CyMCUElfTool 1.0

User Guide

Document Number: 002-22934 Rev. **

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
http://www.cypress.com

Contents

1 Introduction

The CyMCUElfTool version 1.0 is a command line utility used in the build process of PSoC® 6 MCUs. This utility provides facilities for signing important data structures, including generating digital signatures, merging ELF files, and generating bootloadable data for use with the PSoC 6 MCU Bootloader SDK.

Installation

The CyMCUElfTool is bundled with the Peripheral Driver Library (PDL) version 3.0.1 or later, so you must install the PDL to use the tool. See *Cypress Peripheral Driver Library v3.0 Quick Start Guide* for details.

ELF Symbols and Sections

The CyMCUElfTool reads ELF files created by the linkers (GCC, MDK, or IAR) used in the PSoC 6 MCU build process. In addition to the command line option chosen, special ELF symbols and sections will need to be present in the ELF files for the tool to act upon. These symbols and sections are intended to self-document the ELF files to minimize the amount of information needed on the tool's command line.

Output Files Created

By default, the CyMCUElfTool will place its output ELF file in the first ELF file found on its command line. This behavior can be overwritten using the `--output` command line option. In addition, HEX files can be generated from the output ELF file using the `--hex` command line option.

OpenSSL Use

To use the digital signing and encrypted patch features of CyMCUElfTool, the OpenSSL executable must be in your path. Depending on your operating system and environment, this may already be the case. If your system does not have OpenSSL already installed, you can download the source from https://www.openssl.org/. CyMCUElfTool requires OpenSSL v1.0.2.

OpenSSL is only required when digitally signing an application or generating an encrypted patch file (see Digitally Signing Applications and Generating an Encrypted Flash patch (.cyacd2) File). If your application doesn't require these features, OpenSSL is not required.

Merge Rules (symbol order, renaming, and error conditions)

When using the `-M/--merge` command line option to merge ELF files, the following rules are used:

- Only the debug symbols and sections from the first ELF file on the command line are retained in the output file.
- Sections from ELF files beyond the first are converted into binary data and renamed to.merged*n*, where *n* starts at 0 and increments for each section merged to the output file.

- If the tool detects that two or more sections from the input ELF files have overlapping address ranges with different data, an error occurs, and there is no merging.

Hex and Patch File Creation Rules

When creating *.hex* and *patch* (*.cyacd2*) files, the CyMCUElfTool uses a set of special linker symbols to determine the sections from the ELF file that are copied to the target file. These symbols define the start, length, and row size of the memories to be output. These symbols are named:

- __cy_memory_*n*_start
- __cy_memory_*n*_length
- __cy_memory_*n*_row_size

The *n* in the symbol name is an integer equal to or greater than 0. CyMCUElfTool uses these symbols and the following rules when generating *.hex* or *patch* files:

- Duplicate symbols are not allowed in an ELF file (that there can be only one instance of `__cy_memory_0_start`, `length`, or `row_size`).
- Only the memory regions described by these symbols are copied to the target *.hex* or *patch* file.
- The tool starts looking for these symbols with *n* equal to 0 and increments by 1, stopping when a value of *n* is not discovered in the ELF file. This means that if you defined `__cy_memory_0_start`, `length`, or `row_size` and `__cy_memory_2_start`, `length`, or `row_size`, the tool will ignore those memory regions defined in `__cy_memory_2_start`, `length`, or `row_size` and subsequent regions with *n* greater than 2.
- When writing to the *.hex* and *patch* file, the output files are written in a lowest to highest device address order. This means addresses in the range 0x1000xxxx are written before addresses in the range 0x1060xxxx, even if the later was defined by a `__cy_memory_n_xxx` symbol set with *n* lower than the former.

Product Upgrades

Cypress provides scheduled upgrades and version enhancements for CyMCUElfTool, free-of-charge. You can download upgrades directly from www.cypress.com under **Support & Community** > **Software Tools**.

In addition, critical updates to system documentation are provided under **Design Resources**.

Support

Free support for the CyMCUElfTool is available online. You can find the version, build, and service pack information from the command line using the `--version` option.

Visit http://www.cypress.com/support for online technical support. The resources include:

- Training Seminars
- Discussion Forums
- Application Notes
- Developer Community
- Knowledge Base
- Technical Support

You can also view and participate in discussion threads about a wide variety of device topics.

Document Conventions

The following table lists the conventions used throughout this guide:

Convention	Usage
`Courier New`	Displays snippets of source code or command line options in procedures within the text.
Italics	Displays file names, file locations. and reference documentation: *sourcefile.hex*

Revision History

Document Title: CyMCUElfTool 1.0 User Guide **Document Number: 002-22934**		
Revision	**Date**	**Description of Change**
**	2/5/18	New document.

2 Quick Start

This chapter describes how to use CyMCUElfTool for the most common use cases:

- Signing Non-Secure Applications
- Digitally Signing Applications
- Merging ELF Files for a Single Application Arm® Cortex®-M0+ and Cortex-M4 into a Single ELF File
- Merging ELF Files for Multiple Applications into a Single ELF File
- Generating a Flash patch (.cyacd2) File for use with the Bootloader SDK
- Generating an Encrypted Flash patch (.cyacd2) File
- Generating a Code Sharing File

Signing Non-Secure Applications

The CyMCUElfTool `--sign` command modifies an ELF file by calculating signatures or checksums for specific sections. Each of the sections is optional.

Table 2-1. Checksum or Signature Actions

Section Name	Checksum or Signature Actions
.cychecksum	A 2-byte, simple summation checksum of the Flash contents of the ELF file is calculated and populated here.
.cymeta	A custom checksum value used by PSoC Programmer is calculated and populated here.
.cy_toc_part2/.cy_rtoc_part2	A CRC-16-CCITT checksum is calculated using a CRC poly=0x1021 and init value=0xffff on the data in this section and populated in the last 4-bytes of the section(s).
.cy_boot_metadata	A CRC-32C is calculated for this section and populated in the final 4-bytes.

For each section, a message is printed to standard out indicating that the section was discovered or created (if necessary), and an appropriate checksum or signature was calculated.

1. Generate your target ELF file using PSoC Creator™ or your preferred environment (e.g., Makefile, uVision, IAR, etc).
2. Run *cymcuelftool.exe*.

```
cymcuelftool.exe --sign unsigned.elf --output signed.elf --hex signed.hex
```

Digitally Signing Applications

In addition to the sections that can have checksums populated by the `--sign` command, a signature can be calculated to digitally sign an application. If a section named `.cy_app_signature` is present in the ELF file and a signature scheme is provided on the command line, the `.cy_app_signature` section will be filled with the calculated signature. The size of `.cy_app_signature` should be big enough to contain the resulting digital signature:

```
/** Secure Image Digital signature (Populated by cymcuelftool) */
CY_SECTION(".cy_app_signature") __USED CY_ALIGN(4)
static const uint8_t appSignature[SECURE_DIGSIG_SIZE] = {0u};
```

Table 2-2. Required Symbols for Digital Signatures

Section/Symbol Name	Description
.cy_app_signature	The section where the digital signature will be written to
__cy_app_verify_start	A symbol that defines the first address of the memory area whose digital signature is being calculated.
__cy_app_verify_length	A symbol that defines the length of the memory area whose digital signature is being calculated.

1. Build your target ELF file in PSoC Creator or your preferred environment, ensuring that it includes the .cy_app_signature section.
2. Run *cymcuelftool.exe*.

```
cymcuelftool.exe --sign unsigned.elf SHA256 --encrypt RSASSA-PKCS --key
key_2048.pem --output signed.elf –hex
```

The algorithms listed in Table 2-3 and their associated command lines are supported. When more than one byte length is supported for an algorithm, the command line is listed with the options delineated by the '|' character. Algorithms that have 'xxx' in their name can have different key or block lengths. Provide only one of the available lengths when calling the `--sign` command.

Some algorithms require a key passed to the command line. Keys are passed in as hex encoded ASCII files except for the two RSA variants, which require keys in the Privacy Enhanced Memory (PEM) format. Cypress does not generate or manage encryption keys. You should use one of the many available toolsets to create and manage keys.

Table 2-3. Algorithms and Command Lines

Algorithm	Example Command Line
CMAC xxx	`cymcuelftool.exe --sign unsigned.elf CMAC --key key.txt AES-{128|256}-CBC`
HMAC SHAxxx	`cymcuelftool.exe --sign unsigned.elf HMAC SHA{1|224|256|384|512} --key key.txt`
SHAxxx	`cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512}`
CRC	`cymcuelftool.exe --sign unsigned.elf CRC`
SHA xxx encrypted with DES	`cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512} --encrypt DES-ECB --key key.txt`
SHA xxx encrypted with TDES	`cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512} --encrypt TDES-ECB --key key.txt`

Algorithm	Example Command Line
SHA xxx encrypted with AES CBC or CFB	`cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512} --encrypt AES-{128|192|256}-CBC --key key.txt --iv iv.txt` [1] `cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512} --encrypt AES-{128|192|256}-CFB --key key.txt --iv iv.txt` [1]
SHA xxx encrypted with AES ECB	`cymcuelftool.exe --sign unsigned.elf SHA{1|224|256|384|512} --encrypt AES-{128|192|256}-ECB --key key.txt`
SHA256 encrypted with RSASSA-PKCS	`cymcuelftool.exe --sign unsigned.elf SHA256 --encrypt RSASSA-PKCS --key rsa_key_1024/2048.pem` [2]
SHA256 encrypted with RSAES-PKCS	`cymcuelftool.exe --sign unsigned.elf SHA256 --encrypt RSAES-PKCS --key rsa_key_2048.pem` [2]

Merging ELF Files for a Single Application Arm® Cortex®-M0+ and Cortex-M4 into a Single ELF File

Follow these steps to create a single ELF file with both the CM0+ and CM4 in it:

1. Build your PSoC 6 MCU CM0+ ELF file.
2. Sign the CM0+ ELF file.

   ```
   cymcuelftool.exe --sign unsigned_cm0p.elf --output signed_cm0p.elf
   ```

3. Build your PSoC 6 MCU CM4 ELF file.
4. Sign the CM4 ELF file.

   ```
   cymcuelftool.exe --sign unsigned_cm4.elf --output signed_cm4.elf
   ```

5. Merge the signed ELF files.

   ```
   cymcuelftool.exe --merge signed_cm4.elf signed_cm0p.elf --output merged.elf
   ```

Note PSoC Creator and projects exported from PSoC Creator use these steps by default.

Merging ELF Files for Multiple Applications into a Single ELF File

Follow these steps to create a single ELF file with both Application 0 and Application 1 in it:

1. Build your PSoC 6 MCU CM0+ ELF file for Application 0.
2. Sign the CM0+ ELF file.

   ```
   cymcuelftool.exe --sign unsigned_app0_cm0p.elf --output signed_app0_cm0p.elf
   ```

3. Build your PSoC 6 MCU CM4 ELF file for Application 0.
4. Sign the CM4 ELF file.

   ```
   cymcuelftool.exe --sign unsigned_app0_cm4.elf --output signed_app0_cm4.elf
   ```

---

[1] --iv provides a text file containing an encryption initial vector, encoded in hex.

[2] RSASSA-PCKS encrypted value size depends on the size of the key provided in the key file.

5. Merge the signed ELF files.

```
cymcuelftool.exe --merge signed_app0_cm4.elf signed_app0_cm0p.elf --output
merged_app0.elf
```

6. Repeat steps 1 to 5 for Application 1.
7. Merge the ELF files of both applications.

```
cymcuelftool.exe --merge merged_app1.elf merged_app0.elf --output merged_apps.elf -
-hex merged_apps.hex
```

Note These steps to merge can be extended for as many applications as you want by repeating steps 6 and 7 for each additional application. The `--merge` command accepts two or more ELF files in its command line. This means step 7 can be done only once, and so use all single application ELF files, if desired.

Generating a Flash patch (.cyacd2) File for use with the Bootloader SDK

To create a patch file, use *cymcuelftool.exe* on a project you wish to bootload and follow these steps:

1. Define the range of Flash memory to be patched using the __cy_memory_0_xxxx sections in your linker script:

```
__cy_memory_0_start    = 0x10001000;
__cy_memory_0_length   = 0x00100000;
__cy_memory_0_row_size = 0x200;
```

Note Multiple memory areas can be defined and written to the patch file. See Hex and Patch File Creation Rules.

2. Build your application using digitally signed build flow.
3. Generate the patch file:

```
cymcuelftool.exe -P patch.elf --output patch.cyacd2
```

Generating an Encrypted Flash patch (.cyacd2) File

Patch files can be generated with encrypted content intended to be decrypted by the bootloader running in the target device and then written to Flash. The algorithms and their associated command lines listed in Table 2-4 are supported. When more than one byte length is supported for an algorithm, the command line is listed in Table 2-4 with the options delineated by the '|' character.

Table 2-4. Algorithms and Command Lines

Algorithm	Example Command Line
DES	`cymcuelftool.exe -P patch.elf --encrypt DES-ECB --key key.txt --output patch.cyacd2`
TDES	`cymcuelftool.exe -P patch.elf --encrypt TDES-ECB --key key.txt --output patch.cyacd2`

Algorithm	Example Command Line
AES CBC or CFB	`cymcuelftool.exe -P patch.elf --encrypt AES-{128|192|256}-CBC --key key.txt --iv iv.txt` [3] `--output patch.cyacd2` `cymcuelftool.exe -P patch.elf --encrypt AES-{128|192|256}-CFB --key key.txt --iv iv.txt` [3] `--output patch.cyacd2`
AES ECB	`cymcuelftool.exe -P patch.elf --encrypt AES-{128|192|256}-ECB --key key.txt --output patch.cyacd2`
RSASSA-PKCS	`cymcuelftool.exe -P patch.elf --encrypt RSASSA-PKCS --key rsa_key_1024/2048.pem` [4] `--output patch.cyacd2`
RSAES-PKCS	`cymcuelftool.exe -P patch.elf --encrypt RSAES-PKCS --key rsa_key_2048.pem` [4] `--output patch.cyacd2`

Keys are passed as hex-encoded ASCII files except for the two RSA variants, which require keys in the PEM format. Cypress does not generate or manage encryption keys. You should use one of the many available toolsets to create and manage keys.

Generating a Code Sharing File

CyMCUElfTool can be used to generate a file that can be used to share linker symbols from one ELF file to another. This is useful when you want to save memory by defining a variable or function once in one ELF file, but use that variable or function in another.

Note When sharing API symbols between ELF files, never share a function defined in a CM4 ELF file with a CM0+ ELF file as not all CM4 instructions are compatible with the CM0+ instructions and will cause CM0+ to fail.

1. Create a text file named symbols.txt containing one symbol per line:

   ```
   SharedAPI
   SharedVariable
   ```

2. Pass the file created in step 1 to cymcuelftool.exe, specifying the compiler you want to share with (GCC, ARMCC, or IAR):
3. cymcuelftool.exe -R source.elf symbols.txt GCC --output shared_gcc.s
4. Add the shard_gcc.s file to your destination project, assembling it, and linking it to the destination ELF file.
5. Call the shared functions and reference the shared variables as desired in the C/assembly source file(s) linked in your destination ELF file.

[3] --iv provides a text file containing an encryption initial vector, encoded in hex.

[4] RSASSA-PCKS encrypted value size depends on the size of the key provided in the key file.