Keith & Koep Bootloader

(PBL, TFTP/BOOTP Loader)

Documentation 2.02

1.0 Introduction

The Trizeps and Arnold products are shipped as default with a firmware, which is able to maintain the hardware across the serial port SD/MMC and ethernet.

This software comes with following features:

- small development turnaround times
- OS powermanagement support
- bootp,tftp protocoll for ethernet
- supports dm9000, smsc Lan91C9x onboard ethernet, NE2000 Compact Flash and Ositech 4 of Diamonds PCMCIA cards
- supports download from SD/MMC cards (autoboot.000 ... autoboot.999)
- serial download through a proprietary format
- flash support for different Intel Flash types
  supports buffered flash access for higher perfomance
- memory size detection
- on board self test support
- extendable command line interpreter
- splash screen during boot
- decompression support
- bootloader may be updated by the bootloader

The bootstrap process will be interrupted, when more than one ESC character is seen on COM1: during the initial bootphase. (FF Uart) @38Kb 8,n,1,noflow.

Latest versions are able to do automatic updates from SD/MMC cards if an „AUTOBOOT.001“ file is found.

2.0 Getting started

2.1 Boot or Update via SD/MMC (TRIZEPS2 and later)

The bootloader is able to boot from a SD/MMC having a FAT filesystem. During system startup it looks wether there is an SD/MMC card present. If there is a card containing an „AUTOBOOT.000“ file with a valid boot infromation header. This file will be booted. One flag inside this header decides, wether the bootloader shall continue booting using the next file (AUTOBOOT.001.... AUTOBOOT.999).

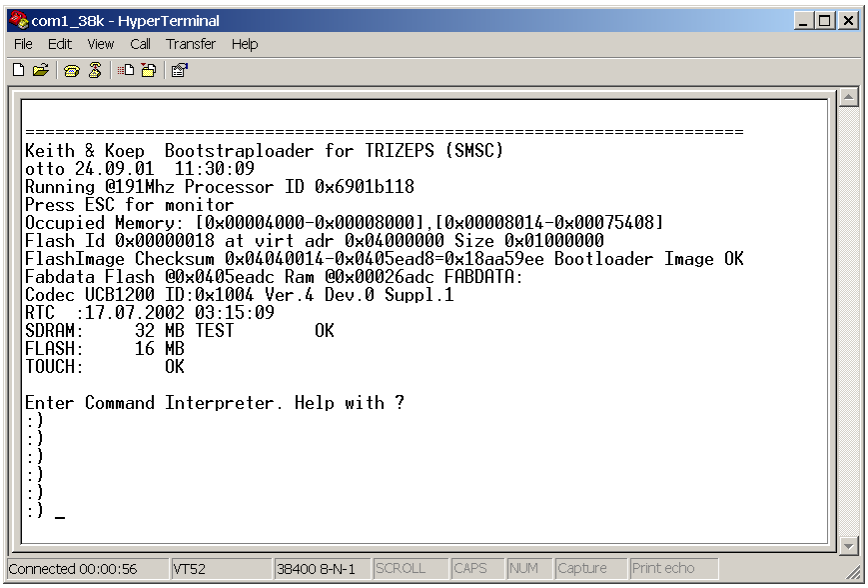
The content of the AUTOBOOT.XXX file header describes what will be done with that file. The header is described There are following options possible

1. Copy File to Memory and execute it.
2. Copy File to Memory and save this to flash. On the next bootphase without SDMMC this file will be copied back to SDRAM and it will be executed
3. Show a Bitmap
4. Show a Bitmap and make it resident for the next start
5. Erase Registry
6. Erase Flashdisk
7. Write file to direct flash address. Using this you are able to

- Update Bootloader
- Update Flashdisk
- Update Registry
- Write OS Image (add. Boothdr required!)

2.2 How to connect the board to a host system

1. Use an RS232 null-modem cable to attach the serial interface of the board to a RS232 port on a terminal or terminal emulator. For example, you could connect it to a PC running Windows and use the Windows Terminal or Hyperterminal application. Configure the terminal to operate at 38 kbaud, 8-bit data, 1 stop bit, no parity, no flow control. If you need more details on choosing an appropriate cable, refer to appendix A of MT6N.
2. Having opened the Hyperterm, make sure, that you have the input focus on the terminal emulator and then push the ESC key on your PC until the autorepeat function starts to send continuously ESC characters to the target.
3. Power On your target board

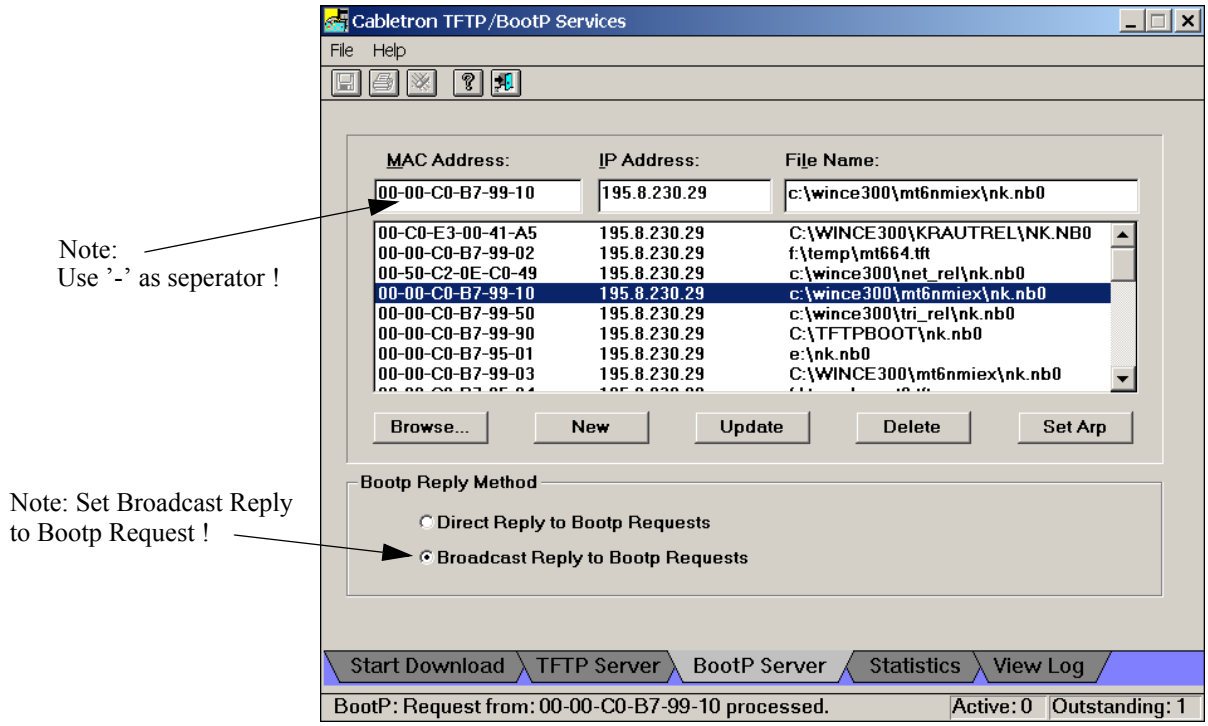


4. Watch the output of the bootloader it will give you some bootloader information and then a command line prompt.
5. Type ? <CR> to get a summary of all known commandsInstalling the host components for bootloading via Ethernet

2.3 Installing the host components for bootloading

The target firmware supports download via bootp/tftp. Usually those services are not available on Microsoft Windows operating systems. In this case you can use the free bootp/tftp server programm from Cabletron.

1. Unpack ther boottft2.zip archive and install this to your PC. The standard installation path is c:\tftpboot.
2. Open the Cabletron program and then select BootP Server.
3. Enter *mac <CR> <ESC>* at your hyperterm window to see the mac address of the board.
4. Fill in this mac address into the upper left field of the cabletron server program. (Note: Cabletron wants '-' as a seperator !)
5. Assign a free IP address for your target board into the 2nd field.
6. Enter the boot file name and path to the upper right field. (Attention: DOS 8.3 filenames!)



7. Press update to update your target list.
8. Select '*Broadcast Reply to Bootp Request*' !

2.4 Starting a bootp/tftp session

Make sure that the ethernet of your board is connected to your pc network. You can either connect directly with a crossed cable, but you should prefer using a hub and a standard network cable.

From the bootloader command prompt enter *tftp*

Now the bootloader issues a bootp request as a broadcast to the ethernet. The request will be answered by the bootp server. The answer contains those parameters needed for booting:

1. Target IP address
2. Netmask
3. Gateway IP address
4. Server IP address
5. Bootfilename

Having the bootp information the bootloader output shows this information to the serial line and starts a tftp session to the given Server with the given filename.

The first 512 Bytes of the image file contain information like the load address and flags telling the bootloader what to do with the file.

Having the transport finished, the image, which is downloaded is either started by the bootloader or it is burned to flash. The burning algorithm only touches the modified flash sectors and it will do an erase cycle only if needed. During the flashing process you will see a progressbar with characters walking from left to the right. One for each sector:

- 'E' for erase
- 'B' for burn
- '*' modification done successfully
- '-' no modification needed for this sector

3.0 How to build a Header for your Bootfile

The bootfile which is loaded carries bootloader information in the first block. The header is described in „Genboot Header“ at Page 13. of this document. This information block describes where to load the image file in dram and wether is will be burned to flash after download. A file without this header will not be fully transferred to target as the bootloader does not know what to do with it.

There is one utility called genboot (same as buildboot.exe) which may be used to generate those 512 Byte headers, which may be pasted before the binary image. Keith & Koep provides the sourcecode of genboot and some scripts to use this tool to support the usual OS development tools like Linux, VxWorks, OS9, and Windows CE.

3.1 How to use Buildboot.exe

Buildboot has two operation modes. The first mode is able to patch a given file with bootheader information (WINCE option). Using this mode, you have to be sure, that the given image has build a placeholder for this information, or you have to be sure, that the first 512 Bytes of the given image file are not used.

You can use buildboot in those 2 ways:

1. *gen_boot WINCE filename [-d[n]] [-w] exec_adr loadadr*
2. gen_boot [-d[n]][-w][-r][-x filename] exec_adr nseg base1 len1 [base len] > File

The second mode is obtained without the WINCE option. Using this mode, the utility ejects 512 Bytes to the standard output. This output may be redirected to a file, which might be pasted with cat or copy to the start of the image file.

The -d[n] option defines the download type of the transfer.

TABLE 1.

d opt.	tftp	sd/mmc	Download Type
-d0	*	*	(default) Transfer file to SDRAM and start it after download.
-d2	*	*	Load file to SDRAM and start a gdb session afterwards (use a udp-based gdb protocol which is quite similar to the remote serial target) This feature is only usable with seperate support for the debugging of standalone programs.
-d3	*	*	Load file to SDRAM and copy this to the flash image area later. After the next power-up the bootloader may copy the flash content back to SDRAM and start it as the default program or OS.

TABLE 1.

d opt.	tftp	sd/mmc	Download Type
-d4	*		Used to burn the given file to sector 0 of the flash. **Note:** If the content of the file is wrong or if you power down the target during the flashing process the target is lost until the boot-loader is restored by a JTAG download process.
-d6		*	(sdmmc only) Show Bitmap File. The Fileformat is shown in
-d7	*	*	Show and Bitmap File and burn this file to Flash
-d8	*	*	Download binary file to specified flash address
Mask			**Add Mask Below to add additional funktionality**
+0x20		*	Add 0x20 to arg to Proceed with next file. Do not enter command shell. For bitmaps: Do not ask to confirm flashing of bitmap.
+0x40		*	Add 0x40 to arg for erasing the IPSM Flashdisk during update
+0x80		*	Add 0x80 to arg for erasing the persistant registry of wince

3.1.1 Example 1

buildboot.exe WINCE imagefile -r 0xA0801000 0xA0080000

Patch image file (using WINDOWS CE: usually nk.nb0) with the bootheader information. Load image to SDRAM Address 0xA0080000 and start execution at 0xA0801000 after download. The filesize is recognized automatically and so buildboot will place this information directly into the header.

buildboot.exe WINCE imagefile -d3 0xA0801000 0x80000

Patch image file (using WINDOWS CE: usually nk.nb0) with the bootheader information. Load image to SDRAM Address 0xA0080000 and burn this image to flash as the standard bootfile. After next power up, the bootloader copies the file back to SDRAM address 0xA0080000 and it will start at entry address 0xA0801000.

Note: load to SDRAM Address 0xA0080000 as the virtual SDRAM base address is 0x0 using cached and buffered access. Loading to 0x80000 instead to 0xA0080000 will speed up bootstrap performance significantly.

3.1.2 Example 2

buildboot -d0 0xA0A00020 1 0xA0A00000 0x1000 >headerfile

Build a header of 512 bytes which let the bootstraploader load 0x1000 Bytes to SDRAM address 0xA0A00000 (PXA250 SDRAM physical address) and jump to address 0xA0A00020 after download. If you exchange -d0 with -d3 the image will

be downloaded and the given filesize (0x1000 Bytes) will be burned to flash. The Flash address is the next free sector behind the bootloader. The bootloader generates an own bootheader which makes it possible to copy the given filesize of the flash content back to the specified SDRAM address after powerup.

This header may be copied before the image to be booted.

On unix environment: *cat headerfile imagefile >bootfile*

On windows environments: *copy /b headerfile + imagefile bootfile*

This kind of invokement may be used if you want to use a compressed image. You should compress your image with gzip -9 imagefile before. The bootloader test the first word of the image against the gzip MAGIC word and is able to decompress the image as soon as it is burned to flash.

Note: Compression is supported with the -d3 option only

3.2 Example 3

windows:

```
buildboot -d8 0xA0800000 1 0x60000 0x100000 >headerfile
copy /b headerfile + imagefile bootfile
```

unix script:

```
#!bin/sh
imagefile=$1
filelen=`ls -l $imagefile | awk '{ print $4 }'`
buildboot -d8 0xA0800000 1 0x60000 $filelen | cat - $imagefile >bootfile
```

This commands generate a bootfile which lets the bootloader load the imagefile to FLASH address 0x60000. The bootloader uses the SDRAM scratchbuffer at address 0xA0800000. This feature may be used to bring up a file system image to a defined place in flash, e.g. IPSM or Linux compressed ramdisk images.

Note: As this is a direct flash load, no additional headers are generated. Without a header at the first sector behind the bootloaders (usually at 0x60000), the bootloader is not able to start a program.

3.3 Example 4:

Even if the boottime is relatively short, some customers want to show a splash screen during the bootstrap phase. For this purpose you can download a file together with a display description. This file can be loaded from SDMMC or from the internal flash. The example describes the case, that you want to load a bitmap into the internal flash.

bitmapboot.bat sx14.dis oembitmap256.bmp bootfile

bitmapboot generates a file which is burned to flash. (uses -d7 option of buildboot). Sx14.dis is a description file which describes the display timing to the bootloader. Oembitmap256.bmp is a bitmap file with 256 Colors, not compressed. Bootfile specifies the output filename which can be booted via TFTP or sdmmc (autoboot.001).

Having an image loaded to flash, the bootloader is able to show this bitmap during the bootstrap process.

4.0 Booting procedure

The Keith & Koep bootloader image consists of two packages: PBL and bootloader. The PBL (Primary Boot Loader) is a very small piece of code, which is useful to encapsulate the memory initialization procedure and the powermanagement hooks needed for resuming the execution after sleep. It is a complete independent package, which is hooked to the main bootloaded by a USRBOOT header.

At reset the processor begins executing at Addr 0 with flash mapped to Addr 0. The first code initializes the memory interfaces and looks for a **USRBOOT** header at Addr 0x4000. This address lies outside of the standard bootloader image.

If PBL will find a valid bootheader there, it copies the specified portion of flash to SDRAM and starts the execution as specified. If the USRBOOT header at this location is invalid, PBL tries to start the default image which is inside the current bootloader image.

If the default bootloader is started whitout having a copy at the address 0x4000, it will burn a copy of its own to this address and then it will stop the execution. This behaviour brings two advantages:

1. PBL and bootloader may be updated seperately

   The bootloader part may be updated without changing the SDRAM memory initialization done by PBL.

   If you develop bootloaders you may easily fall back to the default if new version does not run.

   If you develop PBL you have quick turnaround times as you do not need to rewrite the full 128KB sector.
2. The second bootloader gives you a chance to fall back to a factory default if bootloader update fails. The main functionality is placed in the bootloader package which is maintained by PBL.

5.0 Bootloader Update

The bootloader might be updated using the standard tftp procedure. After download the first sector of the flash is written. As the bootloader runs out of SDRAM, the download does not interfere with the running bootloader.

Using Trizeps1, the bootloader is stored twice. In this case the PBL always tries to run the copy of the real code. If the copy is missing it runs the original code. But this code just looks for the copy. If it finds no copy it makes a copy and stops execution. So it is possible to develop code changes to the PBL quickly as you have to burn only some kilobytes of code via jtag.

The TRIZEPS2 and TRIZEPS3 bootloaders still have the possiblity to run with a copy but the factory default has been changed to use only one bootloader making the update as simple as possible. So using TRIZEPS2 or TRIZEPS3 you only have to download the update. No other interaction is needed.

Using Trizeps1:

There are 2 kinds of bootloader updates

1. Update of bootloader part

   After the tftp procedure the update is finished.

2. Update of PBL + default bootloader.

   In this case the old bootloader overwrites the address 0 in flash and the default bootloader part. The working copy is untouched. After download you have to enter in your hyperterm window: erase_boot <CR> to erase the working copy in flash.

5.0.1 Old Bootloaders do not have the erase_boot command.

Old bootloaders do not contain this command. You may invalidate the working copy by writing a zero to the USRBOOT magic word. Or you may clamp GPIO26 low during powerup. (Close solder bridge on top of Trizeps).

5.0.2 How to erase the bootloader by command line

Look for the virtual address of the flash which is reported during start.

The example takes 0x05000000 as the virtual start address.

Type in:

cw 0x05040000 0xff40

cw 0x05040000 0x0000

The first command issues a write command to flash. The second command writes a 0 to offset 0x40000.

After the next power up, the new bootloader will start building a working copy.

5.1 Alternate Bootloaders

Usually the Trizeps modules are shipped with bootloaders linked to lower SDRAM space. The default bootloader occupies addresses from 0x4000-8000 for the MMU translation table and ~0x8000...0x76000 for the bootloader code and data space. The bootloader will ask for an alternative loader to load images which might inter-

fere with those spaces. Such an alternative bootloader may be loaded on the fly or it might be loaded and made resident before. Keith & Koep delivers bootloader files named with a xxx_**h**.yyy to mark versions linked to higher addresses. (Usually needed for Linux OS)

Trizeps2: Newer versions are able to remap the bootloader code to another physical address. So if the SDRAM part of the bootloader and the code to be boot would overlap, the bootloader moves it's physical position, remap the code and copy the bootcode into the right pyhsical position. Make sure that all references in your boot-header are made to physical addresses if you want to use this feature.

6.0 Internal Memory Organization

Following table describes the typical usage of flash memory for a WINDOWS CE system. Systems running other operating systems share the regions until 4. Take care to use the epsm and ereg command which erase flash at locations 5, 6.

TABLE 2.

No	Flash Offset	**Table for 16MB Strata Flash**
1	0-0x00000fff	Primary Bootloader. This part is responsible to bring a bootloader or application program into memory and run it. No user interface and no real I/O support until here.
2	0x0001000	Bootloader Backup begins with struct USRBOOT
3	0x0040000	Bootloader working copy or user image begins with struct USRBOOT
4	0x0060000	Bootheader used by Bootloader (1, 2) followed by OS Image or Splash sceen bitmap file which may be followed by another Bootheader with OS Image.
5	0xa000000	Intel Persistant Storage Manager Area (command epsm erases the space upwards)
6	0xf800000	Area for Windows CE Persistant Registry (may be erased by ereg command)

No	Flash Offset	**Table for 32MB Strata Flash**
1	0-0x00000fff	Primary Bootloader. This part is responsible to bring a bootloader or application program into memory and run it. No user interface and no real I/O support until here.
2	0x0001000	Bootloader Backup begins with struct USRBOOT
3	0x0040000	Bootloader working copy or user image begins with struct USRBOOT
4	0x0080000	Bootheader used by Bootloader (1, 2) followed by OS Image or Splash sceen bitmap file which may be followed by another Bootheader with OS Image.
5	0x1100000	Intel Persistant Storage Manager Area (command epsm erases the space upwards)
6	0x1F00000	Area for Windows CE Persistant Registry (may be erased by ereg command)

7.0 HEADER

7.1 USRBOOT Header

This header is used by the PBL. It is used to bring the bootloader into memory and start it. PBL looks at the location USRIMAGE first. If there is a valid bootheader magic (0x55aa5a5a) it starts this image. If this magic is not present it starts the Image which is described by the Header at location DFLTIMAGE.

```
#define FLASH_VIRTADR 0x04000000// Virtual Adress of Flash Memory
#define USRIMAGE      0x40000  // This is the address of the real Bootloader
#define DFLTIMAGE     0x1000  //Address of the  piggypack Bootloader

#define USRIMAGEADR   (FLASH_VIRTADR+USRIMAGE)
#define DFLTIMAGEADR  (FLASH_VIRTADR+DFLT)

#define FLASH_MAGIC 0x55aa5a5a

typedef struct  _usrboot
{
unsigned long magic;    // FLASH_MAGIC
unsigned long *dest;    // copy to address
unsigned long count;    // copy this count of bytes to dest
unsigned long (*entry)(); // after copy branch to this address
unsigned long check;    // checksum
unsigned long data[0];  // placeholder for bootdata, image begins here....

} USRBOOT, *P_USRBOOT;
```

7.2 Boot_hdr

This header is used by the bootp/tftp bootloader. It is responsible to boot an Operating System or user progra. The header is validated by a magic word which may be either FLASH_MAGIC (0x31415926) to identify a program to be booted or a BITMAP_MAGIC to identify a splash screen which is shown during boot. In case of a BITMAP_MAGIC the next header will be followed [len] bytes behind this header.

```
#define FLASH_MAGIC  0x31415926
#define BITMAP_MAGIC 0x31415927

#define ZIP_MAGIC   0x8b1f

struct boot_hdr
{
struct boot_hdr *next;  // Sequence some files currently unused
unsigned long    magic; // FLASH_MAGIC, BITMAP_MAGIC
unsigned long    *dst;  // copy to dest
unsigned long     len;  // image size to be copied to dest
void       (*entry)();  // entry Address
unsigned long data[1];  // placeholder for bootdata, image begins here....
};
```

7.3 Genboot Header

This header is generated by buildboot.exe as previously described.

```
struct arnold_bootheader
{
/*  0  16*/  char magic[16];              //  "ARNOLDBOOTBLOCK"
/* 16   4*/  long exec_adr;               //   execution address
/* 20   4*/  long nosegs;                 //   currently use only 1
/* 24 256*/  struct segment lseg[MAXSEG]; // segment to load
/*280   4*/  long stack_p;                // initial stack ptr
/*282   2*/  short debuggit;              // d flag
/*284   2*/  short writesmart;            // unused
```

```
/*286   2*/  unsigned short extrafile;    // unused
/*288  20*/  char otherfile[20];          // unused
/*308   4*/  unsigned long checkall;      // unused
/*312  32*/  char imgname[32];            // unused
/*344   1*/  long extraparams;            // reserved for future use */
/*348 ...*/  long paramarray[xx];         // future params here
};

union bootblock {
  char buffer[512];
  struct arnold_bootheader boot;
};

struct segment { void *base; long len; };  //len MUST be Multiple of 512
```

7.4 Display definition Header (patched to Genboot Header)

```
extraparams=0x10
```

copy this struct to paramarray

```
typedef struct display_def
{
    unsigned long FBAdr;
    unsigned long Bpp;
    unsigned long Use16BitPalette;
    unsigned long CxScreen;
    unsigned long CyScreen;
    unsigned long Control0;
    unsigned long Control1;
    unsigned long Control2;
    unsigned long Control3;
    unsigned long DataBits;
    unsigned long DisplayOn;
    unsigned long ContrastReg;
    unsigned long ContrastVal;
    unsigned long UsePanelLink;
    unsigned long UseEDTSSP;
    char          dispname[64];

} DISPLAY_INF, *PDISPLAY_INF;

//DisplayOn Bits:
//Set those bits in DisplayOn Variable to reach
//some extras which are not described through
//DISPLAY_INF.
//e.g. If DisplayOn is set to 0x80000000
//Wince takes display params from Bootloader.

#define DO_MASK                 0xFF000000
#define DO_OVERRIDE_REGISTRY    0x80000000
#define DO_SOTICHANNEL          0x40000000
#define DO_RESERVED1            0x20000000
#define DO_RESERVED2            0x10000000
#define DO_INVERTPALETTE        0x08000000
#define DO_PALETTEMONO          0x04000000
#define DO_DISPLAYCACHED        0x03000000
```