIdx] = 0;
    oMfc.m_uMp4DeblockFrameIdx[uProcessIdx] = 0;
    oMfc.m_uFrameIndex[uProcessIdx] = 0;

    MFC_IsDecH263AnnexJOn(uProcessIdx, &oMfc.m_bAnnexJOn[uProcessIdx]);

    oMfc.m_uFrameDelayCount[uProcessIdx] = mfcInp32(RET_DEC_SEQ_FRAME_DELAY);

    MFC_GetDecRefFrameNum(uProcessIdx, &uNumOfRefReconFrame);

    uFrameBufNumTemp = (oMfc.m_bDecRotEn[uProcessIdx]) ? uNumOfRefReconFrame+2 :
uNumOfRefReconFrame;
    uFrameBufNum = (oMfc.m_bMp4DecDeblkMode[uProcessIdx]
 && !oMfc.m_bAnnexJOn[uProcessIdx]) ? uFrameBufNumTemp+2 : uFrameBufNumTemp;

    uStride = (picX%16 == 0) ? picX : (picX+15)/16*16;
    uHeight = (picY%16 == 0) ? picY : (picY+15)/16*16;
    MFC_InitDecFrameBuffer(uProcessIdx, uFrameBufNum, uStride, uHeight, uFrameBufStAddr);

    MFC_IssueCmdOfSetFrameBuffer(uProcessIdx, uNumOfRefReconFrame, uStride);
}
```

22　JPEG 编解码器

JPEG编解码器的核心是由控制电路，DCT/量化，哈夫曼编码，标志处理块和AHB从接口控制组成的，如图22-1所示。输入/输出图像总线和压缩数据总线是8位，它控制内部的寄存器。

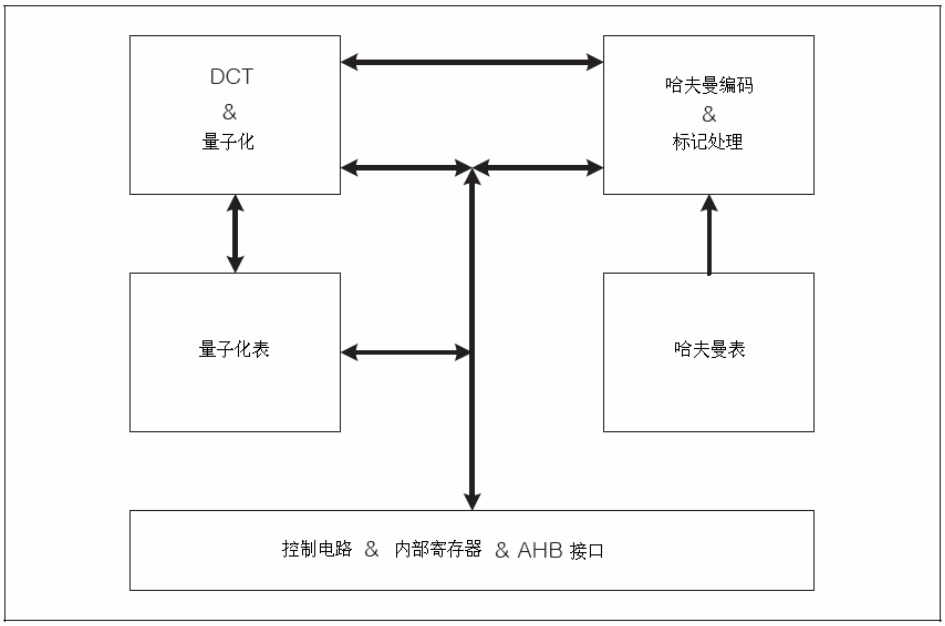

图 22-1 JPEG 编解码器结构框图

22.1 JPEG 编码特性

JPEG 编解码器包含有以下特性：

- 压缩/解压缩高达UXGA。
- 编码格式：YCbCr4:2:2或YCbCr4:2:0(JPEG引擎的输入格式)。
- 解码格式：YCbCr4:4:4，YCbCr4:2:2，YCbCr4:2:0，YCbCr4:1:1或格雷码。
- 支持直接压缩，从相机输出。
- 在YCbCr4:2:2或RGB565格式中，支持内存数据的压缩。
- 支持通用颜色转换器。

22.2 JPEG 编码定义

1.控制电路和 AHB 接口

该模块设置和初始化操作模式，由内置寄存器组成。它设置操作模式，来确定量化哈夫曼表数目和DRI值。

2.DCT/量化

在编码期间，JPEG 编解码器将 8×8 图像数据转换为 DCT 系数。因此，量化处理是通过利用量化表来执行 DCT 系数的。在解码期间，反量化将完成，然后 DCT 系数转化成图像数据。

3.哈夫曼编码和标记处理

各种长度的编码和解码都是基于哈夫曼表的。

4.量化表

主要用来存储量化表。这是 RAM 区域，用户可以对其进行分配。

5.哈夫曼表

主要用来存储哈夫曼表。这是RAM区域，用户可以对其进行分配。

6.寄存器访问

该寄存器被修改：

- 复位后，直到一个新的工作开始。
- 处理完成后，中断信号产生，直到新的工作开始。

其他条件表明，核心是在正常运作，因此不允许修改。对于一些寄存器，无论是写或读都可能被禁止。

7.表访问

在压缩前，必须对四个哈夫曼表（(AC&DC, 每 2 个表）和四个量化表进行配置。设置任何量化表和哈夫曼表，必须首先访问相应的入口寄存器。因此，必须有写传输脉冲跟随。为了更好地了解写传输脉冲，参看图 22-2。每个表的访问顺序显示如下：

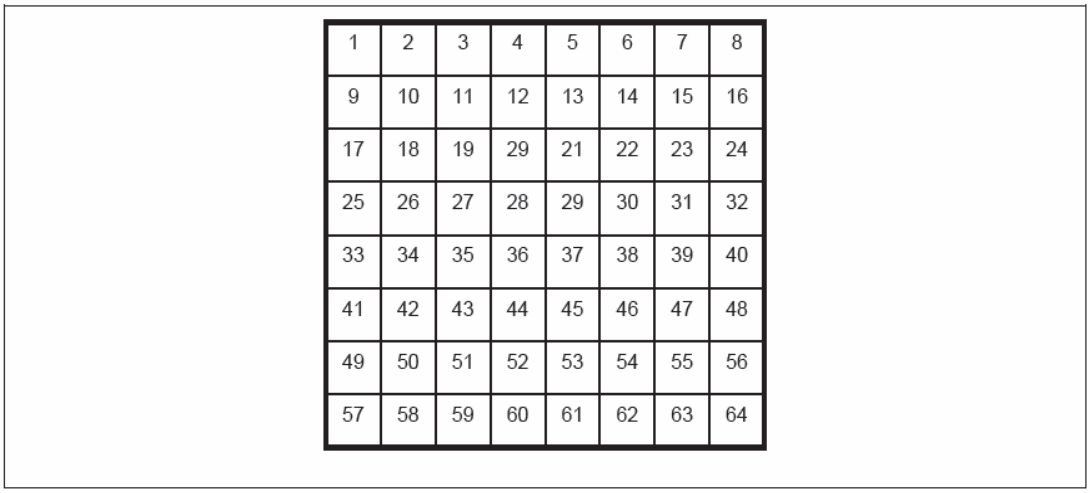

图 22-2 量化表的访问顺序

8.中断信号

中断信号将在以下条件下产生（寄存器 JPGIRQ，用来确定原因）：

- 用于一帧的压缩或解压缩处理完成。
- 在解压缩期间，JPGIRQ[3]被设置为高电平。在标记分析之后，寄存器保存图像的大小和取样因子被读出。

中断 1，正常的处理完成。清除待处理的中断请求，读 JPGSTS 寄存器。如果没有编码或解码的错误，JPGIRQ 将以 0x40 被读出。

取消中断 2 也是通过读 JPGIRQ 来完成的。如果没有标题解析错误，其将以 0x08 读出。中断 2 表示解压缩处理被终止。

中断 3 信号用于区别中断发生的条件。

9.中断设置寄存器

当准备解压缩一个图像时，该寄存器用来设置是否允许中断。为了允许中断，在开始解压缩处理之前，设置 JPGIRQS[3]为高电平。当该中断发生时，JPEG 编解码器终止处理，同时驱动 PGSTS[0]为高电平。通过读取 JPGIRQ，设置该寄存器来取消中断。

10.标记处理

标记是在压缩过程中产生的。如表 22-1 所示。

表 22-1 JPEG 编解码器的标记

标记	编解码器（Hex）	描述
SOI	FFD8	图像开始
SOF0	FFC0	基线 DCT
SOS	FFDA	扫描开始
DQT	FFDB	定义量化表
DHT	FFC4	定义哈夫曼表
DRI	FFDD	重新界定区间
RSTm	FFD0～FFD7	重新开始以模块 8 计数“m”
EOI	FFD9	图像结束

在解压缩期间，表22-1中的标记得到处理。除了SOF1～SOFF和JPG，其他标记将被忽略。

11.压缩文件的位流

创造JPEG的位流显示如图22-3所示。

图 22-3 JPEG 文件的位流结构框图

12.程序员模型

如表 22-2 所示，显示了寄存器编码及解码过程。

表 22-2 寄存器编码及解码过程

寄存器	描述	编码过程	解码过程
JPGMOD	流程模式寄存器	必要的	必要的
JPGQHNO	量化和哈夫曼表数目寄存器	必要的	--
JPGDRI	重置区间寄存器	必要的	--
JPGY	垂直大小寄存器	必要的	
JPGX	水平大小寄存器	必要的	

QTBL0	量化表0入口寄存器	必要的	--
QTBL1	量化表1入口寄存器	必要的	--
QTBL2	量化表2入口寄存器	必要的	--
QTBL3	量化表3入口寄存器	必要的	--
HDTBL0, HDCTBLG0	直流哈夫曼表0入口寄存器	必要的	--
HACTBL0, HACTBLG0	交流哈夫曼表0入口寄存器	必要的	--
HDCTBL1, HDCTBLG1	直流哈夫曼表1入口寄存器	必要的	--
HACTBL1, HACTBLG1	交流哈夫曼表1入口寄存器	必要的	--

表 22-2 中任意寄存器的内容都不会改变，除非重新写入或被重置。因此，只有通过执行开始命令过程，才有可能处理下一帧。通过在 SW_JSTART 寄存器中写 0x1，重新开始。

13.JPEG 编解码器设计向导

JPEG 引擎有它自己的内部等待标记，它是通过软件读取 JPGIRQ 来清除的。如果所有的处理都完成了，读 JPGSTS 以清除所有内部中断的等待标记。在 JPEG 处理和硬件控制解码模式下，JPEG 引擎自动地清除所有等待标记。

基本的 JPEG 编码顺序，如图 22-4 所示。