ANALOG DEVICES

AN-2058
APPLICATION NOTE

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

ADuCM355 User Bootloader

INTRODUCTION

The flash of an erased ADuCM355 must initially be programmed with a user application via serial wire due to the absence of a kernel bootloader.

A user application can implement its own bootloader to provide a mechanism for in field self updating. Implementing its own user bootloader requires the user application to be structured in an appropriate way to be suitable for the user bootloader.

This application note describes one approach, described hereafter as a user bootloader, where the user bootloader can be achieved by partitioning the user flash into two separate regions and implementing a bootloader that is compatible with the CrossCore® serial flash programmer in one region.

TABLE OF CONTENTS

REVISION HISTORY

10/2020—Revision 0: Initial Version

ADUMC355 BACKGROUND

The following features of the ADuCM355 are utilized for the user bootloader implementation.

The ADuCM355 has 128 kB of user flash that is partitioned into separate 8 kB blocks for the purposes of erasing and write protection.

The on-chip kernel executes immediately after a reset to

- Boot the device with manufacturer data.
- Assess the user flash meta data to determine whether to switch over to running the user application or remain within the on-chip kernel.
- Assess the user flash meta data to determine whether to write protect flash blocks within the user flash space. If the user application is valid, it applies block write protection and exits to the user application. If the user application is not valid or the BM/P1.1 pin is asserted, it does not apply block write protection and remains within the on-chip kernel (see Figure 7 for a diagram of kernel execution).

Remaining within the on-chip kernel has the following advantages:

- Recovery. User code is prevented from running and performing operations that may prevent desired access to the device.
- Avoiding write protection. A write protected block cannot be erased, even by a mass erase. The write protection applied to the user flash, indirectly via the user flash metadata or directly via the user application execution, is therefore avoided. A user flash mass erase is then possible because the code execution remains in the kernel.

USER BOOTLOADER IMPLEMENTATION

The user bootloader is placed in the first 8 kB of user flash. The remaining 120 kB is available for user application development (see Figure 1).

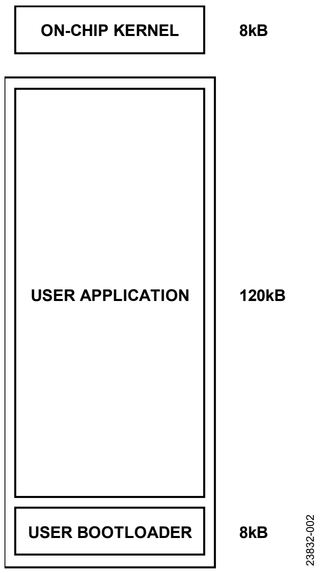


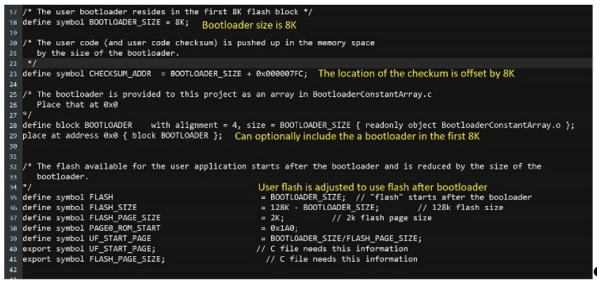
Figure 1. Memory Layout

The user bootloader is a standalone application residing in the first 8 kB of user flash. The user application must be built to execute starting from 0x2000 (that is, 8 kB), which requires minor modifications to a standard application.

The modifications require the use of a custom linker configuration file to place the user application in the appropriate area in the user flash.

Loading the custom linker file can be achieved from within the IAR Embedded Workbench® environment by accessing the **Linker** > **Config** tab, as shown in Figure 5. See the Conversion Steps section for the full procedure.

Comparing the .icf linker script file with the standard one provided on GitHub for the ADuCM355 highlights several differences (see Figure 2).



```
/* The user bootloader resides in the first 8K flash block */
define symbol BOOTLOADER_SIZE = 8K;

/* The user code (and user code checksum) is pushed up in the memory space
   by the size of the bootloader.
 */
define symbol CHECKSUM_ADDR  = BOOTLOADER_SIZE + 0x000007FC;

/* The bootloader is provided to this project as an array in BootloaderConstantArray.c
   Place that at 0x0
*/
define block BOOTLOADER    with alignment = 4, size = BOOTLOADER_SIZE { readonly object BootloaderConstantArray.o };
place at address 0x0 { block BOOTLOADER };


/* The flash available for the user application starts after the bootloader and is reduced by the size of the
   bootloader.
*/
define symbol FLASH                                  = BOOTLOADER_SIZE;  // "flash" starts after the booloader
define symbol FLASH_SIZE                             = 128K - BOOTLOADER_SIZE;        // 128k flash size
define symbol FLASH_PAGE_SIZE                        = 2K;          // 2k flash page size
define symbol PAGE0_ROM_START                        = 0x1A0;
define symbol UF_START_PAGE                          = BOOTLOADER_SIZE/FLASH_PAGE_SIZE;
export symbol UF_START_PAGE;                        // C file needs this information
export symbol FLASH_PAGE_SIZE;                        // C file needs this information

```
Bootloader size is 8K
The location of the checkum is offset by 8K
Can optionally include the a bootloader in the first 8K
User flash is adjusted to use flash after bootloader
23832-003

Figure 2. Linker Configuration File Modifications

BOOTLOADER PLACEMENT

The user bootloader resides in the first 8 kB of the user flash. The user bootloader is built like any other ADuCM355 application with an exception vector table, checksum placement, and user flash metadata.

USER APPLICATION PLACEMENT

The user application is placed starting at 0x2000, and the amount of available application space is reduced by 0x2000. The Cortex®-M3 interrupt vector table is placed at 0x2000. The user bootloader updates the Cortex-M3 vector table offset register (VTOR) to match this placement.

The user flash metadata is offset by 0x2000.

USER BOOTLOADER METADATA

The user bootloader metadata is checked and acted upon by the on-chip kernel. The on-chip kernel ensures the validity of the user bootloader area before transferring control and applying any user flash write protection to the user bootloader region of the user flash.

USER APPLICATION METADATA

The user application metadata is checked and acted upon by the user bootloader. The user bootloader ensures the validity of the user application area before transferring control to it and applying any user flash write protection to the user application region of the user flash.

The user bootloader performs the same types of checks and operations on the user application metadata that the on-chip kernel performs on the user bootloader metadata (or standard application metadata) with changes only to the addresses and ranges used. A standard application can then be easily converted for operation with the user bootloader by offsetting it by 0x2000.

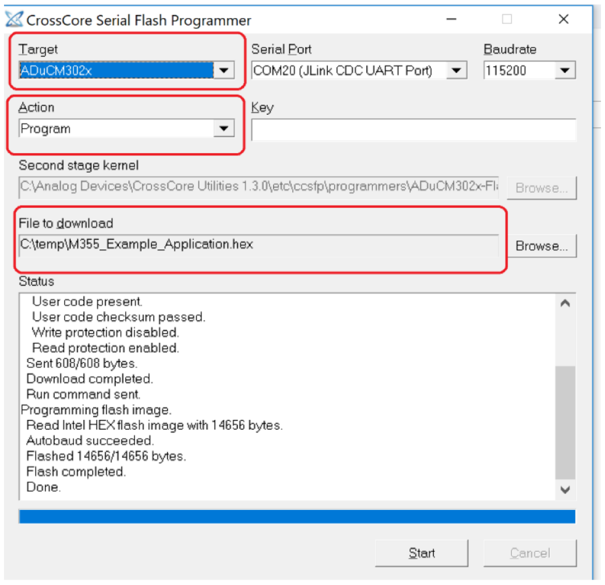
DESKTOP APPLICATION

The protocol is compatible with the Analog Devices, Inc., **CrossCore Serial Flash Programmer** tool, which can be downloaded at www.analog.com/crosscore-utilities.

The **CrossCore Serial Flash Programmer** tool supports several different protocol variants. This implementation of the user bootloader supports the version implemented by the ADuCM3027 and ADuCM3029. Select the **ADuCM302x** option under **Target**, as shown in Figure 3.

The supported options for **Action** are **Program** and **Erase**.

Click the **Browse** button to load the user application Intel hexadecimal file from **File to download**, and click the **Start** button.



Target: ADuCM302x
Serial Port: COM20 (JLink CDC UART Port)
Baudrate: 115200
Action: Program
Key:
Second stage kernel: C:\Analog Devices\CrossCore Utilities 1.3.0\etc\ccsfp\programmers\ADuCM302x-Fl
File to download: C:\temp\M355_Example_Application.hex
Status

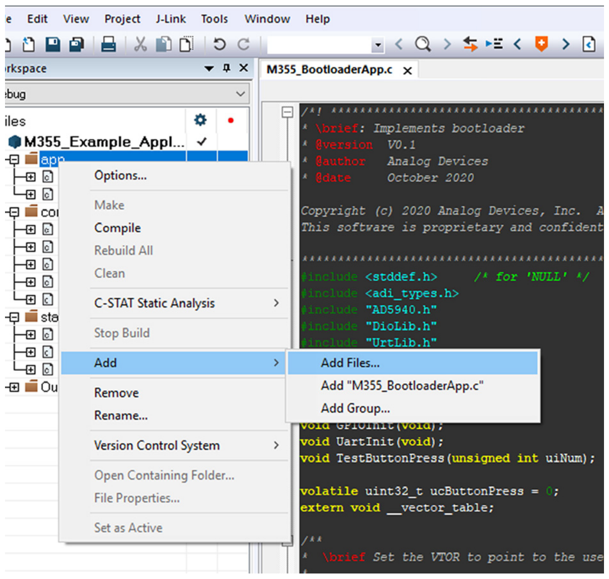
```
   User code present.
   User code checksum passed.
   Write protection disabled.
   Read protection enabled.
 Sent 608/608 bytes.
 Download completed.
 Run command sent.
Programming flash image.
 Read Intel HEX flash image with 14656 bytes.
 Autobaud succeeded.
 Flashed 14656/14656 bytes.
 Flash completed.
 Done.
```

*Figure 3. **CrossCore Serial Flash Programmer** Window*

CONVERSION STEPS

The following steps are required to convert an existing application for use with the user bootloader or user application model.

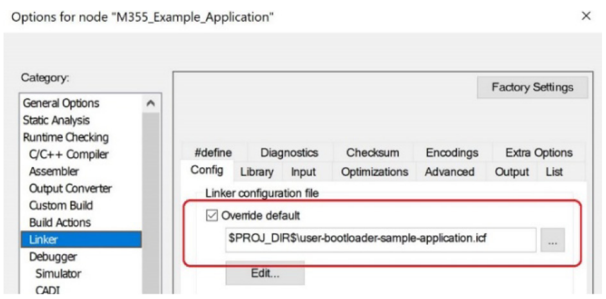
1. Download or clone the ADuCM355 software development kit (SDK) files from GitHub. Search for the ADuCM355 on GitHub to find the **aducm355-examples** SDK files.
2. Within the SDK, navigate to **examples** > **DigitalDie** > **M355_Bootloader**. Copy and paste the **BootloaderConstantArray.c** file and, from the **iar** folder, the **user-bootloader-sample-application.icf** file to the active project directory where the bootloader is to be implemented.
3. Open the desired project in the IAR Embedded Workbench and add the **BootloaderConstantArray.c** file to the project by right clicking the application folder in the project explorer. Then, go to **Add** > **Add Files**, and select the **BootloaderConstantArray.c** file. The user bootloader is provided as a C file, which contains a C array of instruction codes to implement the user bootloader. A custom linker configuration file is provided to ensure that this array is placed correctly at Location 0x0000 0000.



```
/*! ************************************
 * \brief: Implements bootloader
 * @version  V0.1
 * @author   Analog Devices
 * @date     October 2020

Copyright (c) 2020 Analog Devices, Inc.  A
This software is proprietary and confident

***************************************
#include <stddef.h>     /* for 'NULL' */
#include <adi_types.h>
#include "AD5940.h"
#include "DioLib.h"
#include "UrtLib.h"

void GPIOInit(void);
void UartInit(void);
void TestButtonPress(unsigned int uiNum);

volatile uint32_t ucButtonPress = 0;
extern void __vector_table;

/**
 *  \brief Set the VTOR to point to the use
```

Figure 4. Adding the Bootloader

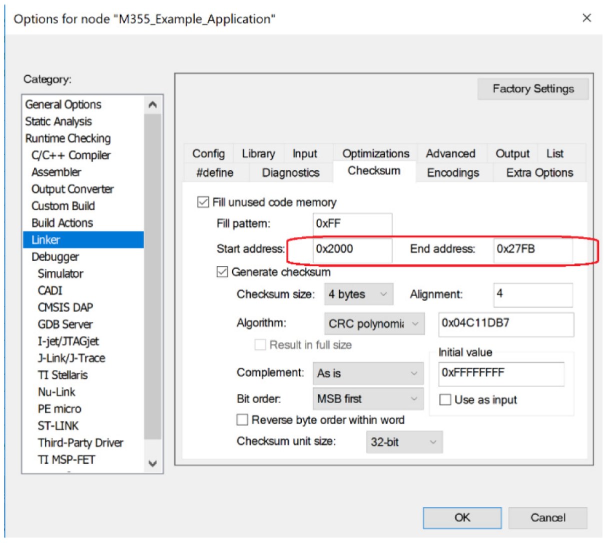
4. Select the custom linker configuration file from within the IAR Embedded Workbench by accessing the **Linker** > **Config** tab as shown in Figure 5. Select the **Override default** box, and browse for the **user-bootloader-sample-application.icf** file.
   This changed linker configuration file ensures that the user bootloader and the user application are placed correctly according to the user bootloader or user application model.



Linker configuration file
☑ Override default
$PROJ_DIR$\user-bootloader-sample-application.icf

Figure 5. Specify Custom Linker Configuration File

5. The range for the calculation of the checksum for the user application must be adjusted for the changed memory layout. To adjust the checksum calculation within the IAR Embedded Workbench, go to **Linker** > **Checksum** tab as shown in Figure 6. Enter **0x2000** in the **Start address** box and **0x27FB** in the **End address** box.
   In the standard application, this calculation is used by the on-chip kernel to check the integrity of an application in the user flash.
   In the user bootloader or user application model, the on-chip kernel is checking the integrity of the user bootloader before switching to it, and the user bootloader is checking the integrity of the user application before switching to it.



☑ Fill unused code memory
Fill pattern: 0xFF
Start address: 0x2000 End address: 0x27FB
☑ Generate checksum
Checksum size: 4 bytes Alignment: 4
Algorithm: CRC polynomia 0x04C11DB7
☐ Result in full size
Initial value
0xFFFFFFFF
☐ Use as input
Complement: As is
Bit order: MSB first
☐ Reverse byte order within word
Checksum unit size: 32-bit

Figure 6. Checksum Calculation

6. Modify the **startup_ADuCM355.c** file.
   a. Add the following line of code:
      ```
      /* Linker provided symbols */
      extern uint32_t FINAL_CRC_PAGE;
      ```
   b. Search for **NumOfCRCPages**, and replace the following line:
      ```
      __root const uint32_t
      NumOfCRCPages=0;
      ```
      with the following line:
      ```
      __root const uint32_t
      NumOfCRCPages=(uint32_t)&FINAL_CR
      C_PAGE;
      ```
7. The final step is to add a function to the main program.
   ```
   void __iar_init_core (void)
   {
     SCB->VTOR = (uint32_t)
   &__vector_table;
   }
   ```
   The purpose of this function is to point the VTOR to the user firmware exception table. The bootloader is already performing this step, but in debug mode, the IAR Embedded Workbench bypasses running the bootloader. The IAR Embedded Workbench debugger performs a Type 0 reset (hardware reset), and then sets the PC to what it considers to be the reset vector. Therefore, for interrupts to work in the debug mode, this function is needed.

This application can now be downloaded and debugged as normal.

BOOT MODE PINS

The on-chip kernel and the user bootloader have different boot mode pins that alter their respective execution flows.

The on-chip kernel boot mode pin has a higher priority than the user bootloader boot mode pin.

ON-CHIP KERNEL BM/P1.1 PIN

BM/P1.1 bypasses the execution of all software in the user flash.

The BM/P1.1 pin is connected to Switch S3 on the EVAL-ADucM355QSPZ evaluation board. Pressing and holding Switch S3 while subsequently performing a reset (via press and release of Switch S1) places the on-chip kernel in boot mode.

USER BOOTLOADER BOOT MODE P1.0/SYS_WAKE PIN

The P1.0/SYS_WAKE pin bypasses the execution of a user application that may be present in the user flash. If asserted, the user bootloader bypasses the check for the presence of a valid user application and immediately enters download mode.

The P1.0/SYS_WAKE pin is connected to Switch S2 on the EVAL-ADucM355QSPZ evaluation board. Pressing and holding Switch S2 while subsequently performing a reset (via press and release of Switch S1) places the user bootloader in boot mode.

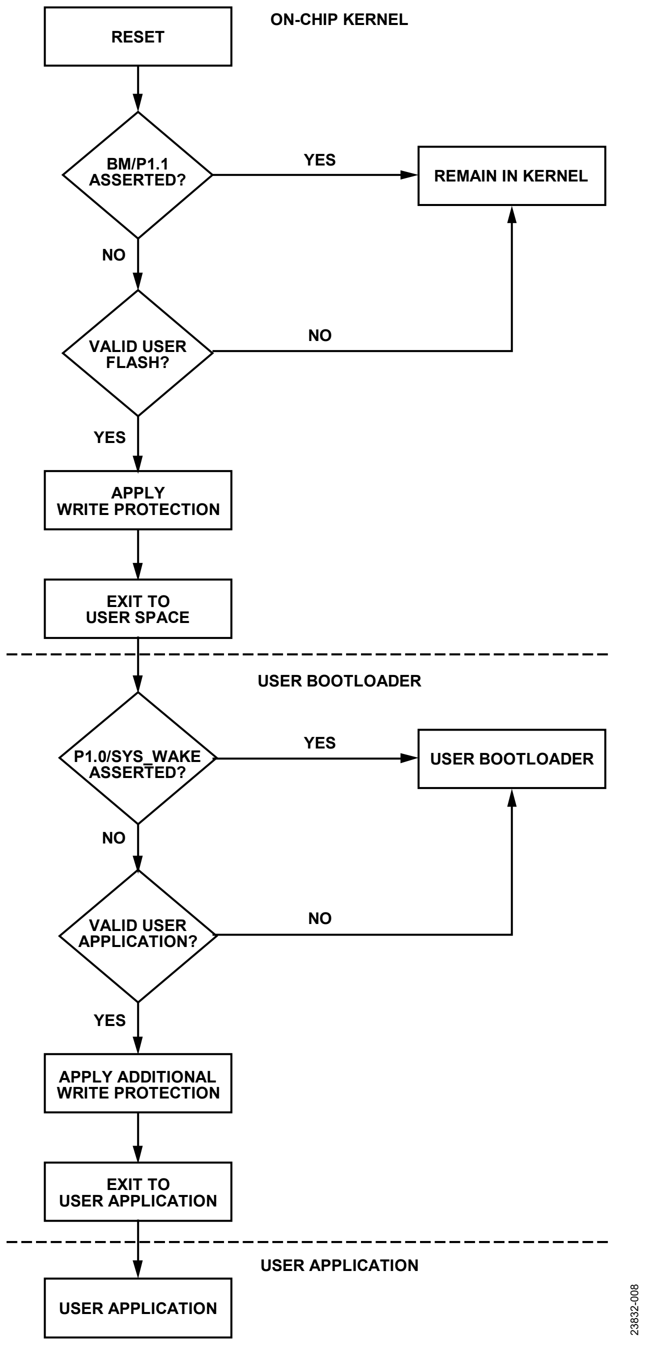


Figure 7. Bootloader Flow Diagram