Ultimate 1MB/Incognito/1088XEL|U1MB

BIOS Technical and Developer Documentation

By Jonathan Halliday

Fourth Edition

Revised 03/06/2021

Updated to cover firmware version 4.0

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 a CPLD update allows full U1MB compatibility and thus the use of plugins. Note that as of v.4.0, plugins have moved to the self-test ROM area and have doubled in size (from 1KB to 2KB). Compatibility with plugins from prior versions is entirely broken.

The v.4.0 firmware may be safely flashed to the Incognito regardless of whether the CPLD has been updated to support plugins. Where no plugin support is found to be available (i.e. when manipulation of PORTB bit 7 fails to map the self-test ROM at $5000), the firmware will simply ignore the plugin code.

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
0x30-35	Pointers
0x38-3B	Pointers
0x0200	VDSLST
0x0222	VVBLKI
0x0114-01BF	OS entry code and buffers
0xC000-C1FF	BIOS metadata and jump table
0xC200-CFFF	BIOS code
0xD000-D7FF	Plugin code (executes at $5000; requires CPLD update on Incognito)
0xD800-DFFF	BIOS code
0xE000-E3FF	Character Set
0xE400-FAFF	BIOS code
0xFF00-FFBF	Editable ROM slot descriptions
0xFFC0-FFDF	System ROM space descriptions
0xFFE0-FFF9	Unused
0xFFFA-FFFF	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	
0xC00C	2	Editable slot name base address	0xFF00	
0xC00E	2	Address of NVRAM configuration buffer		
0xC010	2	Reserved		
0xC012	3	JMP to menu item attribute setting routine		
0xC015	3	JMP to message print routine		
0xC018	3	JMP to get user confirmation routine		
0xC01B	3	JMP to list edit routine		
0xC01E	3	JMP to GetVar routine		
0xC021	3	JMP to SetVar routine		

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.

Active Configuration Record

A copy of the configuration data read from the active profile in NVRAM is permanently stored at 0xD100. On U1MB systems, this data extends to 0xD10E, and on Incognito systems, the data extends to 0xD10C. See the *NVRAM Configuration Layout* section for a detailed description of the contents of this area of memory.

Non-volatile system variables

Immediately after the copy of the configuration record are a series of internal variables whose values remain valid until changed or until the machine is shut down. Note that on the Incognito, this data begins at 0xD10D (although this information is provided for interest only, since plugins are not supported on the Incognito).

Address	Bytes	Variable	Description	Values
0xD110	1	PowerOnFlag	Cold power-on flag	Bit 7: 1 = cold power-on
0xD111	1	ColdFlag	Internal cold boot flag	Bit 7: 1 = cold boot
0xD112	1	LoaderBit	Loader flag	Bit 7: 1 = loader active
0xD113	1	MachineType	Platform info	Bit 0: 0 = PAL GTIA, 1 = NTSC GTIA Bit 1: 0 = PAL ANTIC, 1 = NTSC ANTIC Bit 2: 0 = XL/XE, 1 = XEGS (U1MB only)
0xD114	1	CPUFreqMaj	Processor frequency	0-99 (BCD)
0xD115	1	CPUFreqMin	Processor frequency (fraction)	0-99 (BCD)
0xD116	1	Reserved	Reserved byte	n/a
0xD117	1	WarmFlag	Partition table re-read flag	Bit 7: 1 = Warmstart
0xD118	1	SIDEFlag	SIDE present flag	Bit 7: 1 = SIDE present
0xD119	3	Magic	Magic bytes	‘FJC’
0xD11C	1	SDXFlag	SDX active flag	Bit 7: 1 = SDX enabled
0xD11D	1	BootDrive	Boot drive number	0x00 – 0x0F
0xD11E	1	BASICDisableFlag	BASIC disable flag	Bit 7: 1 = BASIC off
0xD11F	1	PluginFlag	Plugin support/present flag	Bit 7: 1 = plugin enabled
0xD120	4	PluginRAM	Reset-proof plugin RAM	Plugin-defined

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 null-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 firmware uses addresses 0x20-2F, 0x3F, and 0x40-7F of the U1MB's SD1305's NVRAM. The U1MB/Incognito build of SpartaDOS X uses bytes 0x30-31. Addresses 0x50-55 are used by the XEX loader for configuration data.

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		Description
0x20	16	Configuration profile 1
0x3F	1	Profile number (0-3)
0x40	16	Configuration profile 2
0x50	16	XEX loader configuration
0x60	16	Configuration profile 3
0x70	16	Configuration profile 4

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	IORAM flag
	7	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 writes (0 = enabled, 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 = 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	Reserved
	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 = OS default, 1 = disabled)
	6-7	IO Noise (0 = SIO, 2 = ATR+SIO, 3 = HDD+ATR+SIO, 4 = off)
8 (Aux8)	0-5	Reserved
	6-7	BIOS logo setting (0 = every boot, 1 = off, 2 = power on)
9 (Ext ID)	0-7	8-bit extension module ID
10-14 (Ext1-5)	0-39	40 bits of extension configuration
15 Checksum	0-7	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 ; Misc options
Aux8   .byte ; Misc options

ExtID  .byte
Ext1   .byte ; Extension bits
Ext2   .byte ; Extension bits
Ext3   .byte ; Extension bits
Ext4   .byte ; Extension bits
Ext5   .byte ; Extension bits
CRC    .byte ; 8-bit config checksum
       .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)	0-3	CONFIG.SYS drive (0 = disabled, 1-15 = drive number)
	4	Internal BASIC default (0 = OS default, 1 = disabled)
	5	Colleen mode (0 = off, 1 = on)
	6	EXTSEL mode (0 = PBI, 1 = Colleen; plugin JED only)
	7	Reserved
1 (Colleen Cfg)	0-1	RAM size (0-3)
	2-3	OS (0-3)
	4	SDX (0 = on, 1 = off)
	5	Unused
	6-7	BIOS logo setting (0 = every boot, 1 = off, 2 = power on)
2 (Colleen Aux)	0-1	IO Noise (0 = SIO, 2 = ATR+SIO, 3 = HDD+ATR+SIO, 4 = off)
	2	SIDE (0 = off, 1 = on)
	3	Axlon (0 = off, 1 = on)
	4	BASIC (0 = off, 1 = on)
	5	Unused
	6	“Z:” CIO Handler (0 = disabled, 1 = enabled)
	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 = 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	Reserved
	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 (Ext ID)	0-7	8-bit extension module ID
10-14 (Ext1-5)	0-39	40 bits of extension configuration
15 Checksum	0-7	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

ExtID  .byte
Ext1   .byte ; Extension bits
Ext2   .byte ; Extension bits
Ext3   .byte ; Exiension bits
Ext4   .byte ; Extension bits
Ext5   .byte ; Extension bits
CRC    .byte ; config checksum
       .ends
```

BIOS Plugins

Version 4.0 of the BIOS supports 2KB plugin modules assembled at 0x$5000-$57FF (i.e. the XL/XE self-test ROM). Plugins may be inserted into the BIOS ROM image at file offset 0x1000 using a suitable editing tool, or flashed directly to the target system using UFLASH. As well as providing support for diverse hardware controlled by the U1MB P2 header signals, plugins may also store information in the NVRAM for retrieval by hardware without a software configuration system of its own. Plugins also allow tailoring of the firmware to the target machine (via the absence of features pertaining to hardware which is not present on the host computer).

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
- Transfer configuration data to and from a specific device when the user saves the BIOS configuration

The default U1MB plugin adds stereo POKEY and VBXE selections to the 'System Clock and Features' menu and defines three stub toggle settings on the 'Device control' menu, corresponding to the M1, S0, and S1 signals on the U1MB. In addition, it adds 'Audio Hardware' and 'VBXE core' entries to the 'System Information' page.

Further levels of hardware control may be achieved by placing configuration data into the U1MB/Incognito NVRAM and extracting said information via the firmware of the target hardware. The PokeyMAX plugin does exactly this, and thus allows four PokeyMAX configuration profiles, even when using the version of PokeyMAX which has no user accessible NVRAM or EPROM memory of its own. Conversely, the Sophia 2 plugin saves user settings on the EPROM of the Sophia itself. Thus, although all four U1MB/Incognito profiles present the exact same single set of Sophia settings, said settings consume none of the U1MB/Incognito NVRAM. All settings are stored in the Sophia's EPROM.

Plugin User RAM

Naturally a plugin requires some RAM for the storage of internal variables. 32 bytes of RAM (0xD7E0-D7FF) are reserved for this purpose, and their contents will remain valid until the user leaves the BIOS setup menu. The supplied header file includes symbolic references for this range of addresses ('Plugin0' through 'Plugin31').

In addition, four 'reset proof' RAM locations at 0xD120-3 are provided. These locations will retain their contents from power-up to power-down.

If indirect zero-page addressing is required, locations 0x38, 0x39, 0x3A and 0x3B may be used. Note, however, that these locations are volatile and will be overwritten outside of the scope of the plugin function in which they are used.

Plugin Structure

BIOS plugins begin with a 80-byte metadata header arranged thus:

Address	Offset	Size (Bytes)	Description	Content	
				U1MB	Incognito
0x5000	0x0000	8	Plugin signature (8 ATASCII bytes)	'ULPLUGIN'	'INPLUGIN'
0x5008	0x0008	1	Host BIOS minimum major revision number (BCD)		
0x5009	0x0009	1	Host BIOS minimum minor revision number (BCD)		
0x500A	0x000A	2	Reserved		
0x500C	0x000A	16	Mnemonic plugin ID (null-terminated, up to 16 bytes including termination)		
0x501C	0x001C	1	Plugin major revision number (BCD)		
0x501D	0x001D	1	Plugin minor revision number (BCD)		
0x501E	0x001E	1	Plugin Revision Day (BCD)		
0x501F	0x001F	1	Plugin Revision Month (BCD)		
0x5020	0x0020	1	Plugin Revision Year (BCD)		
0x5021	0x0021	2	Pointer to menu setup routine		
0x5023	0x0023	2	Pointer to menu update routine		
0x5025	0x0025	2	Pointer to hardware test routine		
0x5027	0x0027	2	Pointer to hardware setup routine		
0x5029	0x0029	2	Pointer to default settings routine		
0x502B	0x002B	2	Pointer to keyboard handler		
0x502D	0x002D	2	Pointer to list control update handler		
0x502F	0x002E	2	Pointer to configuration load function		
0x5031	0x0031	2	Pointer to configuration save function		
0x5033	0x0033	1	Reserved byte		
0x5034	0x0034	32	Menu title (null-terminated ATASCII string)		
0x5050	0x0050	-	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. As of firmware version 3.00, only the major revision number needs to match that of the host BIOS, since public equates are guaranteed not to change in minor revisions. For example: plugins for firmware 2.xx must not be used with firmware 3.xx, and vice versa, but a plugin which stipulates host firmware version 3.00 should work with firmware 3.00 through 3.99.

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 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 null-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 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 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.

Pointer to Default Settings Routine

New to version 3.0, this is the address of the code which is called whenever the BIOS needs to reset the plugin using its default values. The routine will be called:

1. Whenever the current profile's plugin signature does not match the signature of the installed plugin
2. Whenever the user elects to reset all settings in the current profile to default values

Your 'default settings' routine should zero or otherwise set all bits in the current configuration profile which are managed by the plugin. You can write values directly to the configuration bitfields by loading said value into the A register and calling pSetValue.

Pointer to Keyboard Handler

New to version 3.0, this is the address of the code which is called whenever the BIOS receives a keystroke not handled by the internal menu system. The routine will be called with the internal key code in the A register, and the handler should set the carry flag on exit if a full menu redraw is required.

Pointer to List Update Handler

New to version 4.0, this is the address of the code which is called whenever the user opens a list item. The purpose of this callback is to dynamically limit the start and end items in a list depending on external criteria. The address of the currently open list item is passed in X,Y; the handler should check if this word points to the list item of interest, and if it does, set the bounds for the list by loading the lowest item index into the X register and the highest allowable item plus 1 into the Y register, and calling pSetBounds.

As an example, here is the list handler from the PokeyMAX plugin:

```
.proc ListUpdate
.ifdef PokeyMAX
       cpxy #ItemPokey
       bne Done
       lda PokeyMAXFeatures
       and #2 ; do we have quad pokey?
       bne Done      ; if so, do nothing, since all list items are valid
       ldx #0
       ldy #2
       jsr pSetBounds
Done
.endif
       rts
.endp
```

The list control contains three items ('Mono', 'Stereo', and 'Quad') and is – like the entire menu structure – encoded in ROM. In this case, if Quad POKEY is not available, the third list item (that with index 2) is rendered inaccessible.

Pointer to Configuration Load Function

New to version 4.0, this is a pointer to the plugin's configuration load function. A function may thus register as a callback which is executed every time a configuration is loaded by the firmware. The Sophia 2 plugin uses this facility to read configuration data directly from the Sophia's EPROM.

Pointer to Configuration Save Function

New to version 4.0, this is a pointer to the plugin's configuration save function. A function may thus register as a callback which is executed every time a configuration is written by the firmware. The Sophia 2 plugin uses this facility to write configuration data directly to the Sophia's EPROM.

Menu Title

This is a null-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 0xD7E0-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.

An additional four bytes of 'reset proof' plugin RAM are available at $D120-D123. Values written here will survive reset and remain intact until power-off.

Plugin Menu Items

The menu structure of the BIOS setup menu 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 NUL. However, since a plugin with conditional assembly sections may not always know the name of a forward-referenced node, one may also terminate the menu list with a node whose menu number value has bit 7 set.

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 4, since menus are numbered 0 through 7), although it is possible to add extra items to any 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 twenty items (menus with more than ten items will scroll). Menu items which overrun this quota will be ignored.

As a side-note, on the Incognito, the list-based menu structure makes 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
       CfgOff        .byte ; byte offset into config buffer
       CfgMask       .byte ; config buffer bitmask
.ends
```

A second, more concise, type is provided for 'actionable' items (i.e. items which result in immediate action and have no associated setting).

```
.struct ActionItem
       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 string
       Type   .byte ; item type
       Value  .word ; address of called function less 1
.ends
```

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).

CfgOff

Configuration offset, i.e. byte offset into configuration buffer where the setting's value is encoded. See section on U1MB Configuration data for a list of offsets. There are five bytes (40 bits) of NVRAM reserved for plugin configuration use.

CgfMask

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

Value

For the 'ActionItem' type only, this is a pointer to the address of the called function, -1.

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:

```
        .proc Item40
@       dta ActionItem[0] (6,Item41,Title,ItemType.Action,LaunchLoader-1)
Title
        .byte 'SIDE Loader',0
        .endp
```

The code above defines a menu item called 'SIDE Loader' which runs code pointed to by *LaunchLoader-1* when the user presses Enter. The menu item will appear in menu 7, and the next item in the list is *Item41*.

Note that structure 'ActionItem' is also employed here, since it omits superfluous arguments not required by the 'Action' type. You should also use the 'ActionItem' structure for strings.

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:

```
        .proc StereoToggle
@       dta Item[0] (1,Next,Title,ItemType.OnOff,ConfigBuf[0].Aux,$01)
Title
        .byte 'Stereo POKEY',0
Next

        .endp
```

In this example, item *StereoToggle* defines an item in menu 1 with an on/off state. The on/off state is encoded in bit 0 of *ConfigBuf[0].Aux* and is 1 bit wide (therefore holding the value 0 or 1). The item heading is 'Stereo Pokey'. In this case, because of conditional assembly, we don't necessarily know the name of the next menu item in the list, so we merely point to the next node via a reference to 'Next', which is a label at the end of the structure.

OnOffXOR

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

```
        .proc ItemBtn
@       dta Item[0] (2,ItemSIO,Title,ItemType.OnOffXOR,ConfigBuf[0].Aux2,$08)
Title
        .byte 'ATR swap button',0
        .endp
```

In this example, *ItemBtn* defines an on/off item in menu 2 with reverse binary logic. 0 or 1 is encoded in bit 3 of ConfigBuf[0].Aux2. The item title is 'ATR swap button'.

List

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

```
        .proc Item1
@       dta Item[0] (0,Item2,Title,ItemType.List,ConfigBuf[0].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
        .endp
```

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 encoded in bits 0-1 of *ConfigBuf[0].Config*.

If bit 7 of the item type is set (i.e. 0x80 is added to ItemType.List), then the list elements are displayed as plain, unformatted strings. The user-editable OS and BASIC lists are rendered in this way to avoid problems caused by stray '%' symbols in the strings.

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. A list item with more than two items is modal in order to allow a callback to run when the user has finalised their selection with the RETURN key, but the 'ToggleList' type allows for a non-modal list.

```
        .proc Item1
@       dta Item[0] (0,Item2,Title,ItemType.ToggleList,ConfigBuf[0].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
        .endp
```

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:

```
        .proc Item49
```

```
@       dta Item[0] (5,Item50,Title,ItemType.DriveNum,ConfigBuf[0].Ext1,0x0F)
Title
        .byte 'Boot partition',0
        .endp
```

Here, value 0-15 is encoded in bits 0-3 or ConfigBuf[0].Ext1.

PartNum

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

```
        .proc Item49
@       dta Item[0] (5,Item50,Title,ItemType.PartNum,ConfigBuf[0].Ext1,0x1F)
Title
        .byte 'Boot partition',0
        .endp
```

Note that the encoded value requires 5 bits.

Date

For internal use only, this type 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:

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

Note that the encoding/decoding of the RTC is handled internally, so no configuration buffer or mask are required (and if supplied, they will be ignored).

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:

```
        .proc Item37
@       dta Item[0] (4,ItemClick,Title,ItemType.Spinner,ConfigBuf[0].Aux6,0x0F)
        .byte 0x00    ; minimum value
        .byte 0x0F    ; maximum value
Title
        .byte 'Colour',0
        .endp
```

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 encoded in bits 0-3 of *ConfigBuf[0].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:

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

Note the use of the 'ActionItem' structure here, since the additional arguments of the 'Item' structure are redundant here.

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 null 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)

```
        .proc Item43
@       dta ActionItem[0] (7,Item44,Title,ItemType.Action,SaveSettings-1)
Title
        .byte '%s %16c[S]',0
        .word txtSaveChanges,SpaceChar
        .endp

txtSaveChanges
        .byte 'Save changes',0
```

```
SpaceChar
       .byte $20
```

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)

```
       .proc Item27
@      dta Item[0] (5,ItemVideo,Title,ItemType.List,MachineType,$c0)
       .byte 3
       .word List1
       .word List2
       .word List3

Title  .byte 'Processor',0
       .ifdef Incognito
List1  .byte '6502 %x.%xMHz',0
       .else
List1  .byte '6502C %x.%xMHz',0
       .endif
       .word CPUFreqMaj,CPUFreqMin
List2  .byte '65C02 %x.%xMHz',0
       .word CPUFreqMaj,CPUFreqMin
List3  .byte '65C816 %x.%xMHz',0
       .word CPUFreqMaj,CPUFreqMin
       .endp
```

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.

For example:

```
PluginMenu
       .proc StereoToggle
@      dta Item[0] (1,CovoxToggle,Title,ItemType.OnOff,ConfigBuf[0].Aux,0x01)
Title
       .byte 'Stereo Pokey',0
       .endp

       .proc CovoxToggle
@      dta Item[0] (1,Device3Toggle,Title,ItemType.OnOff,ConfigBuf[0].Aux,0x02)
Title
       .byte 'Covox',0
       .endp


       .proc Device3Toggle
@      dta Item[0] (5,Device4Toggle,Title,ItemType.OnOff,ConfigBuf[0].Aux,0x04)
Title
       .byte 'Device 2',0
       .endp


       .proc Device4Toggle
@      dta Item[0] (5,SysInfoStereo,Title,ItemType.OnOff,ConfigBuf[0].Aux,0x08)
Title
       .byte 'Device 3',0
       .endp


       .proc SysInfoStereo
@      dta Item[0] (3,0,Title,ItemType.List,StereoPokeyFlag,$01)
       .byte 2
       .word List1
       .word List2
Title  .byte 'Audio hardware',0
List1  .byte 'Mono',0
List2  .byte 'Stereo',0
       .endp
```

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 0x501E (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 $03,$16               ; host BIOS version (major, minor)

        .word $0000   ; reserved

        .byte 'EP'    ; example plugin

        .byte 0; string termination

        org PluginAddress+28

        .byte $01,$01               ; plugin version (major, minor, BCD) (2 bytes)
        .byte $02,$01,$21           ; revision date

        .word Init    ; pointer to menu setup routine (called on BIOS setup entry)
        .word UpdateMenu            ; pointer to menu update routine (called after every
menu edit)
        .word HardwareTest          ; pointer to the plugin's hardware test routine
        .word HardwareSetup         ; pointer to the plugin's hardware setup routine
(called every reset)
        .word Defaults       ; pointer to default settings initialisation
        .word Keyboard       ; pointer to keyboard input handler
        .word ListUpdate            ; pointer to list callback
        .word LoadCfg
        .word SaveCfg
        .byte 0              ; reserved byte

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

PluginMenu
```

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:

<table>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td colspan="4">Attribute flags</td><td colspan="4" rowspan="2">Number of items, less 1</td></tr>
<tr><td>Internal Selection bit</td><td colspan="3">Item State</td></tr>
</table>

The *Selected* attribute is for internal use and should never be explicitly set. *Item state* is normally zero, but may be explicitly set to the following values:

Bit	6	5	4	Value	Effect
	0	0	0	0x00	Item is displayed with current setting and in an active state
	0	0	1	0x10	Item is dimmed and displayed as “Default”
	0	1	0	0x20	Reserved
	0	1	1	0x30	Reserved
	1	0	0	0x40	Item is dimmed and inactive but current value is displayed
	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
       Default= 0x10
       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:

```
.proc TestPBIOnOff
       ldxy #Item14         ; if PBI is off, grey out everything else
       lda ConfigBuf[0].Aux2
       and #$04
       bne PBIisON
```

```
.ifdef Incognito
       ldxy #XLMenuData.ItemBASIC
.else
       ldxy #ItemBASIC
.endif
       lda #ItemState.Default      ; disable BASIC state if PBI is off
       jsr SetItemFlags
.ifdef Incognito
       ldxy #ItemZHnd
       lda #ItemState.Disabled
       jsr SetItemFlags
.endif
       ldxy #ItemSIO
       lda #ItemState.Disabled     ; disable HSIO if PBI is off
       jsr SetItemFlags
       ldxy #Item14
       lda #ItemState.Disabled + 2 ; disable first three options of PBI BIOS menu if PBI
is off
       bne @+
PBIisON                     ; if PBI is on, activate device ID and main HiSIO
       lda #ItemState.Active + 2   ; 3 items
       jsr SetItemFlags
       ldxy #ItemSIO
       lda #ItemState.Active
@
       jsr SetItemFlags     ; turn on HiSIO

       jsr TestHDDOnOff
       jsr TestHiSIOOnOff   ; set up HiSIO items for drives 1-4 ldxy #ItemZHand
       jmp TestIOSound
.endp
```

When the PBI BIOS is disabled, several items on the same menu 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 a portion of the menu initialisation code from the default plugin:

```
.proc Init
       jsr UpdateMenu

.ifdef Stereo
       ldxy #SysInfoStereo
       jsr Deactivate

       bit StereoPokeyFlag  ; prepare menu before it's displayed
       bmi @+

       lda StereoPokeyFlag  ; if we can't control stereo, make option reflect fixed
status
       and #$01
       sta StereoPokey
       ldxy #StereoSelect
       jsr Deactivate
@

.endif

.if [.def SIDE2] .or [.def XEL]
.ifndef Incognito
       ldxy #ItemZHand
       lda ConfigBuf[0].Aux2
       and #$04
       bne ZHandOn
       lda #ItemState.Disabled
```

```
        .byte $2c
ZHandOn
        lda #ItemState.Active
@
        jsr pSetItemFlags
.endif
.endif
        rts
.endp


//
//      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.

Here is the code which handles the dynamic dimming and activation of SIDE2-related menu items:

```
.proc UpdateMenu

.ifdef SIDE2
.ifndef Incognito
        ldxy #ItemZHand
        lda ConfigBuf[0].Aux2
        and #$04
        bne @+
        lda #ItemState.Disabled
        jsr pSetItemFlags
@


        ldxy #ItemButton
        lda ConfigBuf[0].Aux2
        and #$04
        beq HDDOff
        bit ConfigBuf[0].Aux3
        bmi HDDIsOn
HDDOff
        lda #ItemState.Disabled
        jsr pSetItemFlags
```

```
        lda #ItemState.Default
        bne SetSIDEROM

HDDIsOn
        lda #ItemState.Active
        jsr pSetItemFlags

        lda ConfigBuf[0].Aux2
        and #$08
        beq Done

        lda #ItemState.Disabled
SetSIDEROM
        ldxy #ItemSIDEROM
        jsr pSetItemFlags
Done
.endif
.endif
        rts
.endp
```

Here, we are first checking that the PBI BIOS is enabled (by testing bit 2 of ConfigBuf[0].Aux2), and setting the state of 'ItemZHand' accordingly. We then set the state of 'ItemButton' in a similar fashion, but also testing for the current state of the PBI HDD (bit 7 of ConfigBuf[0].Aux3).

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

```
.proc TestBootDrive  ; If Boot drive is disabled, grey out D1: redirect
        ldxy #ItemD1Swap
        lda ConfigBuf[0].Aux3
        and #$1f
        cmp #$10
        bcs Off
        cmp #2
        bcs On
Off
        lda #ItemState.Disabled
        .byte $2c
On
        lda #ItemState.Active
@
        jmp SetItemFlags
.endp
```

Note that as well as calling these dynamic functions in response to *changes* to menu settings, they should also be called by the menu initialisation routine so that the state of the menu item reflects the proper context.

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 (null-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
```

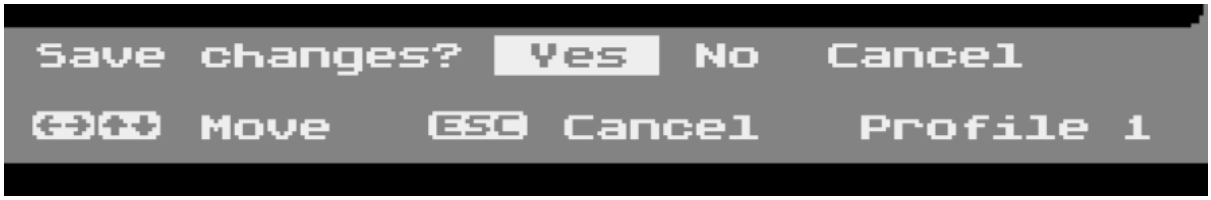
Numbers are assigned to buttons in accordance with their enumerated bit order, so if only 'Yes' and 'No' buttons are displayed, 'Yes' will always return 0 and 'No' will return 1.

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).

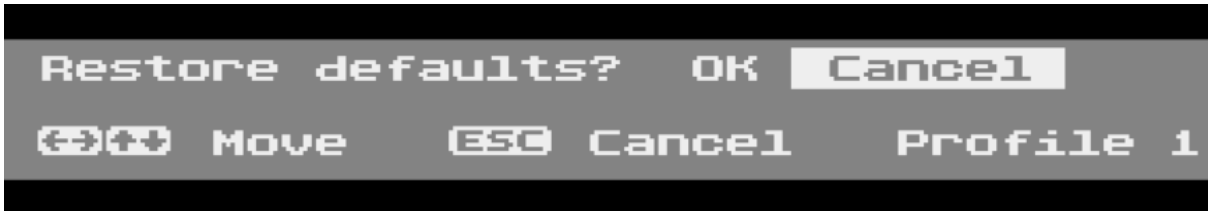
This is how the buttons look on the screen:



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:



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.

Reacting to Keyboard Input

The plugin's keyboard callback will be passed (in the A register) every keystroke issued to the system. It is therefore trivial to react to a user keystroke:

```
.proc Keyboard
.ifdef SIDE3
       cmp #Key.R
       beq RemoveEmuCart
.endif
       rts
.endp
```

Bounds-Limiting a List Control

If you wish to make items in a list control un-selectable on a dynamic basis, the 'UpdateList' callback will accomplish this. The function is called on the opening of any list control, with the address of the control passed in XY. Simply check whether XY points to the control of interest, and pass the index of the lowest permissible list item in X and the highest + 1 in Y to the 'pSetBounds' function.

Example:

```
;      Callback to handle contextual changes to lists

.proc ListUpdate
.ifdef PokeyMAX
       cpxy #ItemPokey
       bne Done

       lda PokeyMAXFeatures
       and #2 ; do we have quad pokey?
       bne Done      ; if so, do nothing, since all list items are valid

       ldx #0
       ldy #2
       jsr pSetBounds

Done
.endif
       rts
.endp
```

Saving and Loading Configuration Data

A plugin may transfer configuration data to and from a device other than the U1MB or Incognito NVRAM by hooking into the firmware configuration load/save configuration functions. The Sophia 2 plugin does this by registering callbacks to the Configuration Load and Configuration Save plugin vectors, thereby storing and retrieving Sophia configuration data to and from the Sophia's EPROM memory, bypassing the U1MB/Incognito NVRAM entirely. Note that this means that only one set of Sophia settings exists, regardless of the currently selected firmware profile. However, if a third-party device has sufficient NVRAM or EPROM storage, there is no reason that a plugin could not maintain multiple configuration profiles on said device's memory.

```
.proc LoadCfg

.ifdef Sophia2
```

```
        bit SophiaDetectFlag
        bmi @+
        sec
        ror SophiaDetectFlag
        jsr DetectSophia
@
        lda SophiaFlag
        beq NoSophia

        jsr WaitSync

        ldy #SPECEN
        sty SGRACTL

        lda SPRIOR
        sta SophiaCfg1
        sta OldSophiaCfg1
        lda SGRACTL
        and #$7E
        sta SophiaCfg2
        sta oldSophiaCfg2
        sta SGRACTL
        jmp @+

NoSophia
        lda #0
        sta SophiaCfg1
        sta SophiaCfg2
@
        jsr RestoreGTIA
.endif

        rts
.endp


.proc SaveCfg

.ifdef Sophia2
        lda SophiaFlag
        beq NoSophia

        jsr WaitSync

        lda #SPECEN
        sta SGRACTL

        lda SophiaCfg1
        sta SPRIOR

        lda SophiaCfg2
        and #$7E
        tax
        lda #SPECEN+NVEN
        sta wsync
        sta SGRACTL
        stx SGRACTL

        jsr RestoreGTIA

NoSophia
.endif

        rts
.endp
```

```
.ifdef Sophia2

.proc RestoreGTIA
       ldy #0
       sty GRACTL
       iny
       sty PRIOR

       rts

.endp

.endif
```

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 can use a string of up to fifteen character (plus null-termination) which mnemonically describes what the plugin does.

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.

PBI BIOS Extensions

As well as main BIOS plugins, the firmware provides one other method of extensibility in the form of PBI BIOS Extensions. Three vectors near the top of page 0xD1 allow new RAM-based code to be 'hooked' into the existing firmware. The full list of public variables and vectors in this area is shown below.

Address	Label	Size (bytes)	Purpose
0xD1AF	FixedBufFlag	1	Bit 7: 1 = Transfer uses internal sector buffer
0xD1B0	ATRCmd	1	Holds ATA command number
0xD1B1	SlaveFlag	1	Bit 4: 1 = Transfer on secondary IDE device
0xD1B2	slenmsb	1	MSB of logical block size (0, 1 or 2)
0xD1B3	rwcmd	1	Internal ATA command number
0xD1B4	CheckSum	2	Checksum (for driver integrity)
0xD1B6	ATRVect	3	24-bit address of ATR IO handler
0xD1B9	RWVect	3	24-bit address of internal sector I/O handler
0xD1BC	SIOVect	2	16-bit address of PBI SIO handler
0xD1BE	Reserved	1	Reserved for future use

The vectors (ATRVect, RWVect and SIOVect) allow the corresponding driver segments to be completely replaced with user-written handlers. SIOVect allows replacing of virtually the entirety of the PBI BIOS with entirely new code (which would first have to be installed in RAM).

Since creating PBI BIOS Extensions is a rather complex proposition, source code for the only existing extension ('Rapidisk') is provided for the reader's perusal. The Rapidisk extension (loaded via DOS or the XEX loader) replaces the sector transfer code (pointed to by RWVect) in ROM with RAM-based code on 65C816-equipped Ataris. The new IO code – since it runs from fast linear RAM – allows significantly improved IO performance on Rapidus machines, and allows for the direct transfer of sectors in and out of linear RAM beyond segment 0.

While it is beyond the scope of this reference to fully document the intricacies of developing an extension in 65C816 assembly language, it is hoped that the provided source code at least provides an illustrative example which interested developers may freely adapt to suit their requirements.

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 for 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
- Eric Bacher (ebigiuy) for his wonderful OSS ROM patches

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, and Voy, to name but a few.

Jonathan Halliday

June 2021

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)