Ultimate 1MB/Incognito/1088XEL|U1MB

BIOS Technical and Developer Documentation

By Jonathan Halliday

Second Edition

Revised 10/05/2018

Revised to cover main BIOS v.2.0, PBI BIOS v.2.0, and XEX Loader v.2.0.

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 replacement BIOS provides a powerful alternative to the stock firmware written by Sebastian Bartkowicz (Candle), who also designed and produced both devices. 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, configuration profiles, hardware detection, diagnostic cart boot, and plug-in modules). A new PBI BIOS and XEX loader were also 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.

The BIOS Initialisation Process

When the Atari computer is booted or reset, the U1MB or Incognito BIOS ROM is mapped at 0xC000-CFFF and 0xD800-FFFF, and the CPU begins execution at the address pointed to by the machine Reset vector at 0xFFFC. Upon reset, the Ultimate configuration register at 0xD380 is unlocked, allowing changes to the configuration until bit 7 of 0xD380 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 0xD000-D7FF to the Self-Test area at 0x5000. 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**
0x30-35	Pointers		Pointers
0x38-3B	Pointers		Pointers
0x0200	VDSLST		VDSLST
0x0222	VVBLKI		VVBLKI
0x0100-0180	OS entry code and buffers		OS entry code and buffers
0xC000-C1FF	BIOS metadata and jump table		BIOS metadata and jump table
0xC200-CFFF	BIOS code		BIOS code
0xD000-D7FF	BIOS code	IO RAM	IO RAM
0xD800-DFFF	BIOS code		
0xE000-E3FF	OS Character Set (for Turbo Freezer)		
0xE400-FAFF	BIOS code		BIOS code
0xFB00-FEFF	BIOS plug-in code		
0xFF00-FFBF	Editable ROM slot descriptions		Editable ROM slot descriptions
0xFFC0-FFDF	System ROM space descriptions		System ROM space descriptions
0xFFE0-FFF9	Unused		Unused
0xFFFA-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 (0x30-35 and 0x38-3B) are designated for external device (PBI) use, while VDSLST (0x0200) and VVBLKI (0x0222) are reinitialised by the OS on system reset. Usage of the lower half 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
0xC000	6	Firmware signature (ASCII)	'ULBIOS'	'INBIOS'
0xC006	1	Major revision (BCD)		
0xC007	1	Minor revision (BCD)		
0xC008	3	Revision date (DD/MM/YY)		
0xC00B	1	SpartaDOS X ROM Size in 8KB banks	0x18, 0x20, or 0x28	0x18, 0x20, or 0x28
0xC00C	2	Editable slot name base address	0xFF00	0xFF00
0xC00E	2	Address of NVRAM configuration buffer		
0xC010	2	Address of internal option table		
0xC012	3	JMP to menu item attribute setting routine		
0xC015	3	JMP to message print routine		
0xC018	3	JMP to get user confirmation 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 0x18, 0x20 or 0x28 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 considerable 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	Address	Type
0xFF00	OS slot 0	0xFF60	BASIC slot 2
0xFF10	OS slot 1	0xFF70	BASIC slot 3
0xFF20	OS slot 2	0xFF80	XEGS slot 0
0xFF30	OS slot 3	0xFF90	XEGS slot 1
0xFF40	BASIC slot 0	0xFFA0	XEGS slot 2
0xFF50	BASIC slot 1	0xFFB0	XEGS slot 3

Incognito ROM description fields are as follows:

Address	Type	Address	Type
0xFF00	Colleen OS slot 0	0xFF40	XL/XE OS slot 0
0xFF10	Colleen OS slot 1	0xFF50	XL/XE OS slot 1
0xFF20	Colleen OS slot 2	0xFF60	XL/XE OS slot 2
0xFF30	Colleen OS slot 3	0xFF70	XL/XE OS slot 3

NVRAM Configuration Layout

The new BIOS uses addresses 0x20-2E and 0x40-6D of the U1MB's SD1305's NVRAM. The U1MB/Incognito build of SpartaDOS X uses bytes 0x30-31, so the entire 0x3x address range is deliberately avoided by the BIOS. The Incognito BIOS uses slightly fewer bytes, but 0x20-2F and 0x40-6F should now be considered reserved for U1MB and Incognito. Moreover, addresses 0x50-55 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 (0x7x).

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**	
0x20	14	12	Configuration profile 1
0x2E	1	1	Profile number (0-2)
0x40	14	12	Configuration profile 2
0x50	6	6	XEX loader configuration
0x60	14	12	Configuration profile 3

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)
	5-6	SIO device selection (0 = D1-D4, 1 = Disks+PCLink, 2 = 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 = 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	D1: Redirect (0 = disabled, 1 = enabled)
+7 (Aux7)	0-3	CONFIG.SYS drive (0 = disabled, 1-15 = drive number)
	4	“Z:” CIO Handler (0 = disabled, 1 = enabled)
	5	Internal BASIC default (0 = enabled, 1 = disabled)
	6-7	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)
	5-6	SIO device selection (0 = D1-D4, 1 = Disks+PCLink, 2 = 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 = HSIO, 1 = SIO2BT, 2 = 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	D1: Redirect (0 = disabled, 1 = enabled)
+9 (Aux7)	0-3	CONFIG.SYS drive (0 = disabled, 1-15 = drive number)
	4	“Z:” CIO Handler (0 = disabled, 1 = enabled)
	5	Internal BASIC default (0 = enabled, 1 = disabled)
	6-7	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 0xFB00. Plugins may be inserted into the BIOS ROM image at file offset 0x3B00 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:

- Define editable menu items in any of the eight BIOS setup menus
- Provide a custom title for the 'Device Control' menu
- Save and retrieve state of said menu items in the U1MB NVRAM
- Activate, deactivate ('grey out') and temporarily disable newly defined menu items
- Establish whether a specific hardware device is controlled by U1MB via the Aux signals
- Toggle the state of a hardware device controlled by Aux signals
- 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 72-byte metadata header arranged thus:

Address	Offset	Size (Bytes)	Description	Content
0xFB00	0x0000	8	Plugin signature (8 ATASCII bytes)	'ULPLUGIN'
0xFB08	0x0008	1	Host BIOS minimum major revision number (BCD)	
0xFB09	0x0009	1	Host BIOS minimum minor revision number (BCD)	
0xFB0A	0x000A	2	Mnemonic plugin ID (2 ATASCII bytes)	
0xFB0C	0x000C	16	Plugin name (NUL terminated ATASCII string)	
0xFB1C	0x001C	1	Plugin major revision number (BCD)	
0xFB1D	0x001D	1	Plugin minor revision number (BCD)	
0xFB1E	0x001E	1	Plugin Revision Day (BCD)	
0xFB1F	0x001F	1	Plugin Revision Month (BCD)	
0xFB20	0x0020	1	Plugin Revision Year (BCD)	
0xFB21	0x0021	2	Pointer to menu setup routine	
0xFB23	0x0023	2	Pointer to menu update routine	
0xFB25	0x0025	2	Pointer to plugin's hardware test routine	
0xFB27	0x0027	2	Pointer to plugin's hardware setup routine	
0xFB29	0x0029	31	Menu title (NUL terminated ATASCII string)	
0xFB48	0x0048	-	First node of plugin menu structure	

Plugin Signature

Plugin modules must begin with the 8-byte string 'ULPLUGIN' in order to be recognized by the flashing tool.

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.

Mnemonic Plugin ID

This is a descriptive signature. The default plugin ID is 'SC' (denoting Stereo/Covox). This ID 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.

As of BIOS v.1.36, plugin configuration bits in NVRAM are automatically cleared if the BIOS discovers during the boot process that the plugin ID in NVRAM does not match the ID in the BIOS ROM. This can happen whenever a BIOS with a different plugin is flashed to ROM, or when a plugin module is merged into an existing BIOS. This check guarantees that plugin configuration data is zeroed (and therefore not out of bounds) the first time a new plugin becomes active.

Plugin Name

The plugin name describes the general purpose of the plugin. For example, the default plugin name is ‘Stereo/Covox’. The name should be NUL terminated and should not exceed 16 bytes (including terminator).

Plugin Revision Date

The plugin revision date is stored as three BCD numbers (DD/MM/YY).

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 NUL terminated string of not more than 31 characters (including termination byte) which will be displayed as the title of the 'Device Control' menu. For example, the KMK/JZ plugin titles the menu 'IDEa APT Hard Disk'.

Plugin RAM

Sixteen bytes of IO RAM at 0xD7F0-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.

Plugin Menu Items

The menu structure of the new BIOS is 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 menu which is about to be opened. Menus may contain no more than fourteen items. Menu items which overrun this quota will be ignored.

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
      CfgOff        .byte ; byte offset into config buffer
      CfgMask       .byte ; config buffer bitmask
.ends
```

Note: as of BIOS v.1.36, the 'Help' pointer (which originally linked to a NUL terminated explanatory string describing the menu option) has been removed. Needless to say, this breaks compatibility with older plugins, although aside from those the author wrote himself, I don't think any incompatible plugins exist.

A description of fields follows.

Menu

Number of menu in which item appears (0-7; if you add items to the Device Control menu, use 6).

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.

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 the 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
       ToggleList    ; list comprising only two items (binary list)
       DriveNum      ; drive number spinner (0 = Off, 1-15 = D1:-DO:)
       PartNum              ; drive number spinner + 16 = "APT setting" (FDISK
value)
       Date          ; three 8-bit ints
       Time          ; three 8-bit ints
       Spinner              ; numeric spinner
       String        ; string
.ende
```

A description of types follows.

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,Cfg.Aux,0x01)
Title
        .byte 'Stereo Pokey',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’. 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,Cfg.Aux,0x80)
Title
        .byte 'Flash writes',0
        .endl
```

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'.

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,Cfg.Config,0x03)
        .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
       .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.

ToggleList

This item type defines a list comprising only two items. The reason for this item type is to distinguish between multi-item lists and those with only a binary state. This distinction informs the help gloss displayed at the foot of the menu which explains the navigation keys.

```
       .local Item1
@      dta Item[0] (0,Item2,Title,ItemType.ToggleList,Config.VBXE,Cfg.Config,0x03)
       .byte 2             ; note: the list item count is ignored, but MUST be
present
       .word List1
       .word List2
Title
       .byte 'VBXE Address',0
List1
       .byte '0xD640',0
List2
       .byte '0xD740',0
       .endl
```

Note that although the list item count is superfluous, it still must be supplied. The value will be ignored. The toggle list control appears to the user as a toggle option, but with software defined options (instead of 'Enabled' and 'Disabled').

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,Cfg.Ext1,0x0F)
Title
       .byte '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,Cfg.Ext1,0x1F)
Title
```

```
       .byte '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)
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 (decimal, 0-99) 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,Cfg.Aux6,0x0F)
       .byte 0x00    ; minimum value
       .byte 0x0F    ; maximum value
Title
       .byte 'Colour',0
       .endl
```

In this example, the minimum value is 0 and the maximum value 0x0F (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*.

The maximum value of 99 is a consequence of the value being internally converted to BCD before being displayed. If you need to represent values higher than 99, use a list control.

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)
Title
       .byte 'BIOS version',0
txtBIOSVersion
       .byte '%x.%02x',0
       .word Version.Maj,Version.Min
       .endl
```

Formatted Printing

As can be seen from some of the prior examples, literal strings may contain formatted output. The BIOS setup tool's internal *printf* function is similar to that found in most C implementations, albeit somewhat simplified and with a reduced selection of output options. Formatting directives appearing in the string should have a corresponding 16-bit pointer (word) appearing immediately after the NUL terminator of the string. If there are several formatting directives in the string, each must have a corresponding data pointer.

Formatting directives take the following format:

%[width]type

Width is an optional decimal number which describes the field width, while *Type* is one of the following:

Character	Data type
c	Character
p	Pointer to string
s	String
x	8-bit Hex (or BCD) value

Note that output is always right justified if the field width is specified, otherwise output is left justified.

Examples

The following are examples of formatted output, derived from the actual BIOS source code.

Pointer (%p)

```
        lda Config.BIOSKey
        asl @
        tax
        lda BIOSKeyTable,x
        sta Num1
        lda BIOSKeyTable+1,x
        sta Num1+1
        jsr pMessage
        .byte 'Press %p for Setup',0
        .word Num1    ; pointer to string
        rts
```

Note: The printf function is described later in this document.

Character (%c) and String (%s)

```
        .local Item43
@       dta Item[0] (7,Item44,Title,ItemType.Action,SaveSettings-1,0,0)
Title
        .byte '%s %16c[S]',0       ; repeat space character sixteen times
        .word txtSaveChanges,SpaceChar
        .endl

SpaceChar
```

```
       .byte $20    ; space character

txtSaveChanges
       .byte ‘Save changes’,0
```

Note that string formatting is not recursive, so in the example above, *txtSaveChanges* may not itself contain more formatted output directives. In the example, note also that the actual text of *txtSaveChanges* appears in several different menu items, and that it is included in the parent strings by means of the *%s* directive as a method of primitive string tokenisation (i.e. only a single example of the substring common to all the ‘parent’ strings exists).

Hex/BCD (%x)

```
       .local Item27
@      dta Item[0] (4,ItemVideo,Title,ItemType.List,CPUType,0,0)
       .byte 3
       .word List1
       .word List2
       .word List3

Title  .byte 'Processor',0
List1  .byte '6502C %x.%xMHz',0
       .word CPUFreqMaj,CPUFreqMin
List2  .byte '65C02 %x.%xMHz',0
       .word CPUFreqMaj,CPUFreqMin
List3  .byte '65C816 %x.%xMHz',0
       .word CPUFreqMaj,CPUFreqMin
       .endl
```

Here, *CPUFreqMaj* and *CPUFreqMin* have been pre-initialised as BCD values.

Initialising Menu Items

Menu items appear active and editable when 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.

The menu structure of the default plugin is shown below:

```
PluginMenu
       .local StereoToggle
@      dta Item[0] (1,CovoxToggle,Title,ItemType.OnOff,StereoPokey,Cfg.Aux,0x01)
Title
       .byte 'Stereo Pokey',0
       .endl

       .local CovoxToggle
@      dta Item[0] (1,Device3Toggle,Title,ItemType.OnOff,CovoxFlag,Cfg.Aux,0x02)
Title
       .byte 'Covox',0
       .endl


       .local Device3Toggle
@      dta Item[0] (5,Device4Toggle,Title,ItemType.OnOff,Device3,Cfg.Aux,0x04)
Title
       .byte 'Device 2',0
       .endl


       .local Device4Toggle
@      dta Item[0] (5,SysInfoStereo,Title,ItemType.OnOff,Device4,Cfg.Aux,0x08)
Title
       .byte 'Device 3',0
       .endl


       .local SysInfoStereo
@      dta Item[0] (3,0,Title,ItemType.List,StereoPokeyFlag,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 0xFB1E (offset 0x001E in the plugin file header). Below is an example of a populated header from the default plugin:

```
       .byte 'ULPLUGIN'            ; plugin signature (8 bytes)
       .byte 0x02,0x00             ; host BIOS minimum version (major, minor) (2
bytes)
       .byte 'SC'                  ; extension signature (Stereo and Covox) (2 bytes)
       .byte 'Stereo/Covox',0,0,0,0      ; plugin name (16 bytes)
       .byte 0x01,0x01                   ; plugin version (major, minor, BCD) (2
bytes)
       .byte 0x02,0x01,0x18              ; plugin revision date (BCD: DD,MM,YY) (3
bytes)
       .word Init                  ; pointer to menu initialisation 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. Since menu items are stored in ROM, it's not possible to alter flags inside of menu item definitions. Instead, we call *pSetItemFlags* with the address of the menu item we want to change in the X and Y registers (X = LSB, Y = MSB), flags in the upper nibble (bits 4-7) of the accumulator, and the number of consecutive items to process, minus 1, in the lower nibble of the accumulator.

Bit usage in the accumulator is as follows:

7	6	5	4	3	2	1	0
Attribute flags				Number of items, less 1			
Internal Selection bit	Item State						

The *Selected* attribute is for internal use and should never be explicitly set. *Item state* is ignored unless the *Dimmed* bit is set. When *Dimmed* is 1, *Item State* may take on of the following values:

Bit	6	5	4	Value	Effect
	0	0	0	0x00	Item is displayed with current setting and in an active state
	1	0	0	0x40	Item is dimmed and inactive
	1	0	1	0x50	Item is dimmed and displayed as "Disabled"
	1	1	0	0x60	Item is dimmed and displayed as "Enabled"
	1	1	1	0x70	Item is dimmed and displayed as "Not Present"

By default, every time a menu is about to be drawn on the screen, the attributes of all of its menu items are set to 0x00 (active). However, since the menu initialisation function is called immediately before the menu is actually rendered, we get the opportunity to customise menu item states before the menu is actually drawn.

To set the attribute of a single menu item, we need simply load the address into XY and the desired attribute into A. But if we wish to process several contiguous items at once (for example, if we want to deactivate a group of menu items when the user switches off a particular setting on which all the target items depend), we add *NumItems-1* to the attribute value in A. Obviously this means that a maximum of sixteen items can be affected via a single call to *pSetItemFlags*, but this isn't really a problem since no single menu may contain more than fourteen items.

To help, the plugin equate file defines enumerated variables for menu item attributes:

```
.enum  ItemState
       Active        = 0x00
       Dimmed        = 0x40
       Disabled      = 0x50
       Enabled              = 0x60
       NotPresent    = 0x70
.ende
```

You need simply load one of these values into A and (optionally) add to it the number of items to be processed, less 1.

For example, the System Information menu consists of a series of items, all greyed out and inactive (since their purpose is simply to present information). The BIOS's menu initialisation routine accomplishes this via the following code:

```
       ldxy #Item24               ; first menu item in group
       lda #ItemState.Dimmed + 7  ; 0x40 + number of items less 1, so eight items
       jsr SetItemFlags           ; set the attributes
```

Similarly, when the user activates or deactivates the PBI BIOS, several items which follow become activated or deactivated. Here is the code which handles it:

```
       ldxy #Item14               ; if PBI is off, grey out everything else
       lda Config.PBI
       bne PBIisON
       lda #ItemState.Disabled + 8      ; 9 items
       jsr SetItemFlags
       ldxy #ItemSIO
       lda #ItemState.Disabled + 6      ; 7 items
       jsr SetItemFlags
       ldxy #ItemBASIC
       lda #ItemState.Enabled
       jmp SetItemFlags
PBIisON                                 ; if PBI is on, activate device ID and main
HiSIO
       lda #ItemState.Active + 1  ; 2 items
       jsr SetItemFlags
       ldxy #ItemSIO
       lda #ItemState.Active
       jsr SetItemFlags           ; turn on HiSIO
       jsr TestHiSIOOnOff         ; set up HiSIO items for drives 1-4
       ldxy #Item15
       lda #ItemState.Active
       jsr SetItemFlags
       jmp TestHDDOnOff
```

When the PBI BIOS is disabled, nine following items and all the items on the 'SIO and CIO Drivers' menu are greyed out and marked 'Disabled', regardless of their actual settings. This allows the current settings to reappear when the PBI BIOS is enabled again, since they never actually changed (only the PBI selection bit changed).

Here's the menu initialisation code from the default (Stereo/Covox/Soundboard) plugin:

```
//
```

```
//     Menu initialisation
//

       .proc Init
       bit SoundBoardControl            ; if there's no soundboard
       bmi @+
       ldxy #SoundBoardToggle
       lda #ItemState.Disabled    ; Dim and display "Disabled"
       jsr pSetItemFlags
@
       ldxy #SysInfoSoundBoard
       lda #ItemState.Active
       bit SoundBoardControl
       bmi GreyOut
       ldy #ItemState.NotPresent  ; Dim and display "Not Present"
       jsr pSetItemFlags
       jmp @+
GreyOut
       jsr Deactivate                   ; Grey out existing text
@

       ldxy #SysInfoStereo
       jsr Deactivate
       bit StereoControlFlag            ; prepare menu before it's displayed
       bmi @+

       lda StereoPokeyFlag        ; if we can't control stereo, make option reflect
fixed status
       sta StereoPokey
       ldxy #StereoToggle
       jsr Deactivate
@
       rts
       .endp




//
//     Helper routine: Deactivate 1 menu item at XY
//

       .proc Deactivate
       lda #ItemState.Dimmed
       jmp pSetItemFlags
       .endp
```

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 to settings made by the user while the menu is open. We handle this via code pointed to by the *Update* vector. In this case, we are passed the current menu number in the A register, and the address of the current menu item in the XY register pair. We can therefore check XY to see if the user's selection changes something which requires other items on the same menu to immediately change their value or state. When changing an item's value, 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). If we exit with the carry flag clear, nothing else on the menu will be redrawn, even if we changed the state of other items.

If our update routine changes a menu item's *state*, on the other hand (if it deactivates an item or sets it to 'Disabled', etc, by calling *pSetItemFlags*), we don't need to worry about the state of the carry flag, since *pSetItemFlags* implicitly causes a complete menu redraw.

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
	lda #0		; prime A with 0 so we can quickly turn settings off
	cpxy #Item1	; has the user changed item 1?
	bne NotItem1
	ldy Config1	; If we turn on Config1, force Config2 off and redraw 
menu
	beq Done
	sta Config2
	sec		; set carry to tell BIOS to redraw menu
	rts
NotItem1
	cpxy #Item2
	bne NotItem2
	ldy Config2
	beq Done
	sta 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 A register with 0 for convenience (since we have no use for the menu number here), and test (by comparing XY) whether the user has changed the state of Item1 or Item2. If so, we check the corresponding 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 0x02,0x00			; host BIOS minimum version (major, minor)
	.byte 'EX'			; extension signature (Stereo and Covox)
	.byte 'Example',0		; plugin name (16 bytes)
	.byte 0,0,0,0,0,0,0,0
	.byte 0x00,0x01			; plugin version (major, minor, BCD) (2 
bytes)
	.byte 0x02,0x01,0x18		; plugin revision date
```

```
       .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,Cfg.Ext1,0x80)
Title
       .byte 'Hard disk',0
       .endp


       .proc BootDrive
@      dta Item[0] (2,0,Title,ItemType.PartNum,BootDrive,Cfg.Ext1,0x1F)
Title
       .byte '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
       cpxy #HDD                  ; has the user changed HDD?
       beq TestHDDOnOff
       clc                        ; user changed something else, so exit without
redraw
       rts
       .endp

       .proc TestHDDOnOff
       ldxy #BootDrive            ; pre-load A,X
       lda HDDFlag                ; check hard disk
       bne HDDEnabled
       lda #ItemState.Disabled    ; if HDD is off, disable the boot drive setting
       jmp pSetItemFlags
HDDEnabled
       Lda #ItemState.Enabled
       jmp psetItemFlags          ; 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. Note also that no information is passed to the *Init* routine in XY or A.

Displaying Messages and Interacting with the User

So far we've talked solely about managing menus, but there are two final forms of user interaction possible via plugins: status line messages and confirmation dialogs.

Displaying a Message

To display a message on the status line, use the following code:

```
        jsr pMessage
        .byte 'My message',0
```

The message will auto-clear after 2-3 seconds, being replaced by the standard context-sensitive help text. See the earlier section concerning formatted printing for information on how to include numbers and other information in your string.

Obtaining a Response from the User

It's possible to display 'Buttons' in the status line which the user can select via the cursor keys and the Return/Esc keys or via the joystick. To display buttons and allow the user to choose one, you should load the accumulator with the button mask, the X register with the default button, and JSR to *pConfirm.* A prompt string (NUL terminated) should immediately follow the call to *pConfirm*; this string will be displayed to the left of the buttons and a question mark will be automatically appended to it. Upon return, the accumulator will hold the sequential value of the button chosen by the user (starting at 0), and correspondingly the Z flag will be set if the user chose the first button in the selection. Meanwhile, if the user cancelled selection by pressing the Escape key, the Z flag will be zero and the N flag will be set.

The button masks are enumerated as follows:

```
.enum cmdButton
        OK     = 1
        Yes    = 2
        No     = 4
        Cancel = 8
.ende
```

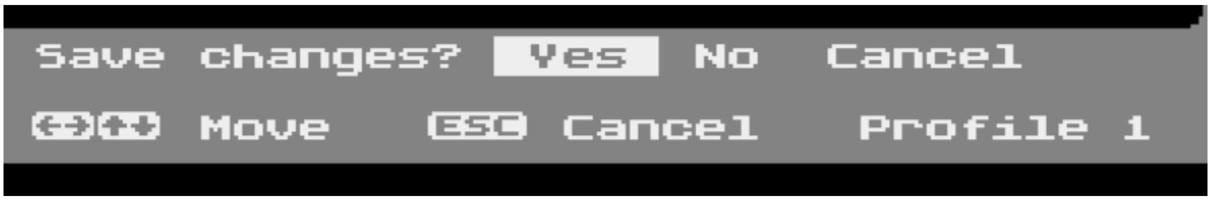
Numbers are assigned to buttons in accordance with their enumerated bit order, so if only 'Yes' and 'No' buttons are displayed, 'Yes' will always return 0 and 'No' will return 1.

For example, to present 'Yes', 'No' and 'Cancel' buttons with the default action being 'Yes', and then act on the response:

```
        lda #cmdButton.Yes + cmdButton.No + cmdButton.Cancel
        ldx #0
        jsr pConfirm
        .byte 'Save Changes',0
        bne Abort
        jsr SaveSettings
Abort
        rts
```

In the above example, selection of the ‘Yes’ button will return 0 in the accumulator, ‘No’ will return 1 and ‘Cancel’ will return 2. After the *Abort* label, one may test for N=1 and branch accordingly, or check whether the value in the accumulator is 1 (No) or 2 (Cancel).

This is how the buttons look on the screen:



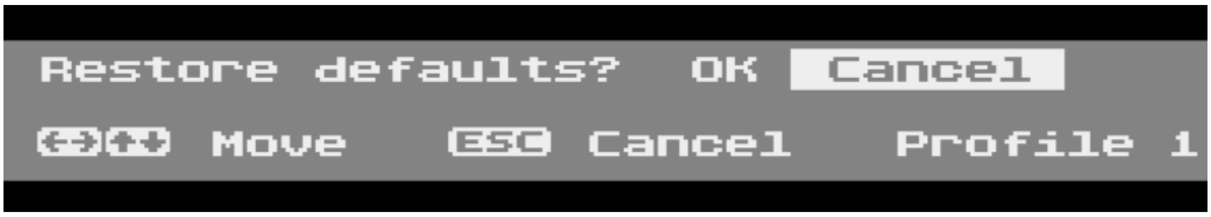
Save changes? Yes No Cancel

Move ESC Cancel Profile 1

A second example:

```
        lda #cmdButton.OK + cmdButton.Cancel
        ldx #1
        jsr pConfirm
        .byte ‘Restore defaults’,0
```

In this example, ‘OK’ and ‘Cancel’ buttons are displayed, and the default action (and the default highlighted button) is ‘Cancel’ (since X = 1). Upon return, the accumulator will contain 0 if the ‘OK’ button was selected, 1 if the ‘Cancel’ button was selected, and the N flag will equal 1 if the user pressed Escape. The buttons will look like this:



Restore defaults? OK Cancel

Move ESC Cancel Profile 1

Testing Conditions and Probing Hardware

So how do we know – for example – whether stereo audio is under the control of the U1MB? 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      ; probe hardware for second POKEY; returns Z=1 if
present
       beq @+
NotActive                  ; if it's still enabled, it's present but not controlled
by M0
       rts
@
       mva #0x01 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.

We can use this information to decide how to display menu items. For instance, if there's no stereo POKEY regardless of the state of M0, we should deactivate any settings pertaining to stereo audio and set their state to 'Disabled'. If stereo is present regardless of the state of M0, we should deactivate any stereo settings, but might set their state to 'Enabled'.

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 0xD1xx and 0xD600-D7FF address space. IO RAM is controlled via bit 6 of 0xD380, with 1 enabling RAM and 0 disabling it (and exposing any hardware registers in the same address space).

Note that the IORAM region at 0xD500-D5BF *cannot be disabled while the configuration is unlocked,* i.e. while the BIOS setup menu is open. Therefore, special provision has been made for writes to the underlying CCTL area (see the example later in the text).

As an example, let's copy an internal flag to a register at 0xD600:

```
        lda Plugin1
        ldy #0
        sty UltimateConfig
        sta 0xD600
        ldy #0x40
        sty UltimateConfig
```

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

The 1088 XEL BIOS plugin uses the hardware write hook to initialise the Sophia VGate function, as well as to force ATR button sensing off.

```
//
//      Setup hardware on reset
//

        .proc HardwareSetup
        lda ConfigBuf[0].Aux2       ; make sure the ATR button is disabled
        and #0xFF-08
        sta ConfigBuf[0].Aux2
        mva #SophiaUnlock SophiaControl
        ldx Device3
        lda VGateTable,x
        sta SophiaControl
        rts
VGateTable
        .byte VGateOff,VGateOn
        .endp
```

CCTL Write Example

Finally, to illustrate the chosen method of overcoming the fact that IORAM at 0xD500-D5BF cannot be disabled while the U1MB configuration is unlocked, we will look at some example code from the plugin function originally suggested and coded by Marcin Sochacki, which sets AtariMax flash cartridges to bank 0 on every system reset.

Here's the hardware setup routine from Marcin's plugin:

```
//
//     Setup hardware on reset
//

       .proc HardwareSetup
       lda CartResetFlag   ; check state of cart reset flag
       beq Off
       ldx #.len[PatchCode]-1     ; it's set, so copy patch code to stack (must not
exceed 16 bytes)
@
       lda PatchCode,x
       sta PluginPatch,x
       dex
       bpl @-
       rts
Off
       lda #$60     ; if cart reset is off, we must place an RTS at the patch
address
       sta PluginPatch     ; since the plugin can't reference the current flag
setting in IORAM
       rts
       .endp


       .proc PatchCode
       lda #0
       sta $D500
       rts
       .endp
```

The hardware setup code is executed just before the operating system is initialised, following a power-on, forced reboot or warm system reset. *CartResetFlag* is equal to 0 or 1 and is initialised by a menu setting which is part of the plugin. Here, we check the state of the flag and branch accordingly, copying patch code to *PluginPatch* or simply writing an RTS instruction if the cartridge bank reset feature is disabled. The Patch code will be executed immediately after the U1MB configuration is locked, but before the operating system is initialised.

It's vitally important that the patch is no more than sixteen bytes in length, and that an RTS instruction is written to *PluginPatch* if the patch functionality is to be disabled.

Identifying Your Plugin

If you modify one of the existing plugins or decide to write your own, please be sure to populate the metadata area appropriately in order that your module can be easily differentiated from other plugins. You should choose a short, descriptive name (which appears on the System Information page), and a two character plugin ID which mnemonically describes what the plugin does. Of course it can be difficult to uniquely express a plugin's functionality using two ASCII characters, but in the case of the ID, uniqueness is more important than readily identifiable meaning (for that, we have the sixteen character descriptive name). The BIOS uses the two character ID to establish whether the active plugin has changed at runtime, and to ensure plugin NVRAM is cleared prior to first use.

It's also useful to maintain major and minor version numbers, as well as revision dates. Be aware also that UFLASH – as a safety precaution – will not flash a plugin to the firmware plugin slot if the version number in the BIOS revision field of the plugin is higher than that of the active BIOS. In this way, we at least ensure that the BIOS should always support the functionality of the plugin being flashed.

Testing Plugins

Although it's perfectly possible to develop plugins *in situ* on the Atari, since plugins can completely hang the system and render it unbootable if they contain bugs, it's strongly advised that they are developed and tested using cross-assembly and emulation. UFLASH is fully functional under the Altirra emulator and one may test plugins in a non-critical environment before saving the firmware once it's been established that things work correctly. While this is not a 100 per cent guarantee of identical functionality on real hardware, it is generally a very strong indicator, owing to the extreme accuracy of Altirra's emulation. Even if proprietary third-party hardware behaviour is unpredictable or unemulated, one may fully establish the reliability of plugin menu items using emulation, and at least be sure that the same plugin flashed to actual hardware will not prevent the machine from booting. And if the machine is able to boot, the plugin can be re-flashed once changes have been made.

I accept no liability for machines rendered inoperable owing to misadventures in plugin development, however, even if caused by errors or omissions in this documentation. Although the flashing of official BIOS updates by means of UFLASH has become almost free of risk, I cannot be held responsible for problems caused by plugin development, and I would strongly advise the purchase of a suitable USB flash ROM programmer by anyone considering plugin development. The inexpensive MiniPro TL866 is ideal for this purpose.

Should you discover any bugs or issues with regard to plugin development, I would nevertheless be interested to hear about them, and I welcome suggestions pertaining to improvements in the API. If you want to accomplish something via an Ultimate 1MB BIOS plugin which is not possible via the current framework, I will consider making the changes necessary to allow it.

Acknowledgements

The author would like to thank the following:

- 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
- Sebastian Bartkowicz (Candle O'Sin) for Ultimate 1MB and Incognito hardware and for providing me with the opportunity to write the original PBI BIOS for both devices
- Lotharek for providing further Ultimate 1MB boards
- Marius Diepenhorst (ProWizard) for his unswerving devotion to testing the original betas of the alt-BIOS, sometimes spending whole consecutive evenings testing new builds and then reporting bugs and suggesting new features
- Marcin Sochacki (TheMontezuma) for SIO2BT hardware and documentation, for help with SIO2BT support, and for his interest in and suggestions for plugins
- Michael St. Pierre (mytek) for his support, donations (material and financial), and for placing the alt-BIOS powered U1MB at the heart of his 1088XEL motherboard project
- Jürgen van Radecke (tf_hh) for his technical expertise
- Every member of the 1088XEL beta testing team for their invaluable feedback and bug reports

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 DrVenkman, Kyle22, rdea6, Stephen, Voy, to name but a few.

Jonathan Halliday

May 2018

References

Altirra Hardware Reference Manual by Avery Lee

Altirra Atari 8-bit emulator by Avery Lee

High Speed SIO Patch by Matthias Reichl

SIO2BT by Marcin Sochacki

Atari 1088XEL Mini-ATX Motherboard by Michael St Pierre

The SpartaDOS X Upgrade Project

Lotharek (Ultimate 1MB, SIDE2)

Candle (Ultimate 1MB, Incognito, SIDE2)