AN-1160
应用笔记

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com/cn

基于Cortex-M3的串行下载协议

简介

ADI公司Cortex-M3器件系列具有一个关键特性：可在电路运行状态下，将代码下载到片上FLASH/电子可擦除(EE)程序存储器。在电路运行状态下下载代码是通过器件通用异步接收发送器(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Cortex-M3器件的串行端口，就可以在现场通过串行下载特性执行系统升级。此串行下载特性意味着制造商可以在现场升级系统固件，而不必更换器件。

在上电时或者在任何复位之后，通过特定引脚配置，可以将任何Cortex-M3器件配置为串行下载模式。

参见具体器件的参考手册，了解串行下载模式的进入条件。例如，ADuCM360在内核执行期间会检查P2.2输入引脚。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。

在串行下载模式下，片上加载器程序会启动，从而配置器件的UART，并通过特定的串行下载协议与主机通信，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小端模式。

串行下载模式工作在器件的额电源范围内，无需特别高的编程电压，因为它是在片上产生的。

用户可以使用ADI公司提供的一款Windows®版开发工具（CM3WSD.exe），通过PC串行端口COM1至COM31向Cortex-M3器件下载代码。但值得注意的是，无论主机是PC还是微控制器还是数字信号处理器(DSP)，只要主机遵守本应用笔记中详述的串行下载协议，就可以下载代码到基于Cortex-M3的器件。

本应用笔记详细描述了Cortex-M3器件的串行下载协议，以便最终用户能够理解该协议，并且能将该协议应用到目标系统中。

为明确起见，这里使用的术语“主机”(Host)指的是尝试向基于Cortex-M3的器件下载数据的主机(PC、微控制器或DSP)。术语“加载器”(loader)指的是Cortex-M3器件内置的片上串行下载固件。

ADI公司基于Cortex-M3的器件系列包括ADuCM310、ADuCM320/ADuCM322、ADuCM360/ADuCM361/ADuCM362/ADuCM363和ADuCRF101。

目录

修订历史

2017年11月—修订版A至修订版B

更改应用笔记标题和“简介”部分 1
更改“运行微转换器加载器”部分、“物理接口”部分、图 1、“校验和字段”部分和表 1 3
更改“ADuCM360/ADuCM361 和 ADuCRF101 的验证命令”部分、表 3、表 4、表 5 和表 6 4
增加“ADuCM362/ADuCM363、ADuCM320/ADuCM322 和 ADuCM310 的验证命令”部分、“循环冗余校验(CRC)计算”部分、表 7、表 8 和表 9；重新排序 5
更改“远程复位命令”和表 10 5
更改“擦除命令示例”部分、“写入命令示例”部分、“ADuCM360/ADuCM361 和 ADuCRF101 的验证命令示例”部分、“ADuCM362/ADuCM363、ADuCM320/ADuCM322 和 ADuCM310 的验证命令示例”部分和“远程复位命令示例”部分 6

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Cortex-M3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，并对器件进行复位。翻转器件RESET输入引脚的电平，即可复位器件。其他复位（如看门狗复位、上电复位和软件复位）与特定GPIO下拉操作，会导致除ADuCRF101外的所有基于Cortex-M3的器件进入串行下载模式。参见器件硬件参考手册，了解串行下载模式的进入条件。

例如，ADuCM360在内核执行期间会检查P2.2输入引脚。若该引脚为低电平，且在检查P2.2输入引脚时RSTSTA寄存器 ≠ 0x00，器件可进入串行下载模式。

物理接口

触发后，加载器等待主机发送退格(BS = 0x08)字符进行同步。加载器会测量此字符的时序，并相应地配置器件的UART串行端口，即无奇偶校验、8个数据位、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，加载器立即发送如下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 = 保留字段，以备后用
- 2字节 = 换行和回车

ID数据包示例参见图1。

定义数据传输包格式

UART配置完成后，数据传输即可开始。表1给出了通用通信数据的传输包格式。

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字段。

字节数字段

接下来的字段是字节数字段。字节最小数目是5，对应命令字段和值字段。允许的最大字节数目是255:1字节的命令字段、4字节的值字段和250字节的数据字段。

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个命令功能分别用4个ASCII字符表示：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）

值字段包含一个大端模式的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

校验和字段用于数据包校验。该字段对应的二进制补码校验和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段计算）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，从字节数到校验和，所有字节之和的LSB均为0x00。可以通过数学方式表示为：

$$CS = 0x00 - (Number\ of\ Bytes + \sum_{N-1}^{255} Data\ Byte_N)$$

其中，*CS*为校验和。

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响应：否定响应BEL (0x07)或肯定响应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。如果下载的新数据覆盖了没被擦除的旧数据，则加载器不会提供警告。PC接口必须确保代码下载的所有位置都被擦除。

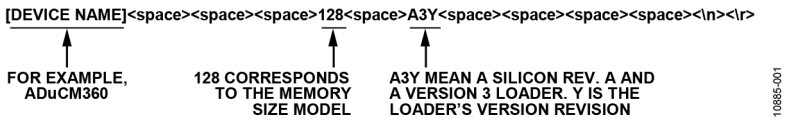

图1. ID数据包示例

表1. 数据传输包格式

起始ID			命令	值				数据字节	校验和
ID0	ID1	字节数	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	[N/A][1]	[N/A][1]	LSB	0x00至0xFF	0x00至0xFF

[1] N/A表示不适用。

命令

表2所示为片上加载器上实现的命令的完整列表。

擦除命令

擦除命令允许用户从值字段决定的特定起始页地址开始擦除Flash/EE存储器。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，即会擦除整个用户的代码空间。

擦除数据包命令如表3所示。

写入命令

写入命令包括数据字节数(5 + x)、命令、要编程的第一个数据的字节地址和要编程的数据字节。这些字节被写入Flash/EE存储器中，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，主机必须中止下载过程，并重新开始整个下载过程。

ADuCM360/ADuCM361和ADuCRF101的验证命令

加载器需要两段信息来验证页面内容：页面最后的 4 字节的内容和页面除最后 4 字节以外的 24 位线性反馈移位寄存器 (LFSR)（参见“LFSR 代码示例”部分）。

要验证页面，必须遵循如下两个步骤。对每个需要验证的页面，都需要重复这两个步骤。

1. 在值字段中发送值 0x80000000，在数据字节字段中发送页面的最后 4 字节。
2. 在值字段中发送起始页面地址，在数据字节字段中发送页面的 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)。

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			命令	值				页面数	校验和
ID0	ID1	字节数	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. 写入Flash/EE存储器命令

起始ID			命令	值				数据字节	校验和
ID0	ID1	字节数	数据1	数据2	数据3	数据4	数据5	数据[x]	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

表5. ADuCM360/ADuCM361和ADuCRF101的Flash/EE存储器验证命令，第一步

起始ID			命令	值				数据字节				校验和
ID0	ID1	字节数	数据1	数据2	数据3	数据4	数据5	数据6 (LSB)	数据7	数据8	数据9 (MSB)	CS
0x07	0x0E	0x09	V (0x56)	0x80	0x00	0x00	0x00	页面的最后四个字节				0x00至0xFF

表6. ADuCM360/ADuCM361和ADuCRF101的Flash/EE存储器验证命令，第二步

起始ID			命令	值				数据字节				校验和
ID0	ID1	字节数	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[23:16]	ADR[15:8]	ADR[7:0]	LFSR[0:7]	LFSR[15:8]	LFSR[23:16]	0x00	x00至0xFF

ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310的验证命令

加载器需要两段信息来验证页面内容：页面最后8字节的内容和页面除最后8字节外的32位前向签名。

要验证页面，必须遵循如下三个步骤。对每个需要验证的页面，都需要重复这三个步骤。

1. 在值字段中发送值0x80000000，在数据字节字段中发送页面的最后一个字。
2. 在值字段中发送值0x90000000，在数据字节字段中发送页面的倒数第二个字。
3. 在值字段中发送起始页面地址，在数据字节字段中发送页面的前向签名。

接收到这三个数据包后，加载器使用闪存控制器FSIGN命令来计算指定页面的前向签名，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

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

主机须确保用于启动串行编程的特定GPIO引脚在发出该命令前不会再被置为有效。因为在器件复位重新正常进入内核后，加载器入口检查会再一次执行。因此，这一次特殊的GPIO引脚必须被置为无效。（内核不会修改RSTSTA寄存器，因此外部复位检查还是会检测到发生了外部复位。）表10给出了一个远程复位的实例。

循环冗余校验(CRC)计算

对于ADuCM360/ADuCM361和ADuCRF101，签名是一个24位CRC，使用如下多项式：

$$x^{24} + x^{23} + x^6 + x^5 + x + 1$$

初始值为0xFFFFFFFF。

对于ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310，签名是一个32位CRC，使用如下多项式：

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

初始值为0xFFFFFFFF。

表7. ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310的Flash/EE存储器验证命令，第一步

起始ID			命令	值				数据字节				校验和
ID0	ID1	字节数	数据1	数据2	数据3	数据4	数据5	数据6 (LSB)	数据7	数据8	数据9 (MSB)	CS
0x07	0x0E	0x09	V (0x56)	0x80	0x00	0x00	0x00	页面的最后四个字节				0x00至0xFF

表8. ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310的Flash/EE存储器验证命令，第二步

起始ID			命令	值				数据字节				校验和
ID0	ID1	字节数	数据1	数据2	数据3	数据4	数据5	数据6 (LSB)	数据7	数据8	数据9 (MSB)	CS
0x07	0x0E	0x09	V (0x56)	0x90	0x00	0x00	0x00	页面的倒数第二个四字节				0x00至0xFF

表9. ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310的Flash/EE存储器验证命令，第三步

起始ID			命令	值				数据字节				校验和
ID0	ID1	字节数	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)	ADR [32:24]	ADR [23:16]	ADR [15:8]	ADR [7:0]	LFSR [0:7]	LFSR [15:8]	LFSR [23:16]	LFSR [32:24]	0x00至0xFF

表10. 远程复位命令

起始ID			命令	值				校验和
ID0	ID1	字节数	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抓取数据的示例。

擦除命令示例

要擦除地址0x00000200处的1页，请发出以下命令：

```
IRP_MJ_WRITE   Length 10: 07 0E 06 45 00 00 02 00 01 B2
IRP_MJ_READ    Length 1 : 06
```

要批量擦除整个用户空间，请发出以下命令：

```
IRP_MJ_WRITE   Length 10: 07 0E 06 45 00 00 00 00 00 B5
IRP_MJ_READ    Length 1: 06
```

写入命令示例

要从0x00000200开始写入16个数据字节，请发出以下命令：

```
IRP_MJ_WRITE   Length 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    Length 1 : 06
```

ADuCM360/ADuCM361和ADuCRF101的验证命令示例

如果下一个验证命令在0x1FC处的值指定为0x11223344，请发出以下命令：

```
IRP_MJ_WRITE   Length 13: 07 0E 09 56 80 00 00 00 44 33 22 11 77
IRP_MJ_READ    Length 1 : 06
```

如果验证地址0x00000200处的页面，并且LFSR指定为0x00841B81，则最后的值将与0x11223344对比检查。请发出以下命令：

```
IRP_MJ_WRITE   Length 13: 07 0E 09 56 00 00 02 00 81 1B 84 00 7F
IRP_MJ_READ    Length 1 : 06
```

ADuCM362/ADuCM363、ADuCM320/ADuCM322和ADuCM310的验证命令示例

如果下一个验证命令在0x7FC处的值指定为0x11223344，请发出以下命令：

```
IRP_MJ_WRITE   Length 13: 07 0E 09 56 80 00 00 00 44 33 22 11 77
IRP_MJ_READ    Length 1 : 06
```

如果下一个验证命令在0x7F8处的值指定为0x55667788，请发出以下命令：

```
IRP_MJ_WRITE   Length 13: 07 0E 09 56 90 00 00 00 88 77 66 55 57
IRP_MJ_READ    Length 1 : 06
```

如果验证地址0x00000800处的页面，并且LFSR指定为0x00841B81，则最后的值将与0x11223344和0x55667788对比检查。请发出以下命令：

```
IRP_MJ_WRITE   Length 13: 07 0E 09 56 00 00 08 00 81 1B 84 00 79
IRP_MJ_READ    Length 1 : 06
```

远程复位命令示例

要对加载器执行复位，请发出以下命令：

```
IRP_MJ_WRITE   Length 9: 07 0E 05 52 00 00 00 01 A8
IRP_MJ_READ    Length 1: 06
```

LFSR代码示例

签名是一个24位CRC，使用多项式$x^{24} + x^{23} + x^6 + x^5 + x + 1$。初始值为0xFFFFFF。

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

   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);
}
```