CONNOTECH Experts-conseils inc.
PPCMB/850 Product Family Documentation

ABCD Proto-Kernel™ Software Link and Load Process

(Embedded Software Document)

Document Number C002468

2004/03/09

Document Revision History

C-Number	Date	Explanation
C001446	2003/01/19	Very incomplete version
C002007	2003/10/06	Initial version with reasonable subject area coverage
C002241	2003/12/06	Updated the section 3.1.6, moved text into new section 3.1.8, added section 3.2.5
C002468	2004/03/09	Changes reflecting the release of the lab-comm utility and ABCD Proto-Kernel version 1.2
C002468		Current version

Table of Contents

1. Introduction

This document covers the use of development tools for the creation and loading of software for the PPCMB/850 microprocessor module with the ABCD Proto-Kernel™ software library. This document covers a number of narrow subject areas in the chain of software development activities that convert source code into an operational system with the proper software loaded in.

2. References

[PPCMB850_INIT]
CONNOTECH Experts-conseils inc., *PPCMB/850 Initialization Sequence, (Embedded Software Document)*, Document Number C001804, 2003/09/29

[PPC_ABI]
Steve Zucker, SunSoft, and Kari Karhi, IBM, *System V Application Binary Interface, PowerPC Processor Supplement*, September 1995

[PPC_EABI]
Stephen Sobek, Motorola, and Kevin Burke, IBM, *PowerPC Embedded Application Binary Interface*, 1995/01/10

[LAB_COMM_UG]
CONNOTECH Experts-conseils inc., *The ABCD Proto-Kernel Networking Specifications, including the LAB-COM Utility Guide*, Document Number C002424, 2004/03/09

3. Link and Load Process Overview

3.1 Software Creation Steps

The software creation steps described in the following sub-sections are a global view of the software development procedures. Their description is should assist the reader in quickly becoming productive as a software developer, despite the fact that the ABCD Proto-Kernel™ distribution has no canned installation procedure.

3.1.1 Source Files Collection

This step occurs before the software source files are ready for compilation. Source file collection refers to the retrieval of source files from a centralized repository (e.g. a CVS repository).

3.1.2 Source Files Adaptations

This step occurs before the software source files are ready for compilation. Source file adaptations refers to source files that are created by special procedures or utilities, e.g. when algorithmic constants are pre-computed.

3.1.2.1 Integrated Peripheral Registers Declaration Utility

In the ABCD Proto-Kernel™ implementation for the Motorola MPC8xx processor family, there is a rich set of integrated peripheral registers. The software access to these registers, in assembler and C/C++, require definitions that are generated using a special procedure that ensure an exact match of these parallel definitions, from a common definition file. This is done by the development project driven by the file `projects/misc/mpc850spr_immr/makefile` in the CVS source repository.

3.1.2.2 Processor Clock Distribution Configuration

Another specific source code adaptation occurs for the processor clock distribution configuration. In highly integrated microprocessors, a few external clock sources (often a single one) drives a larger set of timing and clocking functions (e.g. baud rate generators) in hardware timers and other integrated peripherals. In the ABCD Proto-Kernel™ implementation for the Motorola MPC8xx processor family, this clock distribution configuration is turned into a C/C++ #include file using a source code adaptation utility.

3.1.3 Source Code Filtering

The source code filtering step is usually transparent and occurs as the standard C preprocessor performs conditional compilation directives (#if / #ifdef / #ifndef / #elif / #else / #endif). In the ABCD Proto-Kernel™ implementation, the assembler language source code also uses a C preprocessor step, and shares some #include files with the C source code for using common definitions of preprocessor symbols. This can be viewed as good programming practice because it avoids many potential discrepancies between assembler source code and high level language source code.

Explicit source code filtering is advantageous in cases where the exact representation of the compiled source code is useful:

- for software development or diagnostic purposes, where the filtered version of the source code is leaner and simpler than the original source code,
- for software certification purposes, where it can be advantageous to submit the simpler filtered source code to the certification process,
- for software distribution, one may choose to provide the source code of a program as the

filtered source.

Explicit source code filtering is achieved with a utility called `prcpp` that is currently *not* distributed.

3.1.4 Compilation and Link, Creating a .Elf File

This is the traditional software building process using compiler, linker and standard run-time libraries. The ABCD Proto-Kernel™ project started with the intent of using the GCC tools and at least one compiler from a commercial source. As of now, only the GCC tools are supported.

The output of the compilation and link step comprises a .ELF file as the main output, and the linker map file, as diagnostics support information.

3.1.5 Post-link Processing

The post-link processing is centered on the `elf_post_ld` utility that adapts the software image to the embedded target run-time environment. This `elf_post_ld` utility is the main topic of the present document, although most issues handled by it have impacts in other software creation steps.

The `elf_post_ld` utility is distributed under the GNU GPL.

3.1.5.1 Electronic Signature Generation Option

In cases where the target application software should be protected against the sabotage software loading threat, or otherwise require cryptographic-strength integrity protection, the secure software creation process would encompass an electronic signature generation at the post-link processing step. Under the term *electronic signature*, we group either a secret-key based MAC (Message Authentication Code) or a public key digital signature. Like any cryptographic protection scheme, the challenging part is organizational:

- Can the cryptographic key needed by the electronic signature be managed with significantly more care and precaution than the software itself?
- What are the procedures to recover from a security breach in the event that this cryptographic key is compromised?

The key management issues set aside, the electronic signature generation option would affix some undeniable integrity codes to the software image, which would be checked before the software loading operation would be allowed to complete (see section 3.1.7.1 below).

3.1.6 Load Utilities

The load utilities are host programs that transmits the software image to the target embedded system, using an ABCD Proto-Kernel™ download protocol. The load utility is a small Linux utility called the `lab-comm` utility (see document [LAB_COMM_UG]) with a command-line user interface. The download protocol implementation is adequately isolated from this specific load utility, so it can be included as part of other tools.

For the ABCD Proto-Kernel™ embedded loader, the current loading interface is a serial port with the following protocol specifications:

transmission speed: 115200 bps,
asynchronous character format: 8 data bits, no parity, 1 stop bit,
packetized using asynchronous HDLC framing, à la RFC-1549,
RS-422 electrical specification,
full-duplex,
point-to-point connection.

The `lab-comm` utility supports other protocol configurations, and an Ethernet connection as well, so it should be compatible with diverse PPCMB/850-based systems.

For the very initial loading of software on an embedded target hardware, the Motorola MPC8xx BDM connection is used. The current load mechanism uses the same `lab-comm` utility as the embedded loader, and a PPCMB/850 as the BDM interface to load the initial software to another PPCMB/850 embedded target (or another Motorola MPC8xx embedded target). The PPCMB/850 that acts as the BDM interface controller runs an embedded application software that is based on the ABCD Proto-Kernel™.

Note: The mechanism that loaded the first PPCMB/850 unit that ever existed is no longer supported.

3.1.7 Embedded Target Loader

In the embedded target memory, there are two permanent and perhaps one transient software image:

1) the embedded application software image, stored permanently in the flash memory,

2) the embedded target loader software image, also stored permanently in the flash memory,

3) an ad-hoc application software image, stored in the volatile RAM by the embedded target loader immediately before being run.

The embedded target loader is an (almost) dormant embedded software application that is activated only when it appears appropriate to receive a new application software image.

3.1.7.1 Electronic Signature Verification Option

In cases where the target application software should be protected against the sabotage software loading threat, or otherwise require cryptographic-strength integrity protection, the embedded loader would be a *secure* embedded loader, that would not complete the field loading of software unless the electronic signature is verified.

The requirement for software integrity protection for a whole embedded device product family is that no devices would ever be fielded exclusively with the appropriate secure embedded loader, and the BDM loading interface would be blocked by physical anti-tamper protections. This ensures that the signature verification test can hardly be bypassed.

3.1.8 The Very Initial Loader

Loading the embedded target loader itself is a special procedure that uses the BDM interface to the MPC8xx processor. A special software image, the *very initial loader* is created for this purpose. There is no embedded loader in the case of the very initial loader; the intrinsic BDM capability of the Motorola MPC8xx plays that role.

In summary, the *very initial loader* loads the *embedded loader* to the flash, and the *embedded loader* loads either the *embedded application* to the flash, or the *ad-hoc application* to the RAM.

3.1.9 System Startup Sequence

The system startup sequence implements many software run-time environment characteristics (e.g. the zeroization of uninitialized global variables in the C source code). Thus, although the startup sequence is actually part of the developed software, it has a far reaching influence on the software creation process.

Upon a normal system power-up, the system startup sequence starts in the embedded target loader software image and jumps into the application software image (see the reference [PPCMB850_INIT]).

3.2 Relevant Control Files

In this document sub-section, we briefly describe the purpose of relevant files in the software creation procedure.

Many of the files below are centralizing software customization controls. For ABCD Proto-Kernel™ customization (e.g. *board support package* development), many of these files may have to be modified somehow. For embedded application software development, fewer modifications should be required.

Some of the files below contain fixed global definitions, e.g. from external specifications, as an internal interface specification, or as validation rules (e.g. using #if #error #endif preprocessor directives). A developer should not modify these files unless he understands the potential compatibility issues between software created using various revisions of the files.

3.2.1 Makefiles for the Software Creation Procedure

The makefile technique is used throughout the software creation procedure to control the procedure elements to a great level of details. The GNU make utility is used (version 3.79.1).

Even with the best determination to keep it simple, the makefile technique usually turn into questionably readable control file contents. The ABCD Proto-Kernel™ makefiles are not necessarily different.

3.2.2 Application Software Files

The file abcd_config.h is the main configuration header file. It contains the preprocessor symbols used throughout the compilation process. It is the central repository for specifying configuration items for application software variants, many of them having an impact on the ABCD Proto-Kernel™ compilation (the reader should recall that the application software and ABCD Proto-Kernel™ software are not isolated by a formal API).

The files abcd_cfg_defs.h and abcd_cfg_rules.h contain related fixed global definitions. The file abcd_cfg_defs.h contains the definitions that can be referred to by the abcd_config.h definitions (e.g. defining preprocessor symbols for valid choices when a selection is expected in the file abcd_config.h). The file abcd_cfg_rules.h contain preprocessor validation rules (using #if #error #endif preprocessor directives) that can detect configuration inconsistencies (in the file abcd_config.h) as early as possible in the software creation steps.

These three header files are used by both C/C++ source files and assembler source files. Thus, they are limited to preprocessor symbol definitions.

The file abcd_applic_tasklst.h contains the list of application tasks managed by the ABCD Proto-Kernel™ scheduler.

3.2.3 Processor Clock Distribution Configuration

The file mpc8xx_clk_cfg.txt contains input to the clock distribution configuration utility. This file has the same input fields as the web version of the clock distribution utility (see http:www.xyz). The input field syntax in this file is specific, e.g. "{ 123.9}" and consists of an opening curly bracket {, one or more spaces, a numeric value, and a closing curly bracket }. The input fields semantic is position-dependent, but the file mpc8xx_clk_cfg.txt has sufficient comments (text between input fields). Actually, the output of the configuration utility can be used as its input.

3.2.4 Embedded Target Configuration Files

The file mpc8xx_syst_hw_cfg.h contains definitions applicable to the embedded target hardware.

The file mpc8xx_syst_hw_cfg.cpp contains the static table initializations for the MPC8xx parallel I/O configuration.

3.2.5 The Target Definitions for BDM Interfacing Purposes

The principle of operation for the BDM interface to any processor requires the BDM handler software to have some minimal a-priori knowledge and understanding of the target system. Typically with commercial tools, this a-priori knowledge is partly hardcoded in the BDM handler software (e.g. according to the target microprocessor family selection), and sometimes in configuration files and/or debugger script files and/or register definition files.

In the case of the tools provided with the ABCD Proto-Kernel™, the BDM handler software is itself an ABC Proto-Kernel™ application. As free software that can be re-compiled by the user-developer, the whole a-priori target knowledge and understanding is located in the source files. Even if the target hardware is very similar, if not identical, to the hardware on which the BDM handler software runs, it was deemed wise to isolate the *remote target hardware* definitions. The table that follows shows lists the relevant target-related files, and their closest equivalent for the BDM handler software.

	Remote Target Hardware	BDM Handler Software Environment
A-priori knowledge understanding of the MPC8xx remote processor architecture	ppcmb850_bdm/remote_mpc8xx/remote_mpc8xx_instr.asm, ppcmb850_bdm/remote_mpc8xx/remote_mpc8xx.cpp, and ppcmb850_bdm/remote_mpc8xx/remote_mpc8xx.h	multiple source files
MPC8xx SPRs (Special Purpose Registers) definition files	mpc8xx/mpc850spr_immr/spr.txt (note 1)	mpc8xx/mpc850spr_immr/spr.txt (note 1)
.sed files that process the above definition files	ppcmb850_bdm/remote_mpc8xx/rem_spr_macdefs.sed	mpc8xx/mpc850spr_immr/spr_asm.sed, and mpc8xx/mpc850spr_immr/spr_h.sed
MPC8xx IMMR-based registers definition files	mpc8xx/mpc850spr_immr/immr_reg_defs.txt (note 2)	mpc8xx/mpc850spr_immr/immr_reg_defs.txt (note 2)
.sed files that process the above definition files	ppcmb850_bdm/remote_mpc8xx/rem_immr_macdefs.sed	mpc8xx/mpc850spr_immr/immr_asm.sed, mpc8xx/mpc850spr_immr/immr_mac.sed, and mpc8xx/mpc850spr_immr/immr_str.sed
PPCMB/850 configuration definitions	projects/950-0001-01/remote/remote_mpc8xx_defs.cpp, and projects/950-0001-01/remote/remote_mpc8xx_defs.h	projects/752-0002-01/remote/mpc8xx_syst_hw_cfg.cpp, and projects/950-0001-01/mpc8xx_syst_hw_cfg.h (note 3)

Notes:

1) Currently, a unique SPR definition file is used for both environments, but this need not be so if any discrepancy arises. The indicated .sed files implement the environment differentiation in the software image creation procedure.
2) Currently, a unique IMMR-based registers definition file is used for both environments, but this need not be so if any discrepancy arises. The indicated .sed files implement the environment differentiation in the software image creation procedure.
3) The file projects/950-0001-01/remote_mpc8xx_defs.h is a generic declaration file for the specific definitions and initializations in the file projects/752-0002-01/remote/remote_mpc8xx_defs.cpp. Indeed, the part number 950-0001-01 refers to a generic PPCMB/850 unit while the part number 752-0002-01 refers to a PPCMB/850 unit

connected to a BDM interface circuit.

3.2.6 Integrated Peripheral Registers Declaration Utility

The files spr.txt and immr_reg_defs.txt are used respectively to describe the MPC8xx CPU special purpose registers and the MPC8xx IMMR-based registers. These two files use an ad-hoc syntax that is understood by specific command files intended for the linux sed utility.

Variations in the MPC8xx processor family member should be reflected in these files spr.txt and immr_reg_defs.txt.

3.2.7 Linker Script File

The GNU ld linker tool is used in the ABCD Proto-Kernel™ development. The linker script file feature of the GNU ld tool is used extensively to prepare a .ELF file that is reasonably close to the software image file format suitable for the load utilities.

There are three linker script file,

- the file ppcmb850.ld for the embedded application software image that is stored permanently in the flash memory,
- the file ppcmb850_load.ld for the embedded target loader software image that is also stored permanently in the flash memory,
- the file ppcmb850_ram.ld for ad-hoc application software images that are stored in the volatile RAM by the embedded target loader immediately before being run.

3.2.8 Post-link Processing Files

As stated before, the post-link processing is centered on the elf_post_ld utility. This utility reads the .ELF file for the software image and expects special *SHT_NOTE* linker sections grouped in a *PT_NOTE segment* by the GNU ld linker tool (*SHT_NOTE* and *PT_NOTE segment* are defined in the .ELF specification terminology). The data in the PT_NOTE segment are very specific to the ABCD Proto-Kernel™ software creation procedure. This organization is an alternative to an elf_post_ld utility that would read a separate specifications file in addition to the .ELF file.

The file abcd_post_ld_info.asm fills the special *SHT_NOTE* linker sections. It contains overall information like the total size of volatile memory in the target hardware, and the list of supported flash memory organizations, with their detailed specifications.

The assembler syntax required by the file abcd_post_ld_info.asm is not the easiest to work with, but we appreciate having a precise control of what actually gets loaded in the *SHT_NOTE* linker sections. For troubleshooting, it can be useful to read the listing file output created when the assembler processes the file abcd_post_ld_info.asm.One reason for using the file abcd_post_ld_info.asm in the software creation procedure is that it shares all the definitions occurring in the file abcd_config.h with the rest of the software source files.

The post-link processing uses three files for fixed global definitions:

- the file abcd_link_and_load.h, for definitions applicable to the file abcd_post_ld_info.asm,
- the file abcd_load_file_hdr.h, specifying details of the download file format specific to the ABCD Proto-Kernel™ load utilities.
- the file abcd_network_defs.h, for additional definitions applicable to the download file format (which is not perfectly isolated from network definitions, as the name abcd_network_defs.h implies).

These three files are shared with the rest of the software source files. Notably, the last two files are shared with the load utilities.

3.2.9 Load Utilities

As described above, the load utilities use fixed global definitions found in the files abcd_load_file_hdr.h and abcd_network_defs.h.

Unless a developer intends to modify or create a load utility compatible with an ABCD Proto-Kernel™ embedded loader, there are no other files worth mentioning. The core logic for the ABCD Proto-Kernel™ download protocol is in the files abcd_bin_file_download.cpp, and abcd_bin_file_download.h. These files provide a C++ base class definition from which a derived class should be crafted to get a working download utility.

4. Specific Mechanisms

4.1 Linker Script Files and Flash Memory Organization

There are three linker script files that drive the linker step, respectively

(A) the file ppcmb850.ld for the boot-and-run software image, for applications loaded in the flash memory,
(B) the file ppcmb850_load.ld for the embedded loader software image,
(C) the file ppcmb850_ram.ld for the load-and-run software image, for applications loaded immediately before being run.

For the purpose of the link-and-load process, *flash organization data* includes the list of flash sectors (a specification obtained from the flash integrated circuit data sheet) and the usage of each such sector. The possible flash usage indications are:

ABCD_SECTOR_USAGE_APPLICATION,
for the application software image stored in the flash,

ABCD_SECTOR_USAGE_LOADER,
for the embedded loader software image stored in the flash,

ABCD_SECTOR_USAGE_CONFIG_N_LOG_A,
for the flash section ‘A’ dedicated to the FlashCnL API,

ABCD_SECTOR_USAGE_CONFIG_N_LOG_B,
for the flash section ‘B’ dedicated to the FlashCnL API, and

ABCD_SECTOR_USAGE_RESERVED,
for a flash usage not controlled by the present flash usage definitions.

Note: If the need arises for an application-specific flash memory section usage, the usage menmonic ABCD_SECTOR_USAGE_RESERVED may be used, or the above list may be expanded.

When the system resets, the embedded loader software takes control of the CPU. It quickly detects whether an application or itself should run. When an application is given the CPU, it also receives pointers to flash organization data and flash low-level routines for erasing flash sectors and writing data to the flash. With this arrangement, the application source code need neither be tailored to the specific flash of the target system unit nor encompass flash low-level routines for every known flash (however the application link-and-load process must still be aware of possible flash organization data through the file abcd_post_ld_info.asm as explained below).

4.2 PT_NOTE Segments Contents

The ABCD Proto-Kernel™ Software Link and Load Process uses an obscure ELF mechanism for embedding specific control information relevant to the ELF file loading process. This mechanism is the PT_NOTE program segment type.

The contents of PT_NOTE program segments originate from the file abcd_post_ld_info.asm. A foremost benefit of this is that the flash organization alternatives are described in a single place in the set of ASCII files. The final selection of a flash organization alternative occurs through a command argument to the elf_post_ld utility. Finally, the embedded loader receives the software image in buffers tailored to the actual flash memory organization, which simplifies the embedded loader memory management.

Actually, the PT_NOTE program segments are collected by the linker into a single PT_NOTE program header in the ELF file. The `elf_post_ld` utility processes the PT_NOTE entries identified by the entry's originator "ABCD Proto-Kernel (TM)" and ignores other PT_Note entries.

Currently, there are two defined PT_NOTE entry types, described in the following document subsections. Planned PT_NOTE entry types are as follows:

- entry type 2, for a list of peripheral memory regions,
- entry type 4, for algorithms and key reference information for cryptographic integrity protection, and
- entry type 5, for target compatibility information, outside of memory subsystem and irrespective of cryptographic integrity protection.

The contents of the PT_NOTE entries must be compatible with the target system, in the following respects:

- memory configuration,
- flash organization data,
- in the case of application software image, the embedded loader,
- when the entry type 5 is implemented, target configuration stored e.g. in the target system FlashCnL sections, and,
- when the entry type 4 is implemented, integrity algorithms supported by the target system secure embedded loader and cryptographic key information present in the target system.

The basic approach is to tailor the `elf_post_ld` utility output as closely as possible to the target system. This forces the system support organization to keep track of fielded systems configurations. As a consequence, a successful software load leaves little room for undetected incompatibilities.

4.2.1 PT_NOTE Entry Type 1, Basic Information about the ELF File.

The symblic name for the PT_NOTE entry type 1 is ABCD_PT_INFO_TYPE_BASIC_INFO. It must occur exactly once in an ELF file. In this entry type, every data item is a 32 bits unsigned number.

(1) The first data item is the target type. Here is the list of possible values:

ABCD_TARGET_BOOT_AND_RUN,
: for the boot-and-run software image, for applications loaded in the flash memory (compatible with the linker script file ppcmb850.ld),

ABCD_TARGET_LOAD_AND_RUN,
: for the load-and-run software image, for applications loaded immediately before being run (compatible with the linker script file ppcmb850_ram.ld),

ABCD_TARGET_MINI_LOADER, ABCD_TARGET_SECURE_LOADER, or ABCD_TARGET_DEV_LOADER,
: for the embedded loader software image (compatible with the linker script file ppcmb850_load.ld), and

ABCD_TARGET_DEV_VI_LOADER,
: for the very initial loader software image (compatible with the linker script file ppcmb850_ram.ld).

(2) The second data item is the system RAM size, in bytes.

For the target type ABCD_TARGET_DEV_VI_LOADER, there is no other data item.

Otherwise, the following two fields allow the elf_post_ld utility to locate the control information table that drives the software image CRC checking and the ROM-to-RAM copy operation.

Note: This table is the linker section ".flash_segments_desc" with the current linker script files. For an application software image, the following two fields are copied to a special location near the application software entry points.

(3) The third data item, if any, is the address if this control information table.

(4) The fourth data item, if any, is the entry count for this control information table.

(5.a) For the target types ABCD_TARGET_BOOT_AND_RUN, ABCD_TARGET_MINI_LOADER, ABCD_TARGET_SECURE_LOADER, and ABCD_TARGET_DEV_LOADER, there is a fifth and last data item which is the flash memory size.

For the target type ABCD_TARGET_LOAD_AND_RUN, there are two more data items that indicate the embedded loaded footprint in the RAM. This footprint represents the maximum memory used by the embedded loader when loading something to the RAM, respectively at the beginning and the end of the RAM.

(5.b) Thus, the fifth data item for the target type ABCD_TARGET_LOAD_AND_RUN is the maximum memory size occupied by the loader at the *beginning* of the RAM.

Note: This data item value should reflect the symbol ABCD_SRAM_EXCEPT_VECT_FOOTPRINT used when the target system embedded loader was built.

(6.b) The sixth and last data item for the target type ABCD_TARGET_LOAD_AND_RUN is the maximum memory size occupied by the loader at the *end* of the RAM.

Note: This data item value should reflect the symbol ABCD_SRAM_LOADER_FOOTPRINT used when the target system embedded loader was built.

4.2.2 PT_NOTE Entry Type 3, Flash Memory Organization Data

The symblic name for the PT_NOTE entry type 3 is ABCD_PT_INFO_TYPE_FLASH_SECT_LIST. It must occur at least once in an ELF file for the target types ABCD_TARGET_BOOT_AND_RUN, ABCD_TARGET_MINI_LOADER, ABCD_TARGET_SECURE_LOADER, and ABCD_TARGET_DEV_LOADER. For other target types, there must be no PT_NOTE entry type 3.

Each PT_NOTE entry type 3 gives the flash organization data for a possible flash organization in the target system.

The structure of a PT_NOTE entry type 3 is found in the file abcd_link_and_load.h. The name of a flash organization data alternative is a character string set in the file abcd_post_ld_info.asm within each PT_NOTE entry type 3. This name is used with a command argument to the elf_post_ld utility to select the definitive flash organization data in the binary file ready to be loaded by load utilities.

The name of a flash organization data alternative is conveniently set as the flash integrated circuit manufacturer part number. However, if the same part number was used with different flash memory usage allocation (e.g. varying sizes for the FlashCnL sections), the part number alone would not be sufficient identification.

4.3 Dedicated Linker Section Names

The linker script files contain a number of linker section names with specific purposes. Some are justified by the target microprocessor architecture (e.g. the section names for PowerPC exception vectors). Other section names are inherited from the GNU GCC tools (e.g. the .ctors section name for static C++ object constructors).

In addition, the linker script files sometimes handle on an exception basis some specific object files' contribution to specific section names.

The fine details of the linker script files must be understood with other relevant project documentation, including the source code itself.

4.4 Expected ELF Contents

The ABCD Proto-Kernel™ uses assumptions about the ELF file (GNU ld output). Here are some of them:

> RAM at the beginning of the virtual memory address space, flash at the end of the virtual memory address space.
>
> No data is actually loaded in peripheral memory areas, but labels (for variable declarations) can be defined in these areas.
>
> The ABCD Proto-Kernel™ software link and load process makes no attempt to exploit position-independent code and/or position-independent data in the ELF software image. Two exceptions to this rule are 1º the task local data memory section which must be position-independent data and 2º some low-level flash memory interface functions which must run from RAM but are provided by the embedded loader software image to the application software images.

In reference to the ELF specification, the program headers that are relevant to the ABCD Proto-Kernel™ Link and load process are the following ones:

(1.a) PT_LOAD segments having sections with file memory image, and not needing a flash-to-RAM copy operation (either RAM sections for a loader mechanism targeting the RAM directly, or flash sections for code and constant data that is not copied to the RAM upon system startup).

(1.b) PT_LOAD segments having sections with file memory image, and to be copied from the flash memory to the RAM memory at system initialization time.

(1.c) A single PT_LOAD segment holding small data sections with file memory image, having a target address outside of the RAM memory (e.g. for ABCD Proto-Kernel task local data).

> The elf_post_ld utility recognizes up to two special ELF program header segments holding the small data sections (relative to the (E)ABI-defined symbols _SDA_BASE_,

_SDA2_BASE_) whenever a collection of .sdata and .sbss sections (_SDA_BASE_), or a collection of .sdata2 and .sbss2 sections (_SDA2_BASE_), is encountered in a single and complete program header. The utility tolerates that these small data program header segments are linked outside of the flash or RAM memory area.

(2) A single PT_LOAD segment having sections containing virtual memory reservation but no file memory image, used as a placeholder for the control information that drives the system-initialization-time CRC verification and flash-to-RAM copy, this segment address and size being indicated by a PT_NOTE entry within the item (4) below (see section 4.2.1 on page 15 for more details).

(3) Other PT_LOAD segments having sections containing virtual memory reservation but no file memory image, e.g. for peripheral memory definitions. A small data program header segment can fall in this category.

(4) PT_NOTE segments having sections contents conforming to the ELF specification and identified by the entry's originator "ABCD Proto-Kernel (TM)"

One of the duties assigned to the `elf_post_ld` utility is to fulfill the control information table that drives the software image CRC checking and the ROM-to-RAM copy operation. This table is located in item (2) above.

CRC checking occurs for items (1.a), (1.b) and (1.c) above. It is implemented as a 32 bits CRC generation (by the `elf_post_ld` utility) verification (in the system startup sequence).

The traditional ROM-to-RAM copy operation (as is typical in the majority of embedded systems) handles items (1.a) and (1.b).

The item (1.c) is for the ABCD Proto-Kernel™ *task local data*. It is also subject to ROM-to-RAM copy operation, but one copy is made for each execution context (e.g. a task context) in which the software might call some non-reentrant library functions (e.g. the strtok function as defined in the C standard library). The `elf_post_ld` utility handles the item (1.c) as the items (1.a) and (1.b), (except for the fact that it tolerates a target address outside of the RAM or flash for small data sections). The startup sequence processes the item (1.c) differently based on an explicit value for the target address (symbol **`start_of_task_local_data`** with a value defined in the linker script file).

Notes: The task local data is an obscure concept for many application programmers, but its correct implementation in a multi-threading/multi-tasking environment is important to prevent difficult bugs.

The task local data implementation details in the ABCD Proto-Kernel™ are unique.

- The extra small data section (linker section names ".sdata2" and ".sbss2") feature of the EABI specification (references [PPC_ABI] and [PPC_EABI]) is allocated to the ABCD Proto-Kernel™ task local data.

  Note: The link-and-load procedures might slightly depart from the EABI specifications in this respect.

- The source code uses a GNU GCC-specific C language extension for affixing the small section attribute to the task local data variable declarations (this extension is symbolically implemented as TSKLOCAL defined in the file abcd_incl.h). The use of this extension is mandatory for the task local data variables to behave as expected.

- Since the task local data variables are position-independent data, initialization of pointers to these variables may produce unpredictable results.

Here is a description of the elf_post_ld utility processing for the ROM to RAM copy operation. The elf_post_ld utility determines the location of PT_LOAD segments that are copied from the ROM to the RAM taking into account the selected flash organization data. It is thus at this software creation step that an exceedingly large software image may be detected. Moreover, the elf_post_ld utility does not split such PT_LOAD segments, so a single huge PT_LOAD segment might theoretically cause the software creation to fail despite an acceptable total software image size (e.g. if the flash usage information in the flash organization data would fragment the memory areas allocated to the software image).

4.5 Software Image Format and Download Protocol

4.5.1 Software Image File Format

The binary file data is segmented in big segments, each of them being further segmented in small segments. At the file level, the binary data is stored with the segment prefix structures added in front of each small segment. Relevant source code declarations are in the file abcd_load_file_hdr.h.

```
struct abcd_bin_load_file_hdr_str
{
     unsigned short download_reference;
     unsigned short big_segm_rank;
     unsigned short big_segm_number;
     unsigned short small_segm_rank;
     unsigned short small_segm_number;
```

```
    unsigned short small_segm_size;
};
```

At the download transmission time, each segment prefix with its associated data is encapsulated in a protocol data unit and sent according to the download protocol. The byte order in the stored segment prefix structure fields is the one of the target environment.

At the file level, the **`download_reference`** field in the segment prefix structure is present and set to zero. It is intended to be set to an arbitrary constant value for the whole file transmission, so that successive download attempts are not misidentified.

Every small segments of the larger one in which they fit, except perhaps for the last one, are of the same size, indicated by the **`small_segm_size`** field in the segment prefix structure. The **`small_segm_size`** field in the last small segment gives the definitive size for the big segment. The segment prefix structure itself is counted for the size indication. Every small segments contain some data after the segment prefix structure.

Segmentation works as follows. In the file, the **`big_segm_number`** field is a constant value holding the number of big segments in the file. In each big segment, the **`big_segm_rank`** and **`small_segm_number`** fields are constants. The **`small_segm_number`** field value holds the number of small segments in a big segment. Segment ranks (**`big_segm_rank`** and **`small_segm_rank`** fields) appear in increasing order in their respective contexts. The rank counting starts at zero.

4.5.2 Download Protocol Host Processing

The download protocol pertains to the download from a host to a target, of a single software image file. The present specifications is written for a developer of the host side of the protocol.

The host sends small segments from the file and receives receipt acknowledgment frames. The host should re-send small segments that were not acknowledged until every segment is positively acknowledged. Portions of the acknowledgment frame information represents high level feedback information that should be displayed to the download protocol operator for diagnostics assistance.

The download protocol specification leaves the following issues outside of its scope:
- initiation of the download protocol,
- encapsulation of small segments into protocol data units,
- details of the download operator display.

The host shall assign a constant (and preferably unique and unpredictable) value for the

`download_reference` field in the segment prefix structure of every transmission for the downloaded file. In particular, successive runs of the download protocol shall use a different value for the **`download_reference`** field with high probability.

The host starts the download protocol by sending small segment packets for the first big segment. The host shall process the receipt acknowledgment packets as they are received. Once every small segment packets are acknowledged by the target system loader for a big segment, the host may start the start the transmission of the next big segment.

The host may assume that the target system holds sufficient memory buffers for the largest big segment in the file.

4.5.3 Receipt Acknowledgment Packets

The receipt acknowledgment packets are made of a two-byte download reference number (holding a copy of the received the **`download_reference`** field) followed by a sequence of *type-(implicit-or-explicit)length-value* fields. Semantically, these packets may contain up to three layers of status information:

S1, for the small packet receipt acknowledgment,
S2, for the big packet receipt acknowledgment (transmission issues), and
S3, for the big packet processing status.

Each layer can occur at most once in the receipt acknowledgment packet. In each layer actually present in the packet, a type-(implicit-or-explicit)length-value field gives the big segment rank to which the acknowledgment applies. As a practical example, the S1 layer can acknowledge small packets for the big segment packet 1 while the S3 layer reports processing status for segment packet 0.

In the type-(implicit-or-explicit)length-value fields, the type code is a single byte with values among the following symbolic names:

S1 layer (if the first field is present in a packet, exactly one of other two will be present):

> ABCD_LOAD_PROT_ACK_S1_INDEX, for a two-byte value indicating the big segment rank to which the S1 status applies,
>
> ABCD_LOAD_PROT_ACK_ARRAY, for the receipt acknowledgment array part of the S1 status (see below),
>
> ABCD_LOAD_PROT_ACK_BUSY, for a one byte indication (zero=false, non-zero=true) of a temporary lack of receive buffers in the target system,

S2 layer (these fields will appear together in a packet, or be absent):

> ABCD_LOAD_PROT_ACK_S2_INDEX, for a two-byte value indicating the big segment rank to which the S2 status applies,
>
> ABCD_LOAD_PROT_ACK_S2_STAT, for the two-byte S2 status code,

S3 layer (these fields will appear together in a packet, or be absent):

> ABCD_LOAD_PROT_ACK_S3_INDEX, for a two-byte value indicating the big segment rank to which the S3 status applies,
>
> ABCD_LOAD_PROT_ACK_S3_STAT, for the two-byte S3 status code.

The receipt acknowledgment array that follows a type code ABCD_LOAD_PROT_ACK_ARRAY is made of a two-byte bit count, followed by bytes of packed acknowledgment bits. The small packet of rank "n" is acknowledged by the bit of weight "$2^{(n \bmod 8)}$" in the acknowledgment byte "n div 8". An acknowledgment bit value "1" acknowledges the receipt of the corresponding small segment packet. The bit count must equal to the small segment count for the big segment acknowledged by this S1 layer.

The S2 status codes are the following ones:

> ABCD_LOAD_PROT_ACKS2STAT_DONE, for a big segment that has been received completely (a transmission success status), and
>
> ABCD_LOAD_PROT_ACKS2STAT_INCOMPAT, for a big segment incompatibility detected in a small segment packet (this is an indication of a fatal protocol failure).

Some of the S3 status codes are independent of the big segment contents:

> ABCD_LOAD_PROT_ACKS3STAT_IN_PROGR, for a big segment that is being processed,
>
> ABCD_LOAD_PROT_ACKS3STAT_DONE, for a big segment that has been processed successfully (not applicable to the last big segment in the download),
>
> ABCD_LOAD_PROT_ACKS3STAT_LAST_DONE, for the last segment when it has been processed successfully (a download complete indication).

The S3 status codes ABCD_LOAD_PROT_ACKS3STAT_LOWEST_BLOCKING and above are fatal error indications that are specific to the big segment contents, and as such they should be reported by their numeric value to the download protocol operator.

The three status layers should be displayed independently for purposes of user feedback to the download operator. The S1 layer might be dislayed as a temporary progress bar. The S2 layer might be displayed as a permanent progress bar. The S3 layer might be displayed as a single status text line, as the last S3 status received is normally a download success or a fatal error code (the S2 status code `ABCD_LOAD_PROT_ACKS2STAT_INCOMPAT` also falls into the fatal error code category).

4.5.4 Download Segment Contents

So far in the description of the download protocol, the big segment contents has not been considered. The format and semantic of the big segment contents is transparent to the host implementation of the download protocol. Relevant source code declarations are in the file `abcd_load_file_hdr.h`.

Note: With this independence, future revisions of the download protocol may encompass electronic signature of software images without requiring an upgrade of the host loading utilities.

The big segment contents starts with a big segment type code (one byte). The possible values are:

`ABCD_LOAD_SRAM_IMAGE`,

for a memory image portion to be loaded in the RAM,

`ABCD_LOAD_SECTOR_IMAGE`,

for one or more memory image sections to be loaded in a single flash sector that should first be erased,

`ABCD_LOAD_RESET_SECTOR_IMAGE`,

for one or more memory image sections to be loaded in the flash sector that contains the reset instruction for the target CPU.

Note: This big segment type code is currently not used in a software image file to be downloaded. See section 5.6 on page 28.

A software image file must be either all RAM loading or all flash loading. Specifically, if the `ABCD_LOAD_SRAM_IMAGE` type code is not encountered as the first big segment type code by the embedded loader, no other big segment may have this same type code. Conversely, once a ABCD_LOAD_SRAM_IMAGE type code is encountered by the embedded loader, every other big segment must have this same type code.

For the ABCD_LOAD_SRAM_IMAGE type code, the big segment type code is followed by a four-byte address and the binary data to be copied to this address. Loading to RAM is a simple copy operation from the received big segment to the target RAM. The big segment maximum size for downloading to RAM is the symbolic value ABCD_LOAD_SRAM_BUFFER_SIZE.

For the ABCD_LOAD_SECTOR_IMAGE or the ABCD_LOAD_RESET_SECTOR_IMAGE type codes, the big segment type code is followed by a two-byte *portion count*. This refers to the portions that follow immediately. Each such portion contains a 32-bits address, a 32-bits length, and the binary data to be written to this last address. In loading to flash, a big segment contains binary data for a single flash sector.

5. Other Issues Handled by the Link and Load Process

The link-and-load process handles many important details more or less as exceptions to “normal” software creation procedures. In some cases, the elf_post_ld utility represents an opportunity to fix the software image according to the exceptional requirements.

5.1 Reset Instruction Address

The reset instruction is located at a fixed address assuming the reset configuration for the microprocessor memory subsystem. In the current implementation of the link-and-load process, the reset instruction is located at 1 Mega-Byte below the top of flash memory, plus 256 bytes. If it contains an unconditional jump to the first address of the embedded loader software image, the FlashCnL API routines can transparently “host” the reset instruction in a FlashCnL section.

5.2 Enacting the Flash Organization Selection

As explained elsewhere in this document (section 4.2 on page 14), the target flash organization is selected after the linker step in the software image creation procedures. For the application software image, this selection is only relevant to the software loading step (once loaded, the application receives a pointer to the target flash organization upon startup).

In the embedded loader software image, the situation is different. The flash organization selection must be reflected in the code itself (as an initialized global object). The link-and-load design remains committed to the selection done after the link.

Note: The conditional compilation around the global object initialization might achieve the same effect, but perhaps with duplicating the contents of the file abcd_post_ld_info.asm.

So, the elf_post_ld utility has special processing for the embedded loader software image. In this case, the flash organization selection triggers the copy of the flash organization data into a specific location in the software image. This location is the start of the section named ".abcd_flash_organization", where a variable named **`abcd_flash_organization`** must lie.

5.3 Target Flash Organization and the Very Initial Loader

The very initial loader receives an image of the embedded loader. This image is adapted to the target flash by the elf_post_ld utility that created it. The image is indeed C source code that literally tells to erase such flash sectors and write such values at such flash addresses, ... There are very few data validations made by the very initial loader, in contrast with the embedded loader which has access to the flash organization data (against which the incoming software image is validated).

5.4 C/C++ Variable Names for MPC8xx Internal Memory

This document section is specific to the MPC8xx implementation of the ABCD Proto-Kernel™.

As is known by developers familiar with the MPC8xx embedded processor family architecture, an MPC8xx-based target system has some internal memory within the MPC8xx processor. This memory space is partly occupied by various types of memory cells with different characteristics (offsets from the MPC8xx IMMR register contents are indicated in parenthesis):

- internal peripheral registers (IMMR+0x0 to IMMR+0xBFF),
- SIRAM specialized memory (IMMR+0xC00 to IMMR+0xDFF),
- dual-port RAM (IMMR+0x2000 to IMMR+0x3BFF), and
- parameter RAM (IMMR+0x3C00 to IMMR+0x3FFF).

Non-contiguous portions of the dual-port RAM can be used for the MPC8xx RISC CP (Communications Processor) downloadable microcode. Some structures (shared with the MPC8xx RISC CP) in the dual-port RAM or the parameter RAM must obey some alignment restrictions. Overall, the MPC8xx internal memory organization is the door to a rich set of internal peripheral functions and a powerful communications (co-)processor operating with minimal CPU overhead. Moreover, portions the internal memory may be used by the kernel and application programs for its own variables and/or code. However, there are subtle restrictions and interdependencies for the C/C++ access to these memory locations.

In this context, the ABCD Proto-Kernel™ software link and load process attempts to support a simple application programming model: global names of C structure objects for specialized MPC8xx peripheral register groups, channel-specific parameter areas, buffer descriptor tables, and the like. These global names are then available in assembler, and synchronization of C/C++ and assembler source code should be self-verifying. This led to a few exceptional steps in the software creation procedures.

- The structure of the internal peripheral registers is handled by a specific source files adaptation step (see section 3.1.2.1 on page page 5) that ensures proper assembler and C/C++ source file compatibility.

- The small data section feature of the ABI and EABI specifications (references [PPC_ABI] and [PPC_EABI]) is allocated to the MPC8xx internal memory region, the main benefit of this being that the assembler access to the variables in this section are easier to code.

  Note: The link-and-load procedures, up to the `elf_post_ld` utility output, is a departure from the EABI specifications in spirit only, and not literally, due to the following excerpt: "*Even when conforming to the rules above, as long as section size restrictions are met, any variables or unnamed data can be in .sdata, .sdata2 or .PPC.EMB.sdata0, and any variables or unnamed data that are initially 0 can be in .sbss, .sbss2, or .PPC.EMB.sbss0.*" (reference [PPC_EABI], page 8) However, system startup sequence does not initialize the whole .sbss section to zero. Actually it can not change the values of some internal peripherals without preventing further CPU execution, which is a dramatic impact on system operation.

- The source code uses a GNU GCC-specific C language extension for affixing the small section attribute to the internal memory variable declarations (this extension is symbolically implemented as `SDATA` defined in the file `abcd_incl.h`). The use of this extension enables the small data C/C++ optimization opportunity.

- Some special linker script processing is applied to the object files `immr_dpram0.o`, `immr_dpram1.o`, and `immr_param_ram.o` (respectively for source files `immr_dpram0.cpp`, `immr_dpram1.cpp`, and `immr_param_ram.cpp`), so that variables defined in these source files end-up at proper addresses in the target system memory.

- The gaps created by the RISC CP downloadable microcode in the internal memory organization are handled by object file inspection (object files `immr_dpram0.o`, `immr_dpram1.o` are inspected by the free software tool `objdump`, or its cross-tool equivalent) and specific command files intended for the linux `sed` utility. This creates two source files, the files `immr_dpram0_filler.asm` and `immr_dpram1_filler.asm`.

5.5 Application Software Entry Point

The present section is closely related to the startup sequence. See reference [PPCMB850_INIT]. This other document explains what the reset sequence (in the embedded target loader) does before the application software receives the CPU control.

In counterpart, the reset sequence expects the following characteristic from the application software image created by the `elf_post_ld` utility:

The first three 32-bit words at the lowest address allocated to the application software image (in the target CPU memory). With the load-and-run software image, this lowest address is the symbol `ABCD_SRAM_EXCEPT_VECT_FOOTPRINT` used when the target system embedded loader was built. With the boot-and-run software image, this lowest address is the first flash memory address having a usage indication `ABCD_SECTOR_USAGE_APPLICATION`. These three 32-bit words hold, respectively,

- the address of the control information table that drives the software image CRC checking and the ROM-to-RAM copy operation,
- the number of entries in this control information table that drives the software image CRC checking and the ROM-to-RAM copy operation, and
- the first instruction to be executed when the embedded software starts the application software.

5.6 Reset Instruction Recording and Update in the Flash Memory

Currently, the reset instruction in the flash occurs in a flash sector that is part of a FlashCnL section. The very initial loader writes the reset instruction as a jump to the start of the embedded loader software image (plus an 8 bytes offset). Revisions of the embedded loader software image must have the same jump at the reset instruction address. This is because the very initial loader does not know how to preserve the FlashCnL section contents.

When the FlashCnL routines recycle the FlashCnL section, it erases the sector and re-writes the reset instruction immediately.

The embedded loader does not have the capability to load the loader, so it shouldn't encounter a downloaded software image that includes a reset instruction.