XR/station Project

A4X Firmware Manual (Preliminary & Incomplete)

Revision 1.0, May 29, 2024

Table of Contents

1. Overview

The **A4X** firmware forms the contents of the boot ROM of XR/computer platforms.[1] This is the first code that executes in the system after processor reset, and is responsible for initializing the integral devices and enabling the user to select a boot device.

It is designed to be very simple. It executes entirely out of ROM, using a small region in low memory for volatile data. Virtual addressing is disabled and polling is used for all I/O.

This document describes the user interface of the **A4X** firmware, and the fundamentals of its boot protocol, including an overview of the A3X partition table (APT) format.

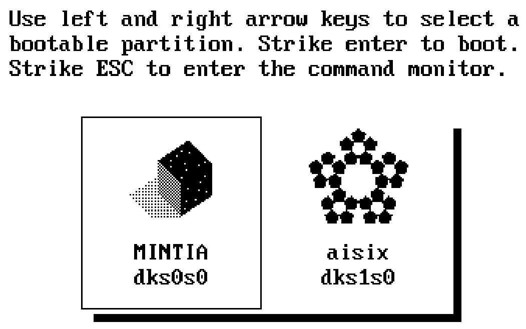
Implementation details of the firmware are not described here; they are best defined by the source code of the firmware itself.

2. User Interface

The firmware contains both a textual user interface (TUI) and a simple graphical user interface (GUI). The TUI is referred to as the "command monitor", and consists of a simple command interface that supports several commands. The GUI consists of a very simple "boot picker".

2.1. Boot Picker

The boot picker appears at system startup when the firmware detects more than one bootable partition is available, and a display adapter is attached. It provides a simple way for the user to select which partition should be booted.


Use left and right arrow keys to select a
bootable partition. Strike enter to boot.
Strike ESC to enter the command monitor.

The appearance of the A4X Boot Picker.

2.2. Command Monitor

The command monitor is available both graphically via an extremely simplistic terminal emulator, and over the serial port. It can be entered graphically via striking ESC at the boot picker, or over Serial Port A by performing a headless boot (i.e. with no displays attached) with either zero or more than one bootable partition installed.

Note that partitions are named in the format **dksXsY**, where **X** is the integer identifier of the disk device, and **Y** is the integer identifier of the partition. If **Y** is 8, such as in **dks0s8**, the entire disk is addressed.

[1]See the *XR/computer Systems Handbook* for more information.

A list of commands available at the time of writing is provided:

Command	Function
help	Display help text about all commands.
autoboot	Boot with default parameters.
reset	Reset the system.
nvreset	Reset the system NVRAM.
listenv	List the NVRAM variables.
setenv	[name] [contents] Set an NVRAM variable.
delenv	[name] Delete an NVRAM variable.
boot	[device (dksNsN)] [args …] Boot from specified device.
listdisk	List all disks, their bootable partitions, and any operating systems installed.
clear	Clear the command monitor.

3. NVRAM

XR/computer systems all contain a 4KB non-volatile RAM (NVRAM) which is used by the system firmware to store persistent information in “NVRAM variables”. These can be viewed and set by the user from the command monitor. All NVRAM variable contents are ASCII strings, but the internal format is undocumented and liable to change. A table of the currently defined NVRAM variables follows:

Name	Function
boot-dev	If **auto-boot?** is set to “true”, contains the name of a preferred partition to boot from (in **dksXsY** format).
auto-boot?	If set to “true”, the system will attempt to automatically boot.
boot-args	Contains the argument string that will be passed to bootstrap software.

Any other variables seen in the **listenv** listing may have been created by the embedded legacy A3X firmware, which is chain-loaded in order to boot legacy operating systems. These variables are undocumented and should be left alone.

4. A3X Partition Table

The partition table format understood by this firmware is the A3X Partition Table format, or **APT**. The following is an overview of this format.

If a disk is formatted with **APT**, sector 0 (bytes 0-511 on disk) will contain the partition table with the following layout:

```
STRUCT AptBootBlock
    // Space for 15 bytes of boot code on relevant platforms.
    BootCode : UBYTE[15],

    // Always contains 0xFF.
    FfIfVariant : UBYTE,

    // Eight partition table entries.
    Partitions : AptEntry[8],

    // The 32-bit magic number must read 0x4E4D494D.
    Magic : ULONG,

    // A 15-character, null-terminated label for the disk.
    Label : UBYTE[16],
END

STRUCT AptEntry
    // A 7-character, null-terminated label for the partition.
    Label : UBYTE[8],

    // A 32-bit count of sectors in the partition.
    SectorCount : ULONG,

    // The status of the partition. Contains zero if the partition
    // table entry is unused. Otherwise, it contains any non-zero
    // value.
    Status : ULONG,
END
```

5. Booting

This section describes the boot protocol used by the **A4X** firmware. The old A3X boot protocol is also supported via an embedded A3X firmware which is chain-loaded when a legacy operating system is selected, but will not be documented here.

Note that all of the client-facing structures and services described here (in general, everything prefixed with `Fw`) can be found in the `Headers/a4xClient.hjk` header file, which should be included in order to access them from programs written in Jackal.

A partition is bootable if it contains a valid OS record at an offset of 1 sector from the partition base (bytes 512-1023 within the partition). The OS record sector has the following layout:

```
STRUCT AptOsRecord
    // The 32-bit magic number must read 0x796D6173.
    Magic : ULONG,

    // A 15-character, null-terminated label for the installed
    // operating system.
    OsName : UBYTE[16],

    // The sector offset within the partition, at which the bootstrap
    // program begins.
    BootstrapSector : ULONG,

    // The count of sectors in the bootstrap program.
    BootstrapCount : ULONG,
END
```

If a valid OS record is found, the partition is assumed to be bootable. In the following sector (sector 2), a 64x64 monochrome bitmap is located. This is used as an icon in the boot picker.

5.1. The Bootstrap Program

When booting from a partition, the bootstrap sectors are loaded in sequence off the disk into physical memory beginning at address 0x3000. The first 32 bits of the bootstrap must be 0x676F646E in order to be considered valid. Control is then transferred to address 0x3004 through the Jackal function pointer with the following signature:

```
FNPTR FwBootstrapEntrypoint (
    IN devicedatabase : ^FwDeviceDatabaseRecord,
    IN apitable : ^FwApiTableRecord,
    IN bootpartition : ^VOID,
    IN args : ^UBYTE,
) : UWORD
```

That is, as per the Jackal ABI for XR/17032, a pointer to the **DeviceDatabase** is supplied in register `a0`, a pointer to the **ApiTable** is supplied in register `a1`, a handle to the boot partition is supplied in register `a2`, and an argument string is supplied in register `a3`. The bootstrap program can return a value in register `a3`.

Note that memory in the range of 0x0 through 0x2FFF should *not* be modified until **A4X** services will no longer be called, as this region is used to store its runtime data (such as the initial stack). After this region is trashed, **A4X** may only be re-entered through a system reset (which can be accomplished by jumping to physical address 0xFFFE1000 with virtual addressing disabled).

5.2. The Device Database

The **DeviceDatabase** is a simple structure constructed in low memory by the firmware. It contains information about all of the devices that were detected. It has the following layout:

```
STRUCT FwDeviceDatabaseRecord
    // 32-bit count of the total RAM detected in the system.
    TotalRamBytes : ULONG,

    // The number of processors detected.
    ProcessorCount : UBYTE,

    // The number of bootable partitions found.
    BootableCount : UBYTE,

    Padding : UBYTE[2],

    // A table of information about all of the RAM slots.
    Ram : FwRamRecord[FW_RAM_MAX],

    // A table of information about all of the physical disks.
    Dks : FwDksInfoRecord[FW_DISK_MAX],

    // A table of information about devices attached to the Amtsu
    // peripheral bus.
    Amtsu : FwAmtsuInfoRecord[FW_AMTSU_MAX],

    // A table of information about the boards attached to the EBUS
    // expansion slots.
    Boards : FwBoardInfoRecord[FW_BOARD_MAX],

    // A table of information about each processor detected in the
    // system.
    Processors : FwProcessorInfoRecord[FW_PROCESSOR_MAX],

    // Information about the boot framebuffer, or lack thereof.
    Framebuffer : FwFramebufferInfoRecord,

    // Information about the boot keyboard, or lack thereof.
    Keyboard : FwKeyboardInfoRecord,

    // The machine type --
    // XR_STATION, XR_MP, or XR_FRAME.
    MachineType : FwMachineType,
END
```

Note that the `Headers/a4xClient.hjk` header file should be used to access the device database and other **A4X** structures - this incomplete information is only provided here for quick reference.

5.3. The API Table

A pointer to an API table is passed to the bootstrap program. The API table consists of function pointers that can be called to receive services from the firmware. The currently defined APIs follow:

```
STRUCT FwApiTableRecord
    PutCharacter : FwApiPutCharacterF,
    GetCharacter : FwApiGetCharacterF,
    ReadDisk : FwApiReadDiskF,
    PutString : FwApiPutStringF,
    KickProcessor : FwApiKickProcessorF,
END
```

5.3.1. PutCharacter

```
FwApiPutCharacterF (
    IN byte : UWORD,
)
```

Puts a single character to the firmware console.

5.3.2. GetCharacter

```
FwApiGetCharacterF () : UWORD
```

Returns a single byte from the firmware console. This is non-blocking and returns −1 (0xFFFFFFFF) if no bytes are available.

5.3.3. ReadDisk

```
FNPTR FwApiReadDiskF (
    IN partition : ^VOID,
    IN buffer : ^VOID,
    IN sector : ULONG,
    IN count : ULONG,
) : UWORD
```

Reads a number of sectors from the specified partition handle into the buffer. The base address of the buffer *must* be aligned to a sector size boundary. Returns **TRUE** (non-zero) if successful, **FALSE** (zero) otherwise.

5.3.4. PutString

```
FNPTR FwApiPutStringF (
    IN str : ^UBYTE,
)
```

Puts a null-terminated string of bytes to the firmware console. Could be easily synthesized from **PutCharacter** but is provided for convenience to small boot sectors written in assembly language.

5.3.5. KickProcessor

```
FNPTR FwApiKickProcessorF (
    IN number : UWORD,
    IN context : ^VOID,
    IN callback : FwKickProcessorCallbackF,
)
```

Causes the processor with the specified number to execute the provided callback. The callback routine is called with the opaque context value and with the number of the processor. Its signature follows:

5.3.5. KickProcessor

```
FNPTR FwKickProcessorCallbackF (
    IN number : UWORD,
    IN context : ^VOID,
)
```

KickProcessor does not wait for the callback to be executed; execution continues on both processors asynchronously. If synchronization is required, it must be implemented manually.

If the processor with the given number (equivalent to its index in the **DeviceDatabase**) does not exist, the results are undefined.

Also note that the firmware does not contain multiprocessor synchronization, so if firmware services may be invoked by multiple processors concurrently, locking must be provided by the user.

The size of the initial stack provided by the firmware for processors other than the boot processor is not explicitly defined here, but is quite small, so the depth of the call stack during execution of the callback must either be very small, or the recipient processor must switch quickly to a new stack.