

基于Cortex-M3的ADuCxxx串行下载协议

简介

基于Cortex-M3的ADuCxxx的一个关键特性是可以在线下载代码到片内FLASH/EE程序存储器。在线代码下载是通过器件UART串行端口进行的，因此一般被称为串行下载。

利用串行下载功能，开发人员可以在将器件直接焊接到目标系统的同时对其重新编程，从而不需要外部器件编程器。此外，只需一个能访问基于Cortex-M3的ADuCxxx的串行端口，就可以在现场通过串行下载特性执行系统升级。这意味着制造商可以在现场升级系统固件，而不必换出器件。

在上电时或者在任何复位或特定复位之后，通过特定引脚配置可以将任何基于Cortex-M3的ADuCxxx配置为串行下载模式。

参见器件特定用户指南了解串行下载模式的输入标准。例如在ADuCM360上时，P2.2输入引脚在内核执行期间检查。如果该引脚在上电或任何类型的复位之后保持低电平，则器件进入串行下载模式。

在此模式下，片内驻留的加载器程序会启动。配置器件UART，并通过特定串行下载协议与任何主机通信，以管理下载的数据，将其存入Flash/EE存储器空间。要下载的程序数据必须是从小到大顺序格式。

注意，串行下载模式工作在器件的标准电源额定值范围。无需特别高的编程电压，因为它是在片内产生的。

用户可以使用ADI公司提供的—个Windows®程序(CM3WSD.exe，—款开发工具)，通过PC串行端口COM1至COM31下载代码到基于Cortex-M3的ADuCxxx器件。但值得注意的是，无论主机为PC还是微控制器或DSP，只要主机遵守本应用笔记中详述的串行下载协议，就可以下载代码至基于Cortex-M3的ADuCxxx。

本应用笔记详细描述了基于Cortex-M3的ADuCxxx串行下载协议，以便最终用户能够理解该协议(嵌入式主机至嵌入式基于Cortex-M3的ADuCxxx)，并且能将该协议成功应用到目标系统中。

为明确起见，这里使用的术语“主机”(Host)指的是尝试向微转换器下载数据的宿主机(PC、基于Cortex-M3的ADuCxxx或DSP)。术语“加载器”(loader)指的是基于Cortex-M3的ADuCxxx内置的片内串行下载固件。

目录

简介.....	1	定义数据传输包格式.....	3
修订历史.....	2	命令.....	4
运行微转换器加载器.....	3	命令示例.....	5
物理接口.....	3	LFSR代码示例.....	6

修订历史

2013年1月—修订版0至修订版A

更改“简介”部分.....	1
---------------	---

2012年9月—修订版0：初始版

运行微转换器加载器

为了运行ADuCxxx器件上的加载器，必须通过一个电阻（通常为1 kΩ下拉电阻）拉低特定的GPIO引脚，并且复位器件，切换器件本身的RESET输入引脚可复位器件。其他复位（如看门狗复位、上电复位和软件复位）和特定GPIO下拉不会导致串行下载模式输入。参见器件用户指南了解串行下载模式的输入标准。

例如在ADuCM360上时，P2.2输入引脚在内核执行期间检查。若该引脚为低电平，且P2.2引脚检查时RSTSTA.EXTRST = 0x1，则器件进入串行下载模式。

物理接口

一旦触发，加载器就等待主机发送退格(BS = 0x08)字符进行同步。加载器测量此字符的时序，并相应地配置ADuCxxx UART串行端口用无奇偶性的8个数据位开始，以主机的波特率进行发送或接收。波特率必须在600 bps至115,200 bps之间(含本数)。

收到退格字符后，加载器即发送如下24字节ID数据包：

15字节 = 产品标识符

3字节 = 硬件和固件的版本号

4字节 = 保留，以备后用

2字节 = 换行和回车

ADuCxxx <space><space><space>128 <space>A3Y <space><space><space><space><ln><lr>

↑ ADuCxxx IS THE PRODUCT ID ↑ 128 CORRESPONDS TO THE MEMORY SIZE MODEL ↑ A3Y MEAN A SILICON REV. A AND A VERSION 3 LOADER. Y IS THE LOADER'S VERSION REVISION

图1. ID数据包示例

定义数据传输包格式

一旦UART配置完成，数据传输即可开始。表1给出了通用通信数据传输包格式。

数据包起始ID字段

第一个字段是数据包起始ID字段，它包括两个起始字符(0x07和0x0E)。这些字节为常数，用于加载器检测一个数据传输包的有效起始。

表1. 数据传输包格式

起始ID		字节数	命令	值			数据字节	校验和	
ID0	ID1		数据1	数据2	数据3	数据4	数据5	数据[x]	CS
0x07	0x0E	0x05至0xFF	E、W、V或R	MSB			LSB	0x00至0xFF	0x00至0xFF

字节数字段

接下来的字段是字节总数。字节最小数目是5，对应命令和值字段。字节最大数目是255：一个命令功能、4字节值和250字节的数据。

命令字段(数据1)

命令字段描述数据包的功能。允许使用4个命令功能中的一个。4个命令功能由4个ASCII字符之一表示：E、W、V或R。数据包命令功能如表2所示。

值字段(数据2至数据5)

值字段包含一个大顺序格式的32位值。

数据字节字段(数据6至数据255)

数据字节字段最多包含250个数据字节。

校验和字段

数据包校验和写入校验和字段。该二进制补码校验和是通过对字节数字段的十六进制值和数据1至数据255字段(以实际存在的数据字节计算)的十六进制值求和而算得的。校验和是该总和的二进制补码值。因此，从数据字节数到校验和的所有字节之和的LSB应当为0x00。这还可以通过数学方式表示为：

$$CS = 0x00 - \left(\text{Number of Bytes} + \sum_{N=1}^{255} \text{Data Byte}_N \right)$$

换言之，除起始ID外的所有字节的8位和必须等于0x00。

命令应答

加载器程序对每个数据包都会发出一个应答：否定应答BEL(0x07)或肯定应答ACK(0x06)。

如果接收到的校验和不正确或地址无效，加载器就会发送一个BEL信号。如果下载数据覆盖旧数据(没有擦除)，加载器不会给出警告。PC接口必须确保代码下载的所有位置都被擦除。

命令

表2所示为片内加载器上实施命令的完整列表。

擦除命令

擦除命令允许用户从值字段决定的特定起始页开始擦除Flash/EE。该命令还包括要擦除的页数。

如果地址为0x00000000，页数为0x00，加载器将认为是批量擦除命令，即会擦除整个用户代码空间。

擦除数据包命令如表3所示。

写命令

写入命令包括数据字节数(5 + x)、命令、要编程的第一个数据字节地址和要编程的数据字节。数据下载后就被编程到Flash/EE中。如果校验和不正确或者接收地址超出范围，加载器将发送一个BEL信号。如果主机从加载器接收到一个BEL信号，主机应中止下载过程，并重新开始整个下载过程。

验证命令

加载器需要两段信息来验证页面内容、页面最后4字节的内容，以及页面除最后4字节以外的24位LFSR(参见“LFSR代码示例”部分)。

表2. 数据包命令功能

命令功能	数据1字段中的命令字节	加载器肯定应答	加载器否定应答
擦除页	E (0x45)	ACK (0x06)	BEL (0x07)
写入	W (0x57)	ACK (0x06)	BEL (0x07)
验证	V (0x56)	ACK (0x06)	BEL (0x07)
远程复位	R (0x52)	ACK (0x06)	BEL (0x07)

表3. 擦除Flash/EE存储器命令

起始ID		字节数	命令	值				页面数	校验和
ID0	ID1		数据1	数据2	数据3	数据4	数据5	数据6	CS
0x07	0x0E	0x06	E (0x45)	0x00	ADR[23:16]	ADR[15:8]	ADR[7:0]	0x01至0xFF	0x00至0xFF

表4. Flash/EE存储器写入命令

起始ID		字节数	命令	值				数据字节	校验和
ID0	ID1		数据1	数据2	数据3	数据4	数据5	数据6	CS
0x07	0x0E	5 + x (0x06至0xFF)	W (0x57)	0x00	ADR[23:16]	ADR[15:8]	ADR[7:0]	0x00至0xFF	0x00至0xFF

验证页面必须遵循如下两个步骤：对每个需要验证的页面重复这两个步骤。

1. 发送值字段的值0x80000000以及数据字节字段的最后4字节。
2. 发送值字段的起始页面地址以及数据字节字段中的页面SIGN命令结果。

收到这两个数据包后，加载器会计算特定页面的LFSR并将其与提供的值进行比较。如果结果正确，并且该页面地址0x1FC的值与第1步指定的值匹配，就会返回ACK (0x06)，否则返回BEL (0x07)。

远程复位命令

一旦主机将所有数据包都发送到加载器，主机便可发送最后一个包以指示加载器执行复位。执行的是软件自复位。值字段应始终为0x1。

主机应确保执行串行编程的特定GPIO引脚在发出该命令前不会置位。当器件复位时，重新正常进入内核。加载器输入检查会再一次执行，这次特定GPIO引脚必须去置位。(内核不会修改RSTSTA，因此外部复位检查还是会检测到发生外部复位)。表7给出了一个远程复位的实例。

表5. 验证Flash/EE存储器命令，第1步

起始ID		字节数	命令	值				数据字节				校验和
ID0	ID1		数据1	数据2	数据3	数据4	数据5	数据6	数据7	数据8	数据9	CS
0x07	0x0E	0x09	V (0x56)	0x80	0x00	0x00	0x00	0x1FC 的数据	0x1FD 的数据	0x1FE 的数据	0x1FF 的数据	0x00至0xFF

表6. 验证Flash/EE存储器命令，第2步

起始ID		字节数	命令	值				数据字节				校验和
ID0	ID1		数据1	数据2	数据3	数据4	数据5	数据6	数据7	数据8	数据9	CS
0x07	0x0E	0x09	V (0x56)	0x00	ADR[2 3:16]	ADR[15: 8]	ADR[7:0]	LFSR[0:7]	LFSR[15:8]	LFSR [23:16]	0x00	0x00至0xFF

表7. 远程复位命令

起始ID		字节数	命令	值				校验和
ID0	ID1		数据1	数据2	数据3	数据4	数据5	CS
0x07	0x0E	0x05	R (0x52)	0x00	0x00	0x00	0x01	0xA8

命令示例

以下为使用端口分析仪获取数据的示例。

擦除命令

擦除0x00000200的1个页面，

IRP_MJ_WRITE 长度10: 07 0E 06 45 00 00 02 00 01 B2

IRP_MJ_READ 长度1: 06

批量擦除整个用户空间

IRP_MJ_WRITE 长度10: 07 0E 06 45 00 00 00 00 00 B5

IRP_MJ_READ 长度1: 06

写命令

从0x00000200开始写入16个数据字节，

IRP_MJ_WRITE 长度25: 07 0E 15 57 00 00 02 00 77 FF 2C B1 00 20 00 F0 5A FC 08 B1 01 20 00 E0 1F

IRP_MJ_READ 长度1: 06

验证命令

下一个验证命令在0x1FC的值指定为0x11223344

IRP_MJ_WRITE 长度13: 07 0E 09 56 80 00 00 00 44 33 22 11 77

IRP_MJ_READ 长度1: 06

验证0x00000200的页面，LFSR指定为0x00841B81，最终值将与0x11223344对比检查

IRP_MJ_WRITE 长度13: 07 0E 09 56 00 00 02 00 81 1B 84 00 7F

IRP_MJ_READ 长度1: 06

远程复位命令

IRP_MJ_WRITE 长度9: 07 0E 05 52 00 00 00 01 A8

IRP_MJ_READ 长度1: 06

AN-1160

LFSR代码示例

签名是一个多项式为 $x^{24} + x^{23} + x^6 + x^5 + x + 1$ 的24位CRC。初始值为0xFFFFFFFF。

```
long int GenerateChecksumCRC24_D32(unsigned long ulNumValues, unsigned long *pulData)
{
    unsigned long i, ulData, lfsr = 0xFFFFFFFF;

    for (i = 0x0; i < ulNumValues; i++)
    {
        ulData = pulData[i];
        lfsr = CRC24_D32(lfsr, ulData);
    }

    return lfsr;
}

static unsigned long CRC24_D32(const unsigned long old_CRC, const unsigned long Data)
{
    unsigned long D    [32];
    unsigned long C    [24];
    unsigned long NewCRC [24];
    unsigned long ulCRC24_D32;
    unsigned long int f, tmp;
    unsigned long int bit_mask = 0x000001;

    tmp = 0x000000;
    // Convert previous CRC value to binary.
    bit_mask = 0x000001;
    for (f = 0; f <= 23; f++)
    {
        C[f] = (old_CRC & bit_mask) >> f;
        bit_mask = bit_mask << 1;
    }

    // Convert data to binary.
    bit_mask = 0x000001;
    for (f = 0; f <= 31; f++)
    {
        D[f] = (Data & bit_mask) >> f;
        bit_mask = bit_mask << 1;
    }

    // Calculate new LFSR value.
    NewCRC[0] = D[31] ^ D[30] ^ D[29] ^ D[28] ^ D[27] ^ D[26] ^ D[25] ^
        D[24] ^ D[23] ^ D[17] ^ D[16] ^ D[15] ^ D[14] ^ D[13] ^
        D[12] ^ D[11] ^ D[10] ^ D[9] ^ D[8] ^ D[7] ^ D[6] ^
        D[5] ^ D[4] ^ D[3] ^ D[2] ^ D[1] ^ D[0] ^ C[0] ^ C[1] ^
        C[2] ^ C[3] ^ C[4] ^ C[5] ^ C[6] ^ C[7] ^ C[8] ^ C[9] ^
        C[15] ^ C[16] ^ C[17] ^ C[18] ^ C[19] ^ C[20] ^ C[21] ^
        C[22] ^ C[23];
    NewCRC[1] = D[23] ^ D[18] ^ D[0] ^ C[10] ^ C[15];
    NewCRC[2] = D[24] ^ D[19] ^ D[1] ^ C[11] ^ C[16];
    NewCRC[3] = D[25] ^ D[20] ^ D[2] ^ C[12] ^ C[17];
    NewCRC[4] = D[26] ^ D[21] ^ D[3] ^ C[13] ^ C[18];
    NewCRC[5] = D[31] ^ D[30] ^ D[29] ^ D[28] ^ D[26] ^ D[25] ^ D[24] ^
        D[23] ^ D[22] ^ D[17] ^ D[16] ^ D[15] ^ D[14] ^ D[13] ^
        D[12] ^ D[11] ^ D[10] ^ D[9] ^ D[8] ^ D[7] ^ D[6] ^
        D[5] ^ D[3] ^ D[2] ^ D[1] ^ D[0] ^ C[0] ^ C[1] ^ C[2] ^
        C[3] ^ C[4] ^ C[5] ^ C[6] ^ C[7] ^ C[8] ^ C[9] ^ C[14] ^
        C[15] ^ C[16] ^ C[17] ^ C[18] ^ C[20] ^ C[21] ^ C[22] ^
        C[23];
}
```

```

NewCRC[6] = D[28] ^ D[18] ^ D[5] ^ D[0] ^ C[10] ^ C[20];
NewCRC[7] = D[29] ^ D[19] ^ D[6] ^ D[1] ^ C[11] ^ C[21];
NewCRC[8] = D[30] ^ D[20] ^ D[7] ^ D[2] ^ C[12] ^ C[22];
NewCRC[9] = D[31] ^ D[21] ^ D[8] ^ D[3] ^ C[0] ^ C[13] ^ C[23];
NewCRC[10] = D[22] ^ D[9] ^ D[4] ^ C[1] ^ C[14];
NewCRC[11] = D[23] ^ D[10] ^ D[5] ^ C[2] ^ C[15];
NewCRC[12] = D[24] ^ D[11] ^ D[6] ^ C[3] ^ C[16];
NewCRC[13] = D[25] ^ D[12] ^ D[7] ^ C[4] ^ C[17];
NewCRC[14] = D[26] ^ D[13] ^ D[8] ^ C[0] ^ C[5] ^ C[18];
NewCRC[15] = D[27] ^ D[14] ^ D[9] ^ C[1] ^ C[6] ^ C[19];
NewCRC[16] = D[28] ^ D[15] ^ D[10] ^ C[2] ^ C[7] ^ C[20];
NewCRC[17] = D[29] ^ D[16] ^ D[11] ^ C[3] ^ C[8] ^ C[21];
NewCRC[18] = D[30] ^ D[17] ^ D[12] ^ C[4] ^ C[9] ^ C[22];
NewCRC[19] = D[31] ^ D[18] ^ D[13] ^ C[5] ^ C[10] ^ C[23];
NewCRC[20] = D[19] ^ D[14] ^ C[6] ^ C[11];
NewCRC[21] = D[20] ^ D[15] ^ C[7] ^ C[12];
NewCRC[22] = D[21] ^ D[16] ^ C[8] ^ C[13];
NewCRC[23] = D[31] ^ D[30] ^ D[29] ^ D[28] ^ D[27] ^ D[26] ^ D[25] ^
D[24] ^ D[23] ^ D[22] ^ D[16] ^ D[15] ^ D[14] ^ D[13] ^
D[12] ^ D[11] ^ D[10] ^ D[9] ^ D[8] ^ D[7] ^ D[6] ^
D[5] ^ D[4] ^ D[3] ^ D[2] ^ D[1] ^ D[0] ^ C[0] ^ C[1] ^
C[2] ^ C[3] ^ C[4] ^ C[5] ^ C[6] ^ C[7] ^ C[8] ^ C[14] ^
C[15] ^ C[16] ^ C[17] ^ C[18] ^ C[19] ^ C[20] ^ C[21] ^
C[22] ^ C[23];

```

```

ulCRC24_D32 = 0;
// LFSR value from binary to hex.
bit_mask = 0x000001;
for (f = 0; f <= 23; f++)
{
    ulCRC24_D32 = ulCRC24_D32 + NewCRC[f] * bit_mask;
    bit_mask = bit_mask << 1;
}
return(ulCRC24_D32 & 0x00FFFFFF);
}

```

AN-1160

注释