Ultimate 1MB/Incognito

Alt BIOS Technical and Developer Documentation

By Jonathan Halliday

Draft 2

Revised to cover main BIOS v.1.06, PBI BIOS 1.64, and XEX Loader v.1.06.

Contents

Introduction

This document describes the technical specifications and memory map of the “Alt” BIOS for the Ultimate 1MB and Incognito upgrades for the Atari XL/XE and 800 machines respectively. The new BIOS provides a powerful alternative to the stock firmware written by Sebastian Bartkowicz (Candle), who also designed and produced both pieces of hardware. Although the new BIOS was initially written to facilitate a few niche requirements not addressed by the original (booting the “GOS” segment of the flash ROM, setting the RTC from the BIOS setup screen, fixing the day-of-the-week numbering discrepancy between the BIOS and the SpartaDOS X RTC driver, adding a dedicated PBI hard disk configuration menu), several other additional features were eventually introduced (high-speed SIO, dual configuration profiles, hardware detection, diagnostic cart boot, and plug-in modules). A companion update to the PBI BIOS was also provided, and a new XEX loader written. This document deals only with the technical details of the new main BIOS, with reference to the PBI BIOS where necessary.

For a complete and detailed reference to the Ultimate 1MB hardware, readers are encouraged to refer to Avery Lee’s Altirra Hardware Reference Manual.

Conventions

Since this document pertains to both Ultimate and Incognito, we shall refer to the hardware as “U1MB”, making specific reference to “Incognito” wherever the Incognito specification deviates from that of Ultimate 1MB.

BIOS Initialisation Process

When the Atari computer is booted or reset, the U1MB or Incognito BIOS ROM is mapped at $C000-$CFFF and $D800-$FFFF, and the CPU begins execution at the address pointed to by the machine Reset vector at $FFFC. Upon reset, the Ultimate configuration register at $D380 is unlocked, allowing changes to the configuration until bit 7 of $D380 is set by the BIOS before control is passed to the Atari OS.

Like the original BIOS, the new BIOS reads the system configuration stored in the NVRAM of the DS1305 RTC chip at every reset and writes this configuration to the unlocked U1MB configuration registers. It also checks whether the BIOS Setup entry hotkey is pressed, and if it is, the BIOS setup menu is entered. In the original BIOS, the hotkey was always “Help”, but the new BIOS allows for use of the “Start” key as an alternative.

In addition to basic configuration, the new BIOS performs a series of hardware tests following the reset phase, depending on whether the machine is performing a warm reset or a cold boot. Certain tests (such as CPU type and speed) are only carried out prior to entering the setup menu.

Certain quirks of the VHDL are capitalised upon by the U1MB BIOS, such as the ability to map the ROM under $D000-$D7FF to the Self-Test area at $5000. The new BIOS does this after first initialising PORTB as an output. Incognito, meanwhile, lacks this capability but the absence of BIOS plugins on the 800 compensates somewhat for the inability to access this useful 2KB ROM block.

The new U1MB BIOS also attempts to establish whether the machine is an XEGS, and if it is not, it deactivates XEGS game ROM selection. Tests are also made for Soundboard and Stereo Pokey hardware, and – if found – the firmware tries to discover whether the hardware is under U1MB control, greying out Soundboard and Stereo options if the this is not the case. Unfortunately, Covox hardware is not detectable in software, but the plugin architecture means that the Covox option can be removed if it is not required.

Memory Usage

Below is a map of ROM and RAM areas used by the U1MB and Incognito BIOS:

Address	**U1MB**		**Incognito**
$30-$35	Pointers		Pointers
$38-3B	Pointers		Pointers
$0200	VDSLST		VDSLST
$0222	VVBLKI		VVBLKI
$0100-$0153	Buffers		Buffers
$C000-$C1FF	BIOS metadata and jump table		BIOS metadata and jump table
$C200-$CFFF	BIOS code		BIOS code
$D000-$D7FF	BIOS code	IO RAM	IO RAM
$D800-$DFFF	BIOS code		BIOS code
$E000-$E3FF	OS Character Set (for Turbo Freezer)		
$E400-$FAFF	BIOS code		
$FB00-$FEFF	BIOS plug-in code		
$FF00-$FFBF	Editable ROM slot descriptions		Editable ROM slot descriptions
$FFC0-$FFDF	System ROM space descriptions		System ROM space descriptions
$FFE0-$FFF9	Unused		Unused
$FFFA-$FFFF	Machine vectors		Machine vectors

RAM Usage

Since the BIOS setup menu needs to be accessible without disturbing the underlying OS and applications, RAM outside of the IO region has been carefully chosen to cause minimal disruption following system reset. Locations used in page zero ($30-$35 and $38-$3B) are designated for external device (PBI) use, while VDSLST ($0200) and VVBLKI ($0222) are reinitialised by the OS on system reset. Usage of the bottom of the stack – though more extensive than with the original BIOS – should not cause issues.

BIOS Metadata and Jump Table

The BIOS metadata and jump table (at the bottom of the 16KB ROM) has the following layout:

Address	Size (bytes)	Description	Defined Content	
			U1MB	**Incognito**
$C000	6	Firmware signature (ASCII)	'ULBIOS'	'INBIOS'
$C006	1	Major revision (BCD)		
$C007	1	Minor revision (BCD)		
$C008	3	Revision date (DD/MM/YY)		
$C00B	1	SpartaDOS X ROM Size in 8KB banks	$18, $20, or $28	$18, $20, or $28
$C00C	2	Editable slot name base address	$FF00	$FF00
$C00E	2	Address of NVRAM configuration buffer		
$C010	2	Address of internal option table		
$C012	3	JMP to activate menu items routine		
$C015	3	JMP to deactivate menu items routine		
$C018	3	JMP to disable items routine		

Although the Incognito BIOS does not support plugin modules, the jump table is still present. The SpartaDOS X ROM size dictates whether the GOS slot is available (i.e. if SDX is 192KB long), and allows optional use of 256KB and 320KB SDX ROMs if the GOS is not required. Software designed for BIOS customisation must write $18, $20 or $28 to the SDX ROM without changing any other part of the BIOS header.

Editable ROM Slot Descriptions

Editable ROM slot descriptions are encoded in a completely different manner to those found in the original BIOS. In the standard firmware, slot descriptions were fourteen bytes long (fifteen bytes in Incognito) and encoded using Antic display codes rather than ATASCII. Moreover, slot descriptions immediately followed the slot heading (OS, BASIC, or XEGS), which was repeated four times for each slot type. Although this repetition of the entire menu item was simply a side-effect of the BIOS menu design, it made it possible for unrestrained editing of the slot descriptions (whether accidental or wilful) to completely destroy the menu layout.

A limitation of the original BIOS menu only noticeable some time after it was written was that fourteen characters was insufficient to accommodate the title of the default XEGS game slot ("Missile Command"). The description size was therefore extended to fifteen characters in the new BIOS (matching the original Incognito description size) to remove the need for such abbreviations.

The new slot descriptions are encoded as NUL terminated ATASCII strings instead of using internal screen codes. Control, reverse video, and other special non-alphanumeric characters should not be used. Spaces are permitted. Maximum string length is fifteen characters.

Lastly, the slot description base address is now the same in both the U1MB and Incognito firmware, and positioned near the end of the ROM.

The U1MB ROM description fields are laid out as follows:

Address	**Type**	**Default Content**
$FF00	OS slot 0	'High-Speed'
$FF10	OS slot 1	'Diagnostic'
$FF20	OS slot 2	'Q-Meg'
$FF30	OS slot 3	'Stock XL/XE'
$FF40	BASIC slot 0	'BASIC'
$FF50	BASIC slot 1	'CAR1'
$FF60	BASIC slot 2	'CAR2'
$FF70	BASIC slot 3	'CAR3'
$FF80	XEGS slot 0	'Missile Command'
$FF90	XEGS slot 1	'XEGS1'
$FFA0	XEGS slot 2	'XEGS2'
$FFB0	XEGS slot 3	'XEGS3'

Incognito ROM description fields are as follows:

Address	Type	Default Content
$FF00	Colleen OS slot 0	‘OS A’
$FF10	Colleen OS slot 1	‘OS A + FastChip’
$FF20	Colleen OS slot 2	‘OS B’
$FF30	Colleen OS slot 3	‘OS B + FastChip’
$FF40	XL/XE OS slot 0	‘High-Speed’
$FF50	XL/XE OS slot 1	‘Diagnostic’
$FF60	XL/XE OS slot 2	‘Q-Meg’
$FF70	XL/XE OS slot 3	‘Stock XL/XE’

NVRAM Configuration Layout

The new BIOS uses addresses $20-$2E and $40-$4D of the U1MB’s SD1305’s NVRAM. The U1MB/Incognito build of SpartaDOS X uses bytes $30-$31, so the entire $3x address range is deliberately avoided by the BIOS. The Incognito BIOS uses slightly fewer bytes, but $20-$2F and $40-$4F should now be considered reserved for U1MB and Incognito. Moreover, addresses $50-$55 are used by the XEX loader for configuration data. While it’s unlikely that NVRAM usage will significantly increase in future BIOS revisions, it would be unwise to make assumptions when designing your own drivers or any other software which makes use of the NVRAM. It would probably be wise, if in doubt, to use the upper range of addresses ($7x, for example).

Fortunately, developers wishing to store configuration data for additional hardware may do so via BIOS plugins (U1MB only). NVRAM space is reserved in the U1MB configuration records for exactly this purpose.

DS1305 User NVRAM Layout

Address	Size		Description
	U1MB	Incognito	
$20	14	12	Configuration profile 1
$2E	1	1	Profile number (0 or 1)
$40	14	12	Configuration profile 2
$50	6	6	XEX loader configuration

U1MB Configuration Data

The U1MB configuration data is arranged as follows (offsets from NVRAM profile address)

Offset	Bit usage	Description
+0 (Cfg)	0-1	RAM size (00 = stock, 01 = 320, 10 = 576, 11 = 1088)
	2-3	OS slot (0-3)
	4	SDX (actually bank switching enable) (0 = on, 1 = off)
	5	Unused
	6	Reserved for IORAM flag
	7	Reserved for config lock
+1 (Aux)	0	Stereo (pin M0, P4) (0 = off, 1 = on)
	1	Covox (pin M1, P4) (0 = off, 1 = on)
	2	Pin S0, P4 (0 = off, 1 = on)
	3	Pin S1, P4 (0 = off, 1 = on)
	4	VBXE address (0 = 0xD640, 1 = 0xD740)
	5	VBXE disable (0 = enabled, 1 = disabled)
	6	Soundboard enable/disable (0 = off, 1 = on)
	7	Flash write disable (0 = flashable, 1 = disabled)
+2 (Aux2)	0-1	PBI device ID (00 = 0, 01 = 2, 10 = 4, 11 = 6)
	2	PBI BIOS (0 = off, 1 = on)
	3	SIDE button (0 = off, 1 = on)
	4-5	BASIC slot (0-3)
	6-7	XEGS slot (0-3)
+3 (Aux3)	0-3	HDD boot partition (0 = off, 1-15 = drive number)
	4	Use boot partition in partition table header (0 = off, 1 = on)
	5	Drive 1 swap (0 = off, 1 = on)
	6	SIO device selection (0 = D1-D4, 1 = All)
	7	HDD (0 = off, 1 = on)
+4 (Aux4)	0-3	SIO flags (bit 0 = drive 1, ..., bit 3 = drive 4)
	4-5	SIO mode (0 = off, 1 = HSIO, 2 = SIO2BT, 3 = HSIO+SIO2BT)
	6-7	HDD write lock (0 = disabled, 1 = physical disk, 2 = global)
+5 (Aux5)	0	BIOS menu hotkey (0 = Help, 1 = Start)
	1	Key click (0 = off, 1 = on)
	2	Reboot hotkey (0 = disabled, 1 = Select)
	3	PBI BIOS partition table re-read hotkey (0 = disabled, 1 = Shift)
	4	BIOS splash screen (0 = off, 1 = on)
	5	PBI BIOS version message (0 = off, 1 = on)
	6	Boot to loader (0 = off, 1 = on)
	7	GOS (0 = off, 1 = on)

Offset	Bit usage	Description
+6 (Aux6)	0-3	Menu colour (0-15)
	4	Joystick (0 = port 1, 1 = port 2)
	5	Joystick disable (1 = disabled, 0 = enabled)
	6	5 minute screen timeout (0 = disabled, 1 = enabled)
	7	Reserved
+7 (Aux7)	0-7	Reserved
+8,9 (Ext ID)	0-15	16 bit extension module ID
+10,11 (Ext)	0-15	16 bits of extension configuration
+12,13 (CRC)	0-15	16 bit CRC of configuration profile

The MADS struct declaration for the U1MB configuration buffer looks like this:

```
        .struct Cfg
Config          .byte ; SDX, OS, RAM
Aux             .byte ; VBXE, Covox, Stereo
Aux2            .byte ; XEGS, BASIC, SIDE, PBI ID
Aux3            .byte ; HDD options
Aux4            .byte ; HSIO options
Aux5            .byte ; Misc options
Aux6            .byte ; Misc options
Aux7            .byte ; reserved
ExtID           .word ; Extension ID
Ext1            .byte ; Extension bits
Ext2            .byte ; Extension bits
CRC             .word ; 16-bit CRC of config data
        .ends
```

The struct tags should be used to refer to the encoding address of plugin configuration data (see section on Plugins).

Incognito configuration data

Offset	Bit usage	Description
+0 (Hardware)	0-4	Unused
	5	Colleen mode (0 = off, 1 = on)
	6	reserved for IORAM flag
	7	reserved for config lock
+1 (Colleen Cfg)	0-1	RAM size (0-3)
	2-3	OS (0-3)
	4	SDX (0 = on, 1 = off)
	5-7	Unused
+2 (Colleen Aux)	0-1	Unused
	2	SIDE (0 = off, 1 = on)
	3	Axlon (0 = off, 1 = on)
	4	BASIC (0 = off, 1 = on)
	5	Unused
	6	Loader (0 = off, 1 = on)
	7	Unused
+3 (XL/XE Cfg)	0-1	RAM size (0-3)
	2-3	OS (0-3)
	4	SDX (0 = on, 1 = off)
	5-7	unused
+4 (XL/XE Aux)	0-1	PBI Device ID (0 - 3)
	2	PBI BIOS (0 = off, 1 = on)
	3	Unused
	4	Joysticks 3-4 (0 = off, 1 = on)
	5	Unused
	6	Loader (0 = off, 1 = on)
	7	Unused
+5 (Aux3)	0-3	HDD boot partition (0 = off, 1-15 = drive number)
	4	Use boot partition in partition table header (0 = off, 1 = on)
	5	Drive 1 swap (0 = off, 1 = on)
	6	SIO device selection (0 = D1-D4, 1 = All)
	7	HDD (0 = off, 1 = on)
+6 (Aux4)	0-3	SIO flags (bit 0 = drive 1, ..., bit 3 = drive 4)
	4-5	SIO mode (0 = off, 1 = HSIO, 2 = SIO2BT, 3 = HSIO+SIO2BT)
	6-7	HDD write lock (0 = disabled, 1 = physical disk, 2 = global)

Offset	Bit usage	Description
+7 (Aux5	0	BIOS menu hotkey (0 = Help, 1 = Start)
	1	Key click (0 = off, 1 = on)
	2	Reboot hotkey (0 = disabled, 1 = Select)
	3	PBI BIOS partition table re-read hotkey (0 = disabled, 1 = Shift)
	4	BIOS splash screen (0 = off, 1 = on)
	5	PBI BIOS version message (0 = off, 1 = on)
	6	Boot to loader (0 = off, 1 = on)
	7	GOS (0 = off, 1 = on)
+8 (Aux6)	0-3	Menu colour (0-15)
	4	Joystick (0 = port 1, 1 = port 2)
	5	Joystick disable (1 = disabled, 0 = enabled)
	6	5 minute screen timeout (0 = disabled, 1 = enabled)
	7	Reserved
+9 (Aux7)	0-7	Reserved
+10,11 (CRC)	0-15	16 bit CRC of configuration profile

The MADS struct declaration for the Incognito configuration buffer looks like this:

```
        .struct Cfg
Hardware        .byte ; type (800 or XL/XE)
ColleenConfig   .byte ; OS, RAM, SDX (Colleen)
Aux             .byte ; SIDE (Colleen)
Config          .byte ; OS, RAM, SDX
Aux2            .byte ; SIDE, controllers 3-4
Aux3            .byte ; XL/XE HDD options
Aux4            .byte ; HSIO options
Aux5            .byte ; misc options
Aux6            .byte ; misc options
Aux7            .byte ; reserved
CRC             .word ; 16-bit CRC of config data
        .ends
```

U1MB BIOS Plugins

The U1MB BIOS supports 1KB plugin modules assembled at $FB00. Plugins may be inserted into the BIOS ROM image at file offset $3B00 using a suitable editing tool. As well as providing support for diverse hardware controlled by the U1MB P2 header signals, plugins may also store information in the U1MB NVRAM for retrieval by hardware without a software configuration system of its own. Another reason for plugins is the removal of extraneous BIOS menu options (such as Covox for systems where Covox hardware is not present).

Plugins perform a limited set of tasks. They can:

1. Define editable menu items in any of the eight BIOS setup menus
2. Provide a custom title for the 'Device Control' menu
3. Save and retrieve state of said menu items in the U1MB NVRAM
4. Activate, deactivate ('grey out') and temporarily disable newly defined menu items
5. Establish whether a specific hardware device is controlled by U1MB via the Aux signals
6. Toggle the state of a hardware device controlled by Aux signals
7. Write to hardware registers on System Reset

The default U1MB plugin adds Stereo Pokey and Covox selections to the "System Clock and Features" menu, and defines two stub menu items ("Device 2" and "Device 3") to the "Device Control" menu (these place holder menu items to work, however, and control the S0 and S1 pins of P2 respectively). In addition, it adds an 'Audio Hardware' entry to the 'System Information' page.

Further levels of hardware control may be achieved by placing configuration data into the U1MB NVRAM and extracting said information via the firmware of the target hardware. As an example, an APT BIOS for the KMK/JZ IDE HDD host adapter exists which derives its configuration from U1MB NVRAM data placed there by a custom U1MB KMK/JZ IDE BIOS plugin. This requires the firmware of the target hardware to a) access the NVRAM and establish whether the expected plugin is present, and b) read the configuration bits. The KMK/JZ is therefore able to derive its boot partition, spin-up delay, etc, from the U1MB NVRAM at boot time. In addition, the KMK/JZ U1MB plugin turns the KMK/JZ on and off via the S0 pin of the P2 header (connected to the KMK/JZ disable jumper via a U-Switch device with the bit logic jumper in the reverse logic position).

The KMK/JZ plugin also completely redefines the Device Control menu as a dedicated HDD control menu and dims menu items according to whether the target hardware is detected, activated, and controlled by the S0 signal.

Plugin Structure

BIOS plugins begin with a 64-byte metadata header arranged thus:

Address	Offset	Size (Bytes)	Description	Content
$FB00	$0000	8	Plugin signature (8 ATASCII bytes)	'ULPLUGIN'
$FB08	$0008	1	Host BIOS minimum major revision number (BCD)	
$FB09	$0009	1	Host BIOS minimum minor revision number (BCD)	
$FB0A	$000A	16	Plugin name (NUL terminated ATASCII string)	
$FB1A	$001A	1	Plugin major revision number (BCD)	
$FB1B	$001B	1	Plugin minor revision number (BCD)	
$FB1C	$001C	2	Mnemonic plugin signature (2 ATASCII bytes)	
$FB1E	$001E	2	Pointer to menu setup routine	
$FB20	$0020	2	Pointer to menu update routine	
$FB22	$0022	2	Pointer to plugin's hardware test routine	
$FB24	$0024	2	Pointer to plugin's hardware setup routine	
$FB26	$0026	26	Menu title (NUL terminated ATASCII string)	
$FB40	$0040	-	First node of plugin menu structure	

Host BIOS revision number

This is the lowest BIOS revision guaranteed to work with the plugin. Software which modifies the BIOS may use this information to establish that the target BIOS is of a sufficiently recent revision to support the plugin.

Plugin Name

The plugin name describes the general purpose of the plugin. For example, the default plugin name is 'Stereo/Covox'.

Mnemonic Plugin Signature

This is a descriptive ID. The default plugin signature is 'SC' (denoting Stereo/Covox). This signature is written to the plugin ID field of both NVRAM profiles (see NVRAM data description). Dependent hardware may therefore check that the currently active BIOS plugin is of the expected type. For example, the custom KMK/JZ BIOS discussed earlier checks for the 'IS' (IDEa/Stereo) plugin ID before deriving its configuration data from the plugin configuration bits in NVRAM. If the plugin ID is not of the expected value, then the target hardware should ignore the plugin configuration bits.

Pointer to Menu Setup Routine

This is a 16-bit pointer to the code which sets up the initial state of the 'Device Control' menu (although this menu may be re-titled by the plugin). Setting up the menu involves enabling and dimming or disabling menu items depending on the discovered hardware and whether said hardware can be successfully controlled by U1MB P2 header signals. The source to the default plugin and the KMK/JZ IDEa plugin are provided with this documentation. If the menu setup routine is not defined, a NULL pointer should be supplied, or the address provided should point to an RTS instruction.

Pointer to Menu Update Routine

This is a 16-bit pointer to code which runs immediately after the user changes any option or chooses an actionable menu item. The code runs before the subsequent screen redraw, so this is the section of code you use to dim or enable menu items or change other settings in response to user action. If

the menu update routine is not defined, a NULL pointer should be supplied, or the address provided should point to an RTS instruction.

Pointer to Plugin's Hardware Test Routine

This is the address of the plugin code which tests for the presence of target hardware and (optionally) establishes whether the hardware is under software control via on the P2 signals. In the case of the default plugin (Stereo/Covox), this routine attempts to detect Stereo Pokey after first setting M0 to 0 and then again after setting M0 to 1. If stereo is detected in neither case, it is assumed that Stereo hardware is not present. If stereo is detected in both cases, it is assumed that stereo hardware is present but is not under software control. If stereo is detected only after M0 is set to 1, it is assumed that stereo hardware is present and that it is under software control.

The plugin should set a flag (in plugin RAM – see the subsequent 'Plugin RAM' section) to tell the menu initialisation routine whether relevant items should be dimmed (if software control is not possible), or placed in an 'always on' or 'always off state', depending on whether the hardware was actually detected.

Pointer to Plugin's Hardware Setup Routine

This is the address of the plugin code which writes to hardware registers after the user leaves BIOS setup and when the system Reset key is pressed. If the target hardware was controlled by – for example – a register in the IO area, the hardware setup routine would write to this register, reflecting the selections the user made while in setup.

Menu Title

This is a 26 byte string (including trailing zero) which will be displayed as the title of the 'Device Control' menu. The KMK/JZ plugin titles the menu 'IDEa APT Hard Disk'.

Plugin RAM

Eight bytes of IO RAM at $D7F8-$D7FF are set aside exclusively for a plugin's internal use. Note that the RAM is volatile and is cleared when the user leaves the BIOS setup menu. A further eight bytes of RAM are provided at $D7F0-$D7F7 for menu item values, which are encoded in NVRAM when the user saves current BIOS settings.

Plugin Menu Items

The menu structure of the new BIOS was deliberately designed to be extensible. The menus are arranged as a single forward linked list whose tail points to the location of the first node of the plugin's menu structure. Therefore a plugin must contain at least one menu item, and the last menu item's 'next node' pointer should be NULL.

Because menu items are not stored 'in order' and instead include a field describing which menu the item should appear in, plugins may add items to any of the eight menus in the BIOS setup utility. Most commonly, however, plugins will add items to the 'Device Control' menu (numbered 5, since menus are numbered 0 through 7). The default plugin, however, also defines two extra items in the 'System Clock and Features' menu and a non-editable item in the 'System Information' menu. When the user moves between menus, the entire menu list is scanned and a sub-list built of all the items

which appear in the selected menu. Menus may contain no more than twelve items (the PBI HDD menu therefore already being full).

As a side-note, although Incognito does not provide BIOS plugin extensibility, the list-based menu structure made switching between the two hardware platforms (Colleen and XL/XE) very easy indeed from a coding perspective. The list head pointer is simply switched between a list of Colleen menu items and a list of XL/XE menu items, and the last node of both these lists points to the first menu item common to both hardware types.

Menu Item Structure

This is the MADS STRUCT declaration for a BIOS menu item:

```
.struct Item
       Menu           .byte ; number of menu to which the item belongs
       Next           .word ; pointer to next menu item (NULL = no more items)
       Title          .word ; pointer to menu item heading
       Type           .byte ; item type
       Value          .word ; pointer to current value
       Help           .word ; pointer to context help string for this item
       CfgOff         .byte ; byte offset into config buffer
       CfgMask        .byte ; config buffer bitmask
.ends
```

Fields are described below:

Menu

Number of menu in which item appears (0-7, commonly 6 for plugins).

Next

Pointer to next menu item (NULL for end of list).

Title

Pointer to menu item heading (NUL-terminated string).

Type

Menu item type (see Menu Item Types below).

Value

Pointer to variable holding current value of menu item's selection.

Help

Pointer to NUL-terminated string which will be displayed in status area when menu item is selected (option, NULL pointer value means no explanatory text).

CfgOff

Configuration offset, i.e. byte offset into configuration buffer where *Value* will be encoded. See section on U1MB Configuration data for a list of offsets. There are two bytes of NVRAM reserved for plugin configuration use.

CgfMask

Bitmask describing position and bit-width of item value when encoded in the NVRAM configuration byte pointed to by *CfgOff*.

Menu Item Types

Below is the MADS declaration of enumerated types of menu items:

```
.enum ItemType
       Action         ; item runs code
       OnOff          ; toggle (0 = Disabled, 1 = Enabled)
       OnOffXOR       ; toggle (bit logic reversed)
       List           ; list of strings
       DriveNum       ; drive number spinner (0 = Off, 1-15 = D1:-DO:)
       PartNum        ; drive number spinner + 16 = "on disk"
       Date           ; three 8-bit ints
       Time           ; three 8-bit ints
       Spinner        ; numeric spinner
       String         ; string
.ende
```

Types are described below:

Action

This menu item jumps to the address pointed to by the *Value* field plus one (by pushing the address onto the stack and then executing an RTS instruction) when the user presses Return. *CfgOff* and *CfgMask* are ignored. For example:

```
       .local Item40
@      dta Item[0] (6,Item41,Title,ItemType.Action,LaunchLoader-1,0,0,0)
Title
       .byte 'SIDE XEX/ATR Loader',0
       .endl
```

The code above defines a menu item called ‘SIDE XEX/ATR Loader’ which runs code pointed to by *LaunchLoader-1* when the user presses Enter. The menu item will appear in menu 6, and the next item in the list is *Item41*. It is highly unlikely that the Action type would be employed in plugin code.

OnOff

This type of menu item simply toggles *Value* between 0 and 1 and displays ‘Disabled’ when the value is 0 and ‘Enabled’ if the value is 1. For example:

```
       .local StereoToggle
@      dta Item[0] (1,CovoxToggle,Title,ItemType.OnOff,StereoPokey,Help,Cfg.Aux,$01)
Title
       .byte 'Stereo Pokey',0
Help
       .byte 'Enable/disable stereo audio',0
       .endl
```

In this example, item *StereoToggle* defines an item in menu 1 with an on/off state. The on/off state is stored in *StereoPokey* and is encoded in bit 0 of *Cfg.Aux* and is 1 bit wide (therefore holding the value 0 or 1). The item heading is ‘Stereo Pokey’ and the help text ‘Enable/disable stereo audio’ is displayed when the user cursors onto the item. The next item pointed to is *CovoxToggle*.

OnOffXOR

This type is identical to OnOff with the exception that the logic is reversed (i.e. 1 = Disabled, 0 = Enabled). Example:

```
        .local Item36
@       dta Item[0] (4,Item37,Title,ItemType.OnOffXOR,Config.FlashLock,Help,Cfg.Aux,$80)
Title
        .byte 'Flash writes',0
Help
        .byte '%sROM flashing',0
        .word txtEnableDisable
        .endl

txtEnableDisable
        .byte 'Enable/disable ',0
```

In this example, *Item36* defines an on/off item in menu 4 with reverse binary logic. 0 or 1 is stored in the internal variable *Config.FlashLock* and encoded in bit 7 of Cfg.Aux when written to the NVRAM. The item title is 'Flash writes' and the help text is 'Enable/disable ROM flashing'. Note that the BIOS includes a formatted *printf* routine similar to that found in standard C libraries, and in the example above, *txtEnableDisable* is employed as a form of tokenisation.

List

This item type defines a list, and the index to the selected item will be placed in *Value*. Example:

```
        .local Item1
@       dta Item[0] (0,Item2,Title,ItemType.List,Config.RAM,Help,Cfg.Config,$03)
        .byte 4
        .word List1
        .word List2
        .word List3
        .word List4
Title
        .byte 'Extended RAM',0
List1
        .byte 'Stock',0
List2
        .byte '320KB RAMBO',0
List3
        .byte '576KB CompyShop',0
List4
        .byte '1088KB RAMBO',0
Help
        .byte 'Select memory size',0
        .endl
```

Here, a four-element list is defined in menu 0. The number of list items immediately follows the item declaration, and following that is a list of pointers to strings, one string per list item. Since the index ranges from 0 to 3, two bits are required. The value is stored in *Config.RAM* and encoded in bits 0-1 of *Cfg.Config* when written to the NVRAM buffer.

DriveNum

The *DriveNum* item type implicitly declares a list comprising sixteen elements. Element 0 maps to 'off', while values 1-15 equate to and display drive specifiers D1: through DO: (drive 15). Example:

```
        .local Item49
@       dta Item[0] (5,Item50,Title,ItemType.DriveNum,Config.User2,Help,Cfg.Ext1,$0F)
Title
        .byte 'Boot partition',0
Help
        .byte 'Internal HDD boot partition',0
        .endl
```

Here, value 0-15 is stored at Config.User2 and is encoded in bits 0-3 or Cfg.Ext1.

PartNum

Identical to DriveNum but using one extra value (16) which denotes ‘On disk’. Example:

```
        .local Item49
@       dta Item[0] (5,Item50,Title,ItemType.PartNum,Config.User2,Help,Cfg.Ext1,$1F)
Title
        .byte 'Boot partition',0
Help
        .byte 'Internal HDD boot partition',0
        .endl
```

Again, Config.User2 holds the value, but now requires 5 bits.

Date

Presents a means of adjusting the system data and is therefore unlikely to be used in plugin code (since the BIOS already uses the same item type in menu 1). Example:

```
        .local Item7
@       dta Item[0] (1,Item8,Title,ItemType.Date,Config.Date,0,0,0)
Title
        .byte 'System Date',0
        .endl
```

Time

Similar to the Date type, but for time of day adjustment.

Spinner

This item type presents a numeric value within a specified range through which the user may ‘spin’ forwards and backwards in single unit increments and decrements. Example:

```
        .local Item37
@       dta Item[0] (4,ItemClick,Title,ItemType.Spinner,Config.Colour,Help,Cfg.Aux6,$0F)
        .byte $00       ; minimum value
        .byte $0F       ; maximum value
Title
        .byte 'Colour',0
Help
        .byte 'Choose colour scheme',0
        .endl
```

In this example, the minimum value is 0 and the maximum value $0F (15). The value therefore occupies 4 bits and is in this case stored in *Config.Colour* and encoded in bits 0-3 of *Cfg.Aux6*.

String

The string type simply writes a string to the menu and performs no further action. It is used extensively in the system information menu. Example:

```
        .local Item25
@       dta Item[0] (3,Item26,Title,ItemType.String,txtBIOSVersion,0,0,0)
Title
        .byte 'BIOS version',0
txtBIOSVersion
        .byte '%x.%02x',0
        .word Version.Maj,Version.Min
        .endl
```

Initialising Menu Items

Once new menu items have been declared, they will automatically appear and be editable as soon as their containing menu is opened, and their values will be retrieved from and saved to NVRAM. But what about menu items which need to be dimmed or disabled depending on discovered hardware or the state of some other setting? Fortunately – as well as being able to probe hardware registers – plugins can run code prior to their menu items being displayed and can react to user edits. For example: if your plugin includes a menu item which sets the boot partition of the hard disk, you would likely want the boot partition setting to be greyed out if the user completely disabled the hard disk, and enabled again when the hard disk was reactivated.

Below is the menu structure of the default plugin:

```
PluginMenu
        .local StereoToggle
@       dta Item[0] (1,CovoxToggle,Title,ItemType.OnOff,StereoPokey,Help,Cfg.Aux,$01)
Title
        .byte 'Stereo Pokey',0
Help
        .byte 'Enable/disable stereo audio',0
        .endl

        .local CovoxToggle
@       dta Item[0] (1,Device3Toggle,Title,ItemType.OnOff,CovoxFlag,Help,Cfg.Aux,$02)
Title
        .byte 'Covox',0
Help
        .byte 'Enable/disable Covox',0
        .endl


        .local Device3Toggle
@       dta Item[0] (5,Device4Toggle,Title,ItemType.OnOff,Device3,Help,Cfg.Aux,$04)
Title
        .byte 'Device 2',0
Help
        .byte 'Enable/disable Device 2',0
        .endl


        .local Device4Toggle
@       dta Item[0] (5,SysInfoStereo,Title,ItemType.OnOff,Device4,Help,Cfg.Aux,$08)
Title
        .byte 'Device 3',0
Help
        .byte 'Enable/disable Device 3',0
        .endl


        .local SysInfoStereo
@       dta Item[0] (3,0,Title,ItemType.List,StereoPokeyFlag,0,0,0)
        .byte 2
        .word List1
        .word List2
Title   .byte 'Audio hardware',0
List1   .byte 'Mono',0
List2   .byte 'Stereo',0
        .endl
```

The list comprises a Stereo Pokey enable/disable item, Covox enable/disable, two place holder items for the remaining two pins of P2, and an entry which will appear at the foot of the System Information menu. Before any of the items are displayed, we need to:

1. Establish whether Stereo Pokey under software control and grey out the stereo on/off option if not.
2. Grey out the item in the System Information screen (since no items there are ever selectable)

To initialise menu items, place the address of your setup routine at $FB1E (offset $001E in the plugin file header). Below is an example of a populated header from the default plugin:

```
        .byte 'ULPLUGIN'              ; plugin signature (8 bytes)
        .byte $00,$54                 ; host BIOS minimum version (major, minor)
        .byte 'Stereo/Covox',0,0,0,0  ; plugin name (16 bytes)
        .byte $00,$01                 ; plugin version (major, minor, BCD) (2 bytes)
        .byte 'SC'                    ; extension signature (Stereo and Covox)
        .word Init                    ; pointer to menu setup routine
        .word Update                  ; pointer to menu update routine
        .word HardwareTest            ; pointer to the plugin's hardware test
        .word HardwareSetup           ; pointer to the plugin's hardware setup

        .byte 'Device Control',0      ; title of plugin menu

        .align PlugInMenuAddress      ; pad to first menu node
```

In this example, the menu initialisation code is at *Init,* and the code at that address will be run before the menus are first displayed and before menus are drawn when moving from one menu to the next. Here's the setup code:

```
//
//      Menu initialisation
//

        .proc Init
        ldax #SysInfoStereo    ; point to the audio item on the System Information menu
        ldy #1                 ; just one item to deactivate
        jsr pDeactivateItems

        bit StereoControlFlag  ; is stereo under software control?
        bmi @+

        lda StereoPokeyFlag    ; if we can't control stereo, make option reflect fixed status
        sta StereoPokey
        ldax #StereoToggle     ; and deactivate the menu item
        ldy #1
        jsr pDeactivateItems
@
        rts
        .endp
```

Here, we deactivate the audio hardware item in the System Information menu, then check whether stereo is under software control and if it isn't, we grey out the stereo on/off option.

There are three library routines we can call to affect the state of menu items.

pDeactivateItems

This call dims or "greys out" menu items and expects the address of the first target menu item in A,X (LSB in A, MSB in X) and the number of consecutive menu items to affect in Y. A full menu redraw is performed.

pActivateItems

This call enables previously dimmed or Disabled items and makes them selectable again. Arguments are identical to *pDeactivateItems*. A menu redraw is performed.

pDisableItems

This call not only dims a menu item, but replaces the text with "Disabled" and redraws the menu. Arguments are identical to the previous two functions. This function is useful if you wish to make it clear than an option not only has a fixed, uneditable state, but is completely deactivated. For example, you might wish to replace the boot partition ID of a hard disk with 'Disabled' if the hard disk itself is deactivated, without overwriting the actual boot drive setting stored in NVRAM. A disabled setting's value is not lost when the item is disabled, and will reappear as soon as the item is re-enabled.

Note that the above three calls must be invoked every time the containing menu is opened, even if the state of the target item does not change while the setup menu is open. This is because the setup

utility is obliged to use as little RAM as possible and does not maintain a state flag for every editable item in the menu system. Instead, it maintains a short list of a maximum of twelve item states and rebuilds this list every time a menu opens.

So how do we know whether stereo is under software control? The BIOS calls the plugin's hardware test early on (before the menus are even open), and it's here that the plugin can probe hardware and – for example – establish whether the M0 pin of the U1MB P2 header is connected to the Stereo Pokey's on/off switch.

```
//
//      Test Stereo Pokey control
//

        .proc HardwareTest
        lsr StereoControlFlag
        mva #0 UltimateAux     ; attempt to disable stereo
        jsr DetectStereoPokey  ; test for stereo
        beq @+
NotActive                      ; if it's still enabled, it's present but not controlled by M0
        rts
@
        mva #$01 UltimateAux   ; try to enable stereo
        jsr DetectStereoPokey  ; test it again
        beq NotActive          ; if there's still no stereo, there's none present
        sec
        ror StereoControlFlag  ; otherwise set flag saying stereo control works
        rts
        .endp
```

Here we use bit 7 of an internal variable (*StereoControlFlag*) to denote whether stereo hardware is under software control. We first clear bit 7 of the flag, then store 0 in *UltimateAux* and call our stereo Pokey test. If stereo hardware is under the control of M0, we should find no stereo hardware (our test returns Z=1 if no stereo hardware is found). If we do detect stereo hardware when M0=0, we can deduce that stereo hardware is present but not switchable in software. If we find no stereo hardware regardless of the value of M0, then we conclude there's no stereo hardware present at all.

Updating Menus

Aside from specifying which menu items are enabled or disabled prior to the menus being drawn, we may also need to respond to changes while the user is actually changing options. We handle this via code pointed to by the *Update* vector. In this case, we are passed the current menu number in the Y register, and the address of the current menu item in the A,X register pair. We can therefore check A,X to see if the user's selection changes something which has ramifications elsewhere. When terminating our update routine, we should set the carry flag if changes require a redraw of the complete menu (this will be needed if we explicitly changed the state of another menu item).

Say, for example, we have two mutually exclusive menu items: *Item1* and *Item2.* Both are on/off items, and if one is enabled, the other must always be disabled, and both may be simultaneously disabled. The *Update* code would look like this:

```
        .proc Update
        ldy #0                 ; pre-load Y
        cpax #Item1            ; has the user changed item 1?
        bne NotItem1
        lda Config1            ; If we turn on Config1, force Config2 off and redraw menu
        beq Done
        sty Config2
        sec                    ; set carry to tell BIOS to redraw menu
        rts
NotItem1
        cpax #Item2
        bne NotItem2
        lda Config2
        beq Done
        sty Config1
        sec
        rts
```

```
NotItem2
        clc                     ; nothing needs to be redrawn aside from the edited item, so
return with carry clear
        rts
        .endp
```

In the example code, we pre-load the Y register with 0 for convenience (since we have no use for the menu number here), and test (by comparing A,X) whether the user has changed the state of Item1 or Item2. If so, we check the item's current value and reset the opposing value if both are "on" and then redraw the menus (by setting the carry flag prior to RTS).

As a second example, we will disable the HDD boot drive if the HDD itself is disabled.

```
        .byte 'ULPLUGIN'              ; plugin signature (8 bytes)
        .byte $00,$54                 ; host BIOS minimum version (major, minor)
        .byte 'Example',0             ; plugin name (16 bytes)
        .byte 0,0,0,0,0,0,0,0
        .byte $00,$01                 ; plugin version (major, minor, BCD) (2 bytes)
        .byte 'EX'                    ; extension signature (Stereo and Covox)
        .word Init                    ; pointer to menu setup routine
        .word Update                  ; pointer to menu update routine
        .word HardwareTest            ; pointer to the plugin's hardware test
        .word HardwareSetup           ; pointer to the plugin's hardware setup
        .byte 'Hard Disk Setup',0     ; title of plugin menu
        .align PlugInMenuAddress      ; pad to first menu node

        .proc HDD
@       dta Item[0] (5,BootDrive,Title,ItemType.OnOff,HDDFlag,Help,Cfg.Ext1,$80)
Title
        .byte 'Hard disk',0
Help
        .byte 'Enable/disable hard disk',0
        .endp


        .proc BootDrive
@       dta Item[0] (2,0,Title,ItemType.PartNum,BootDrive,Help,Cfg.Ext1,$1F)
Title
        .byte 'Boot partition',0
Help
        .byte 'Select boot partition',0
        .endp

        .proc HardwareTest
        rts
        .endp

        .proc HardwareSetup
        rts
        .endp

        .proc Init
        jmp TestHDDOnOff      ; initialise menu before it is rendered
        .endp

        .proc Update          ; react to edits
        cpax #HDD             ; has the user changed HDD?
        beq TestHDDOnOff
        clc                   ; user changed something else, so exit without redraw
        rts
        .endp

        .proc TestHDDOnOff
        ldax #BootDrive       ; pre-load A,X
        ldy HDDFlag           ; check hard disk
        bne HDDEnabled
        ldy #1
        jmp DisableItems      ; if HDD is off, disable boot drive (DisableItems forces menu
redraw)
HDDEnabled
        ldy #1
        jmp ActivateItems     ; if HDD is on, activate boot drive (ActivateItems forces menu
redraw)
        .endp
```

Note that in the example the *Init* routine calls *TestHDDOnOff* to ensure the 'Boot partition' item has the correct state when the menu is initially opened and before the user has made any changes. In the initialisation context, A,X and Y have no meaning on entry and the carry flag is ignored on exit.

Writing to Hardware Registers

While most plugin code seeks to control hardware via the Aux signals of the U1MB's P2 header (which is transparently handled by the BIOS itself), there may be occasions when it's necessary to control hardware via internally mapped registers on every system reset (and every time the user exist the BIOS setup menu). A hook is provided for precisely this purpose: the pointer to the hardware setup routine.

The hardware setup routine vector should point to code you want to execute every system reset and every time the user leaves the BIOS setup utility. Depending on how the user has configured the hardware supported by the plugin, hardware registers in the IO region may be updated accordingly. Care must be exercised, however, if the target hardware registers occupy the same address space as the Ultimate 1MB IO RAM. The plugin will then need to toggle IO RAM in the $D1xx, $D5xx and $D600-$D7FF address space. IO RAM is controlled via bit 6 of $D380, with 1 enabling RAM and 0 disabling it (and exposing any hardware registers in the same address space).

For example, to update a hardware register at $D600 with an internal flag:

```
lda User1
ldy #0
sty UltimateConfig
sta $D600
ldy #$40
sty UltimateConfig
```

The *HardwareSetup* routine will be run on system Reset and every time the BIOS setup utility is exited.

Acknowledgements

The author would like to thank the following:

- Sebastian Bartkowicz (Candle O'Sin) for Ultimate 1MB and Incognito hardware and the opportunity to contribute to the original PBI firmware
- Matthias Reichl (Hias) for permission to adapt his High-Speed SIO code for use in the new PBI BIOS, for his invaluable technical insight, and for his help with debugging and troubleshooting
- Avery Lee (Phaeron) for the peerless Altirra emulator, for the indispensable Altirra Hardware Reference Manual and his detailed technical insights
- Kuba Skrzypnik for enthusiastic testing and user feedback
- Paul Fisher (Mr Fish) for his suggestions regarding user interface design
- Lotharek for providing further Ultimate 1MB boards
- Marius Diepenhorst (ProWizard) for making himself available to test the new Ultimate 1MB firmware almost incessantly for the past nine months, sometimes spending whole consecutive evenings testing new builds and then reporting bugs and suggesting new features. Your dedicated support and enthusiasm for all things Atari is truly remarkable.
- Marcin Sochacki (TheMontezuma) for SIO2BT hardware and documentation, and for guidance provided when SIO2BT support was being implemented.

Last but not least, I must thank everyone on the AtariAge and AtariArea forums who took the time to download and test the new firmware, report issues and make suggestions – among them Kyle22, rdea6, Stephen, Panther, MacRorie, Morelenmir and many others.

Jonathan Halliday

June 2016