# C2Prog User Manual

Version 2.4

October 18, 2026

©2006-2026 CodeSkin, LLC
All rights reserved.

## Disclaimer

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

NO COVENANTS, WARRANTIES OR INDEMNITIES OF ANY KIND ARE GRANTED TO THE USER OF THIS SOFTWARE. CODESKIN AND ITS LICENSORS DO NOT WARRANT THAT THE SOFTWARE WILL OPERATE UNINTERRUPTED OR THAT IT WILL BE FREE FROM DEFECTS OR THAT IT WILL MEET YOUR REQUIREMENTS.

# Contents


**1 Introduction** **5**

**2 Quick Start** **8**
2.1 Installation . . . . . 8
2.2 Agentic Setup (MCP) . . . . . 9
2.3 First Programming Session . . . . . 9

**3 Fundamentals** **13**
3.1 Supported Devices . . . . . 13
3.2 Supported Binary Files . . . . . 13
3.3 Extended Hex Files . . . . . 13
3.4 Target Names and Options . . . . . 14
3.4.1 Dual-core devices . . . . . 15
3.4.2 F29x Flash Bank Modes . . . . . 16

**4 Command-Line Interface** **18**
4.1 Exit Status . . . . . 27

**5 Programming Workflows** **29**
5.1 Programming over Serial Link . . . . . 29
5.1.1 Serial over TCP/IP . . . . . 29
5.2 Programming over JTAG . . . . . 30
5.2.1 Port configuration . . . . . 30
5.2.2 XDS110 Considerations . . . . . 31
5.2.3 Changing an Emulator Serial Number . . . . . 32
5.3 Programming over CAN . . . . . 32
5.3.1 CAN Adapters on Windows . . . . . 34
5.3.2 CAN Adapters on Linux . . . . . 34

**6 Integrity and Security** **35**
6.1 F240x and F28x Devices . . . . . 35
6.1.1 CRC Checksum . . . . . 35
6.1.2 Code Security Module (CSM) . . . . . 37
6.1.3 F28x OTP . . . . . 38
6.2 F29x Devices . . . . . 39
6.2.1 Security States, Security Configuration, and Certificates . . . . . 40
6.2.2 Security Configuration in HS-FS . . . . . 42
6.2.3 Key Provisioning . . . . . 44
6.2.4 Signing Methods . . . . . 50
6.2.5 Initial Security Configuration . . . . . 51
6.2.6 Provisioning in HS-SE . . . . . 53
6.2.7 Reviewing a Security Configuration . . . . . 56

6.2.8 Keeping a Device Reachable . . . 59
6.2.9 Authoring a Custom Security Configuration . . . 63
6.2.10 Isolating Parts of an Application . . . 66

**7 Automation and Agentic Interfaces** **68**
7.1 JSON-RPC Interface . . . 68
7.1.1 Load command . . . 68
7.1.2 Get Status command . . . 69
7.1.3 Shutdown command . . . 70
7.1.4 Version command . . . 70
7.1.5 Request errors . . . 70
7.2 Agentic Interface (MCP) . . . 70
7.2.1 Running the server . . . 70
7.2.2 Configuring the agent host . . . 71
7.3 DLL Interface . . . 73

**8 Debugging Tools** **75**
8.1 GNU Debug Server . . . 75
8.1.1 Running the server from the command line . . . 76
8.1.2 Monitor commands . . . 78
8.1.3 Target description and memory map . . . 82
8.1.4 Interrupting and breakpoints . . . 82
8.1.5 Watchpoints . . . 84
8.2 GDB Client . . . 85
8.2.1 Driving the client from a program . . . 86
8.2.2 Script API . . . 88
8.2.3 Diagnostics recipes . . . 90
8.2.4 Error messages . . . 91

**9 Error Codes** **93**

**Appendices** **94**

**A License** **94**
A.1 License Tiers . . . 94

**B 32-bit CRC Algorithm** **95**

**C C2Prog API** **96**

**D PKCS#11 Infrastructure Setup** **99**
D.1 Required Utilities . . . 99
D.1.1 Installing OpenSSL . . . 99
D.1.2 Installing pkcs11-tool . . . 100

D.2 Public Key Fingerprint . . . . . 100
D.2.1 Computing Fingerprints from PEM Files . . . . . 101
D.3 PKCS#11 Providers . . . . . 101
D.4 YubiKey . . . . . 102
D.4.1 Required Software . . . . . 102
D.4.2 Configuring the Yubico PKCS#11 Provider . . . . . 103
D.4.3 Generating Keys on YubiKey . . . . . 103
D.4.4 Importing Existing Keys to YubiKey . . . . . 104
D.4.5 Exporting Public Key and Computing Fingerprint . . . . . 105
D.5 Nitrokey HSM 2 . . . . . 105
D.5.1 Required Software . . . . . 105
D.5.2 Configuring the OpenSC PKCS#11 Provider . . . . . 105
D.5.3 Initializing the Nitrokey HSM 2 . . . . . 106
D.5.4 Generating Keys on Nitrokey HSM 2 . . . . . 106
D.5.5 Exporting Public Key and Computing Fingerprint . . . . . 107

# 1 Introduction

C2Prog is a suite of programming tools for Texas Instruments C2000™ microcontrollers (MCUs). It programs firmware over multiple communication interfaces and is suited to both development and field deployment.

C2Prog supports programming through various communication channels:

- JTAG (for development and debugging)
- RS-232 and RS-485 serial communication
- TCP/IP network protocols
- Controller Area Network (CAN)

The serial, TCP/IP and CAN interfaces serve field deployment, where JTAG access is often unavailable or impractical.

Some salient features of C2Prog are:

- Smart flash erase with automatic sector detection, or manual sector selection
- Automatic 32-bit CRC generation for flash integrity verification during MCU bootup
- "Extended Hex" file format for firmware distribution, carrying all programming settings including the secondary bootloader
- Firmware password protection
- Code signing and certificate generation
- Fast serial communication protocol that works with USB-to-RS-232 converters
- Communication protocol compatible with multidrop networks (RS-485)
- Support for Texas Instruments XDS JTAG emulators
- CAN communications based on ISO-14229/15765 standards
- GNU Debugger (GDB) server and scripting client for MCU diagnostics
- Command-line and DLL interfaces for batch programming and integration of C2Prog functionality into other applications[1]
- JSON-RPC server for CI/CD pipeline integration

[1] Professional or Integration license required for advanced features. See `CodeShop` for licensing details.

- Agentic integration via the Model Context Protocol (MCP), an open standard that lets MCP-compatible AI agents (such as Claude Code) drive C2Prog and query this manual directly
- Modular design enabling customer-specific solutions
- Small application footprint
- Cross-platform compatibility (Windows, macOS, Linux, Embedded Linux) with binaries for Intel and ARM architectures

C2Prog includes the following modules:

- **C2Prog**: Graphical user interface for interactive programming and configuration
- **c2p-cli**: Headless command-line interface for automated and scripted operations
- **c2p-server**: JSON-RPC server for remote operation and CI/CD integration
- **c2p-gdb**: GDB client with enhanced scripting support for debugging workflows
- **c2p-agent**: MCP server exposing C2Prog to AI agents (e.g., Claude Code)

# 2 Quick Start

## 2.1 Installation

The most recent version of the programmer can be downloaded from the C2Prog website:

`https://c2prog.com/download/`

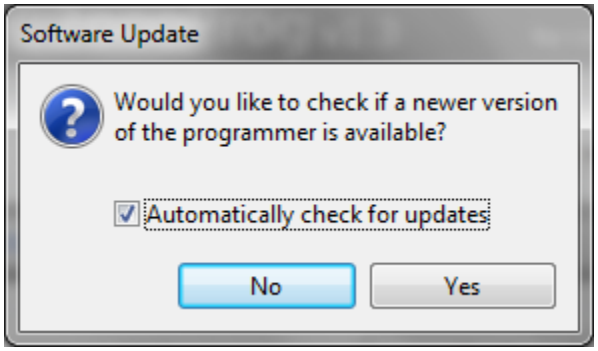
For Windows, an installer executable is provided. C2Prog can also be uninstalled from the control panel like other Windows software.

For macOS, the C2Prog application is distributed as a disk image. Mount the image and copy the application to the desired location.

For Linux, the application files are contained in a self-extracting **makeself** shell script.

During installation, an option is given to add C2Prog to the user path. This is highly recommended, as it will simplify accessing the command line tools (e.g., **c2p-cli**).

When the programmer is launched for the first time, an option is presented to check if a newer version is available. It is highly recommended that you install the most recent version.




If the **Automatically check for updates** option is enabled, the programmer will periodically query the update server to check if a newer version of the application is available.

---

**Privacy:** When communicating with the CodeSkin update server, C2Prog will transmit the C2Prog version number and installation ID. The web server will also have access to the public IP address of the network from which the request is made. CodeSkin may use this data for compiling anonymous statistics. However, no proprietary or personally identifiable data is ever transmitted or collected.

---

## 2.2 Agentic Setup (MCP)

**c2p-agent** is a local MCP server: an AI agent host on the same machine may launch it and call C2Prog as "tools". Register it with your host to ask questions and provide instructions in plain language.

In the graphical interface, open **Help > Set up AI agent integration…**. The dialog lists the supported agent hosts and shows for each whether it is installed and whether the server is registered with it. Select the hosts and click **Register**.

From the command line, **run** `c2p-agent register` **with no option to register every supported host that is installed**, for example Claude Code, Codex, Claude Desktop or Cursor.

```
c2p-agent register
```

On Windows, the installer's **Set up AI agent integration** option runs this registration for you.

A host that is already running continues to use the server it started. Restart the host, or reconnect its MCP servers, after registering.

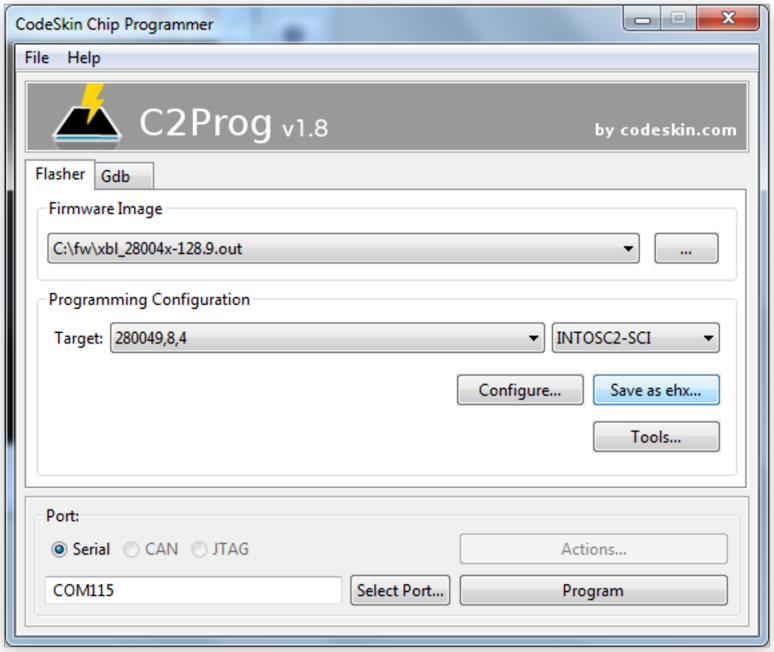
With the server registered, ask the agent a question such as *"Using c2prog, which ports are available to reach my target?"* The agent runs this against your connected hardware, and it can also answer questions about C2Prog from this manual, which it searches locally.

See Section 7.2 on page 70 for the list of supported hosts, registration of individual hosts, and the available tools.

## 2.3 First Programming Session

This walkthrough programs an F28x device from the graphical interface over TI's SCI bootloader. It illustrates the steps common to every C2Prog session; Section 5 on page 29 covers each physical interface in full, and Section 4 on page 18 the command line.

Please refer to the TI Boot ROM Reference Guides for information on how to configure your target for SCI programming. To operate with the C2Prog default settings, the serial link must support full duplex communication at 115200 baud. Slower links are supported but will require custom settings.




Activate the **Flasher** tab and select the **Firmware Image** (binary file, see Section 3.2) using the **…** popup menu.

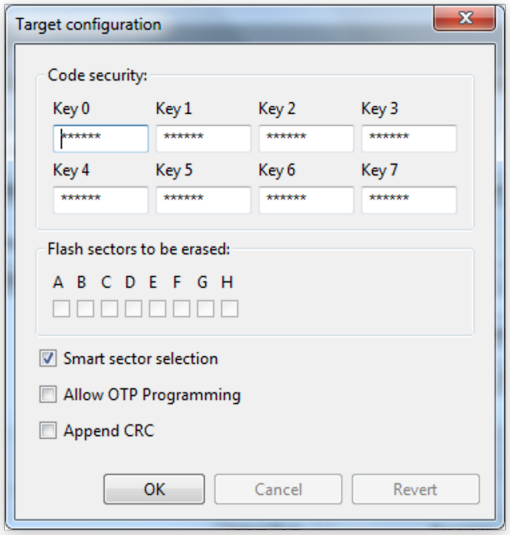
Raw binary files do not define target-specific programming information such as the MCU type to be flashed, oscillator frequency, communication protocol to be used, etc. You must therefore configure this information in the **Programming Configuration** section by choosing the appropriate MCU part name and options (e.g., clock frequency and communication interface; see Section 3.4 on page 14). Contact CodeSkin at info@codeskin.com if no match is found for your hardware.

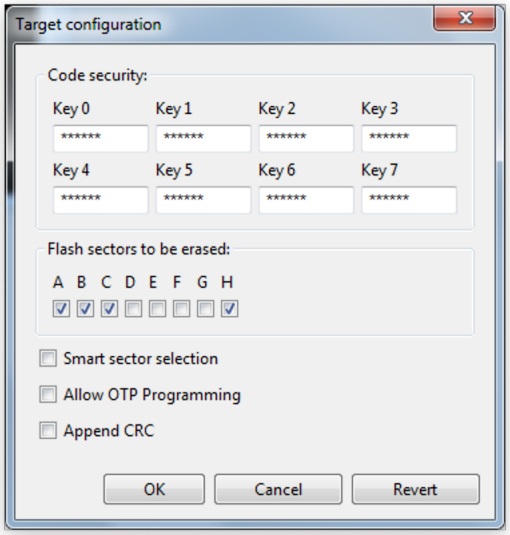
Next, click on **Configure…** to open the configuration dialog.

For programming a locked C2000™ F28x MCU, valid CSM keys (passwords) must be provided as 4-digit hex numbers (with '0x' or '$' prefix). Devices with 32-bit keys present each key as two 16-bit fields, **Key n [31..16]** before **Key n [15..0]**. Unlike other fields, keys are not remembered as defaults: every key starts out as 0xFFFF. The all-0xFFFF and all-0x0000 key patterns select the unlock strategies described in Section 6.1.2 on page 37.

An F29x JTAG target has a **C29 debug password (SSU Mode 3)** field in this dialog. A device whose active security configuration sets its own C29 debug password in SSU Mode 3 needs that password here, as 32 hex digits. For any other device, leave the field empty. See *C29 Debug Password* on page 62.

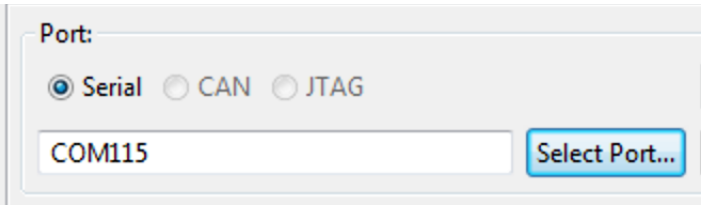
Now you must select which flash sectors should be erased prior to programming. This is done either manually by checking the individual boxes or by choosing **Smart Sector Selection**. The smart sector feature automatically detects which sectors require erasing by parsing the contents of the binary file.




As an additional option, **Append Checksum** can be selected, which instructs the programmer to append a 32-bit CRC checksum to the hex data. This checksum can be used by the MCU to verify the integrity of the flash data, as described later in this document.

Once all programming configurations are completed, configure the COM port by either typing its name directly into the text field or clicking the **Select Port…** button. Valid entries for the serial port on Windows are COM1, COM2, etc.




The **Scan Ports** button in the port selection dialog automatically scans for all available serial ports. On some computers this scan can take considerable time, especially if Bluetooth COM ports are present.

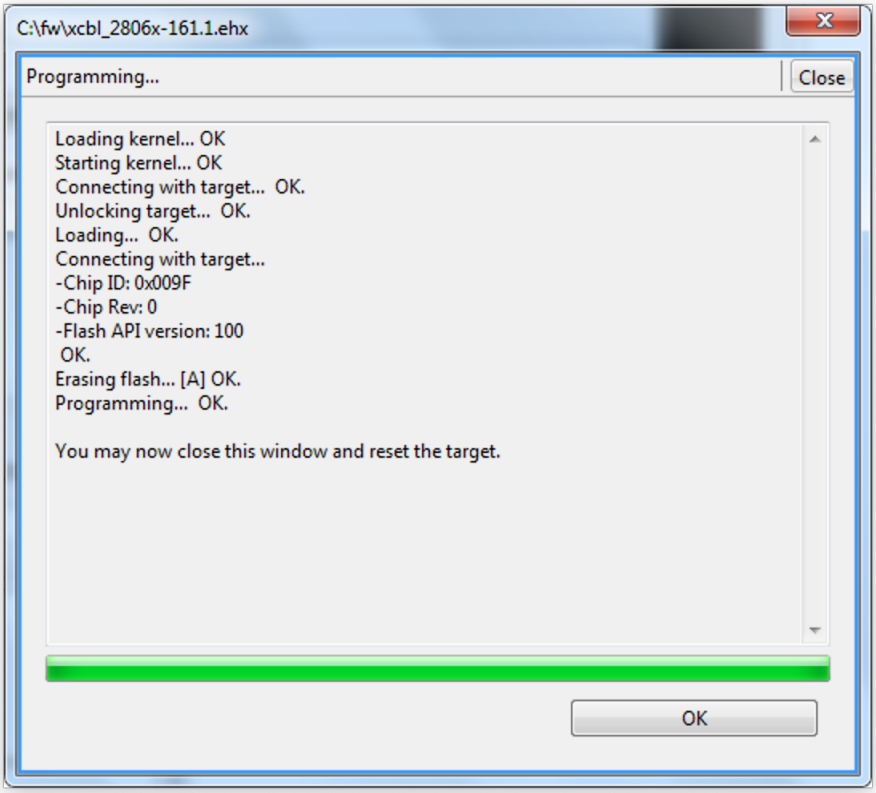
Now the reflashing process can be started by clicking the **Program** button. This will open a new window that displays status information while programming progresses, as shown

below.

---

Do not power-cycle or reset the target during programming.

---




Finally, you may save the programming configuration combined with the firmware image to an Extended Hex file by clicking the **Save as ehx…** button. When this file is subsequently selected in C2Prog, all programming settings are automatically configured. This format is therefore recommended for distributing firmware images.

# 3 Fundamentals

This section covers concepts used throughout the manual: the supported devices, the accepted binary file formats, the Extended Hex container format, and the communication protocols.

## 3.1 Supported Devices

The C2Prog tools target TI C2000™ microcontrollers (MCUs). Primary support focuses on F28x and F29x device families, with limited functionality available for F240x devices.

On F29x devices the distribution of flash between the CPU cores is fixed by a bank mode that C2Prog derives from the selected target and writes while the device is still unprovisioned; Section 3.4.2 on page 16 covers the modes and their targets.

## 3.2 Supported Binary Files

C2Prog supports COFF/ELF (*.out) files and Intel hex (.hex) files.

If you want to use hex files with C2Prog, you must generate them using the following commands:

C2400 Tools:

```
dsphex -romwidth 16 -memwidth 16 -i -o .\Debug\code.hex .\Debug\code.out
```

C2000 Tools:

```
hex2000 -romwidth 16 -memwidth 16 -i -o .\Debug\test.hex .\Debug\test.out
```

C2900 Tools:

```
c29objcopy -O ihex .\Debug\test.out .\Debug\test.hex
```

ARM Tools:

```
armhex -romwidth 8 -memwidth 8 -i -o .\Debug\test.hex .\Debug\test.out
```

## 3.3 Extended Hex Files

The programming configuration, secondary bootloader, HSM runtime (F29 devices), and contents of the firmware binaries can be combined and saved as an “Extended Hex File” (*.ehx). This format is preferable to the raw binary file as it allows programming without

requiring any subsequent manual configuration of the programmer options. Further, in the case of F29x devices, all required certificates and signatures are generated upon creation of the ehx file. An Extended Hex File can also be password protected, making it suitable for distribution while preventing unauthorized use.

---

A secondary bootloader is the flash-programming code that runs on the target. At the start of a programming session it is loaded into target RAM over the communication link (for serial programming, by the primary bootloader in the MCU boot ROM) and then erases and programs the flash with the application code. CodeSkin distributes the compiled secondary bootloaders for the supported MCUs with C2Prog, and they are embedded into the ehx file. Similarly, the HSM runtime is code that executes on the Hardware Security Module (HSM) of F29x devices and takes care of the flash programming of secured devices.

---

By packaging these files into the ehx container, all programming dependencies and settings can be distributed via a single file.

From the graphical user interface of the programmer, an ehx file can be generated by clicking the **Save as ehx…** button. The same can be done by calling C2Prog via its command-line interface (Section 4 on page 18), as below:

```
c2p-cli mkehx --target=28335_30MHz test.out
```

The target parameter corresponds to the concatenation of the target name and option, separated by an underscore. It is recommended that this command be configured as a post-build step in Code Composer Studio:

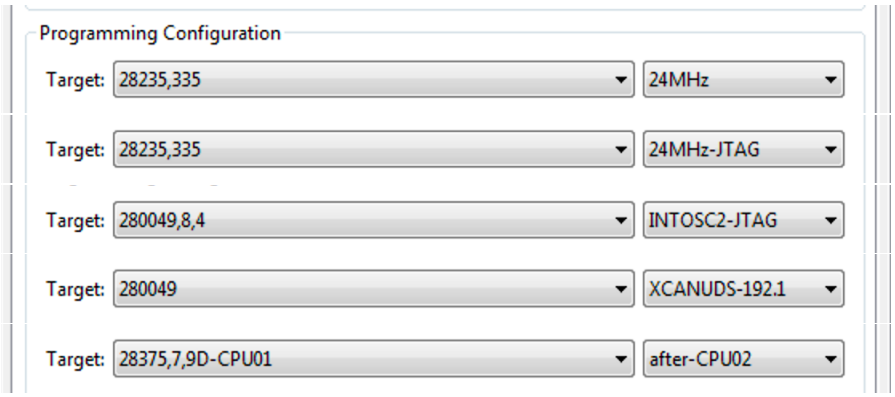
```
"C:\Users\joe\AppData\Local\Programs\C2Prog 2.x\c2p-cli.exe" mkehx --target
   ↪ =28335_30MHz ${BuildArtifactFileBaseName}.out
```

For F29x devices, the commands `mkehx-kp` and `mkehx-cp` generate ehx files for provisioning secured devices.

## 3.4 Target Names and Options

A programming session is defined by two dropdown fields in the **Programming Configuration**: the **Target**, which names the device, and the option next to it, which selects the clock source and the communication interface. On the command line the two are joined by an underscore into the `--target` argument, e.g., **28335_30MHz**.

C2Prog supports multiple communication protocols and interfaces. Originally designed with a focus on serial communication (RS-232/485), C2Prog also supports TCP/IP, Controller Area Network (CAN), and JTAG. Shown below are some examples:




A few comments about target options:

- Frequency values, e.g., **24MHz**, specify the external clock/crystal frequency.
- Options without explicit frequency values use the internal oscillator of the MCU.
- **INTOSC** also refers to the internal oscillator.
- The qualifier after the frequency parameter specifies the communication interface, e.g., **SCI**, **UART**, **CAN**. The default interface is SCI, i.e., **24MHz** stands for SCI communication with an external clock of 24 MHz.
- Options can also refer to customer-specific configurations, such as in the **XCANUDS-192.1** example.

### 3.4.1 Dual-core devices

How a dual-core device is addressed depends on the family.

On the F2837xD family each core has its own target: **28379,378,377,375D-CPU01_SCI** and **28379,378,377,375D-CPU02_SCI**, and likewise for the **_JTAG** variants. A session programs the flash of the selected core, so programming both cores means running C2Prog twice, once per target. The **after-CPU02** option shown in the last example above covers the serial case where CPU1 is programmed in a session that follows one for CPU2.

The F28P65x family needs no such distinction: CPU1 programs the flash of both cores, so a single target such as **28P650DK9,DK8,DK7,DK6,SK7,SK6_JTAG** covers the whole device.

On F29x devices the `--target` name identifies the desired architecture (bank mode), as described next.

### 3.4.2 F29x Flash Bank Modes

F29x devices support configurable flash bank modes that determine how flash memory is mapped and distributed between CPU cores. The bank mode is critical for dual-core applications and is fixed permanently once the device is key provisioned.

The C29 architecture offers multiple bank mode configurations, of which the following are supported by C2Prog:

- **Bank Mode 0**: All flash memory is assigned to CPU1. This mode is used for single-core applications or when CPU1 requires access to the entire flash address space.
- **Bank Mode 2**: Flash memory is split between CPU1 and CPU3. This mode enables true dual-core operation where each CPU has its own dedicated flash regions.

Which of these two modes a device offers depends on its cores. Bank mode 2 distributes the program flash between CPU1 and CPU3. A part without a CPU3 therefore runs in bank mode 0, which is the only mode available for it.

C2Prog configures the bank mode from the selected target. An F29x target name starts with the part numbers the target covers: the **29H85xTU9,TU8** targets program the **F29H850TU9** and the **F29H859TU8**. The core selection that follows the part numbers determines the bank mode. On a part that has a CPU3, the target name must state the selection:

- **29H85xTU9,TU8-CPU1**: Programs flash with code for CPU1 in bank mode 0. Use this target for single-core applications where CPU1 requires access to all available flash memory.
- **29H85xTU9,TU8-CPU1+CPU3**: Selects bank mode 2 and programs every core for which the binary carries data. Use this target for dual-core applications where both CPUs execute from their respective flash regions.
- **29H85xTU9,TU8-CPU3**: Programs CPU3 only in bank mode 2. Use this target when updating only the CPU3 application code while preserving existing CPU1 code.

On a part with a single flash-executing core the target name carries no core selection, because bank mode 0 is the only mode available:

- **29P32xSM2,SM1**: Programs flash with code for CPU1 of the **F29P329SM2** and the **F29P329SM1** in bank mode 0.
- **29P58xDU5**: Programs flash with code for CPU1 of the **F29P589DU5** and the **F29P580DU5** in bank mode 0.

A target name that selects CPU3 on one of these parts is refused, as is a name for an F29H85x part that omits the core selection.

For devices in the HS-FS (High Security-Field Securable) state, C2Prog sets the bank mode of the selected target at the end of the programming session. The device boots in that bank mode on subsequent resets.

Key provisioning also sets the bank mode. A key-provisioning session programs no application code: it commits the bank mode of its target first and then writes the keys (Section 6.2.3).

---

Once a device has been key provisioned, the bank mode can no longer be changed. The bank mode of the key-provisioning target is the one the device keeps (Section 6.2.3).

---

A programming session targets one fixed bank mode. Contact CodeSkin for custom solutions that implement bank-mode swapping.

---

Using the agentic interface (Section 7.2), you can ask an AI assistant to list the available targets and their `--target` options, and to detect the connected communication ports (serial ports and TI JTAG debug probes with their serial numbers).

---

# 4 Command-Line Interface

C2Prog can be launched from a command prompt (shell) with command-line options. This feature supports creating ehx files as part of the build (for example, as a post-build step in Code Composer Studio™). Users with a "professional" or "integration" C2Prog license (Appendix A.1 on page 94) can also program MCUs via the command line and extract hex files from ehx files.

The corresponding executable is **c2p-cli**. It is installed together with the graphical application, in the same directory as the other command-line tools. On Windows, the default installation directory is **%LOCALAPPDATA%\Programs\C2Prog 2.x**. On macOS, the executables are located inside the application bundle, for example in **/Applications/C2Prog.app/Contents/MacOS**. On Linux, they are located in the directory that the self-extracting installer creates.

If the installation directory is not on the user path (Section 2.1 on page 8), **c2p-cli** must be invoked with its full path.

---

Using the agentic interface (Section 7.2), you can ask an AI assistant where **c2p-cli** is installed; it reports the absolute path of each command-line executable of your C2Prog installation.

---

**c2p-cli** accepts the following primary commands and options:

| | |
|---|---|
| `c2p-cli --help` | Display help |
| `c2p-cli --version` | Display version information |
| `c2p-cli get <setting>` | Display a persistent setting |
| `c2p-cli set <setting> <value>` | Change a persistent setting |
| `c2p-cli mkehx [options]` | Create ehx file (F28x and F29x DFU) |
| `c2p-cli mkehx-kp [options]` | Create ehx file for F29x Key Provisioning |
| `c2p-cli mkehx-cp [options]` | Create ehx file for F29x Code Provisioning |
| `c2p-cli mkkeys [options]` | Create the F29x Manufacturer keys |
| `c2p-cli mkkpcert [options]` | Create the F29x key-provisioning certificate |
| `c2p-cli load [options]` | Program ehx file (licensed users only) |
| `c2p-cli extract [options]` | Extract hex file from ehx file (licensed users only) |

The `get` and `set` commands read and change C2Prog's persistent settings, which are shared with the GUI. The settings used in this manual are `ccs-base-path` and `xds110-auto-update` for JTAG programming (Section 5.2 on page 30) and `pkcs11-providers` for PKCS#11 signing (Appendix D on page 99).

The `mkehx` commands, `mkkeys` and `load` additionally accept `--config-file=FILE`, which reads options from a file and combines them with those given on the command line. An option given on the command line wins over the same option in the file. This keeps long invocations, such as a key fingerprint and PIN, out of the build script:

```
target=29H85xTU9,TU8-CPU1_UART
sign-pubkey-fp=85074169edce9ff564f1350d125f304fded0ff73
pkcs11-pin=123456
append-crc=true
```

Each line is one `name=value` entry, with the option name written without its leading dashes. A switch that takes no value on the command line, such as `--append-crc`, is written with an explicit `=true` in the file. An unrecognized name, a name still carrying its dashes, or a line without a value is reported as an error and no ehx file is written. The file supplies options only; the binary and ehx file names remain positional arguments on the command line.

The command for creating an ehx file is

```
c2p-cli mkehx --target=<target> [options] [<binary file>] [<ehx file>]
```

Configured as a post-build step in Code Composer Studio, this produces a ready-to-program ehx file on every build (Section 3.3 on page 13).
The <target> parameter corresponds to the concatenation of the target name and option, separated by an underscore (see page 14).

For example, the following command creates an extended hex file that is protected by a passphrase. All keys to unlock the flash are specified as 0x1234 and the sectors selected to be erased are A, B, C, and D (hex 0xF = binary 1111).

```
c2p-cli mkehx --target=2811_30MHz c:\test.out --keys
    ↪ =1234,1234,1234,1234,1234,1234,1234,1234 --sector-mask=F --append-
    ↪ crc --passphrase="very secret"
```

The command for programming an ehx file is

```
c2p-cli load --port=<port> [options] <ehx file>
```

For example

```
c2p-cli load --port=COM5 test.ehx
```

The command for extracting a hex file is

```
c2p-cli extract [options] <ehx file> [<hex file>]
```

The command for creating an ehx file for F29x Key Provisioning is

```
c2p-cli mkehx-kp --target=<target> [options] <binary file> [<ehx file>]
```

The command for creating an ehx file for F29x Code Provisioning is

```
c2p-cli mkehx-cp --target=<target> [options] [<binary file>] [<ehx file>]
```

The binary file is optional on `mkehx` and `mkehx-cp` when `--seccfg` or `--seccfg-template` names a security configuration: the ehx file then carries the security configuration alone. An optional positional argument ending in **.ehx** specifies the output file.
The command for creating the F29x Manufacturer keys is

```
c2p-cli mkkeys --keyset-password=<password> [options] <keyset file>
```

It writes the four Manufacturer keys to a new keyset, which the signing options below then read. See *Generating the Manufacturer Keys* on page 44.
The command for creating the F29x key-provisioning certificate, the binary `mkehx-kp` packages, is

```
c2p-cli mkkpcert --ti-fek=<TI FEK public key> --keyset=<keyset> [options]
    ↪ <certificate file>
```

See *Generating the Key-Provisioning Certificate* on page 45 for what the options program into the device.

| **mkehx options** | |
|---|---|
| `--target=TARGET_ID` | Selects the target: the target ID is the target name, followed by an underscore "_" and the target option, for example "2812_30MHz". |
| `--keys=KEY1,KEY2,…` | Configures keys for unlocking flash (optional), KEYn is in 16-bit hex notation without '0x' prefix. In the case of 32-bit keys, each key is split into two 16-bit keys in big-endian (BE) order. E.g., keys 0x01234567, 0x89ABCDEF, 0x11112222, 0x33334444 are specified as `--keys=0123,4567,89AB,CDEF,1111,2222,3333,4444`. |
| `--debug-password=PASSWORD` | F29x JTAG only: C29 debug password of the security configuration active on the device (optional). PASSWORD is 32 hex digits, written as in the profile's `debug.c29.password`. Without the option, C2Prog presents the SDK default password. See *C29 Debug Password* on page 62. |
| `--sector-mask=SECTOR_MASK` | Configures which flash sectors are erased, where SECTOR_MASK is a hex number:<br>`--sector-mask=1`: sector A<br>`--sector-mask=2`: sector B<br>`--sector-mask=3`: sectors A & B<br>`--sector-mask=A`: sectors B & D<br>If the `--sector-mask` option is not used, then sectors to be erased are automatically detected. |
| `--append-crc` | Enables addition of CRC checksum (optional) |
| `--allow-otp` | Allows OTP programming (optional) |

| mkehx options (continued) | |
|---|---|
| `--allow-seccfg` | F29x only: allows programming the security configuration carried in the binary (optional). Without it, a binary carrying seccfg blocks is refused. See *Security Configuration in HS-FS* on page 42. |
| `--allow-irreversible` | F29x only: acknowledge the irrecoverable or lockout consequences of the security configuration and write the ehx anyway. Review them with `--explain-only`. |
| `--seccfg=LUA_FILE` | F29x HS-FS only: program the **.lua** security-configuration profile, alone or merged into the binary. Needs no signing key and no `--allow-seccfg`. See *Security Configuration in HS-FS* on page 42. |
| `--seccfg-template=NAME` | As `--seccfg`, but selecting a built-in template by name, such as **default_seccfg_ssumode1** |
| `--ssu-mode=MODE` | Build the profile named by `--seccfg` as if its `ssu_mode` were MODE (1, 2 or 3), leaving the file unchanged. Requires `--seccfg` or `--seccfg-template`. |
| `--explain-only` | F29x only: print the human-readable report of the security configuration carried in the binary and exit without writing an ehx. See *Consequences of a Security Configuration* on page 56. |
| `--explain-only-json` | As `--explain-only`, but emitting the report as a single JSON document for build pipelines and AI assistants. A binary without a security configuration reports `seccfg_present: false` in the same schema. |
| `--passphrase=PASS_PHRASE` | Passphrase for extended hex file |
| `--bitrate=BIT_RATE` | Bit rate for some protocols (such as CAN); the default is 500000 |
| `--ta=TARGET_ADDRESS` | Target address for multidrop protocols (such as CAN); the default is 0x770 |
| `--sa=SOURCE_ADDRESS` | Source address for multidrop protocols (such as CAN); the default is 0x778 |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

In the case of F29x devices, sector-related options refer to *banks* in the address order of the targeted flash bank mode.

| **load options** | |
|---|---|
| `--port` | Specifies communication port |
| `--passphrase=PASS_PHRASE` | Passphrase for extended hex file |
| `--print-progress` | Enables the display of progress information |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

| **extract options** | |
|---|---|
| `--passphrase=PASS_PHRASE` | Passphrase for extended hex file |

| mkehx-kp options | |
|---|---|
| `--target=TARGET_ID` | Selects the target: the target ID is the target name, followed by an underscore "_" and the target option, for example "29H85xTU9,TU8-CPU1_UART". The target also selects the flash bank mode, which must match the mode of the device. On a part with a CPU3, the core the target name selects states the mode (CPU1: bank mode 0; CPU3: bank mode 2); see Section 3.4.2. |
| `--otp-kw=TI_OTP_KW_FILE` | Specifies the TI Keywriter binary (HSM runtime image). |
| `--debug-password=PASSWORD` | JTAG only: C29 debug password of the security configuration active on the device, as for `mkehx` (optional). |
| `--passphrase=PASS_PHRASE` | Passphrase for extended hex file |
| `--dry-run` | Performs key provisioning test without committing keys to OTP. All provisioning steps are executed but keys are not permanently written. Use this option to verify the key provisioning workflow before committing irreversible changes to the device. |
| `--explain-only` | Print the human-readable report of the key-provisioning consequences decoded from the certificate and exit without writing an ehx. See *Reviewing the Certificate* on page 49. |
| `--explain-only-json` | As `--explain-only`, but emitting the report as a single JSON document for build pipelines and AI assistants. |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

| mkehx-cp options | |
|---|---|
| `--target=TARGET_ID` | Selects the target: the target ID is the target name, followed by an underscore "_" and the target option, for example "29H85xTU9,TU8-CPU1_UART". |
| `--seccfg=LUA_FILE` | Compile and sign a **.lua** security-configuration profile into the ehx, alone or alongside the application binary. See *Initial Security Configuration* on page 51 and *Provisioning in HS-SE* on page 53. |
| `--seccfg-template=NAME` | As `--seccfg`, but selecting a built-in template by name, such as **default_seccfg_ssumode1** |

| mkehx-cp options (continued) | |
|---|---|
| `--swrv=REVISION` | Software revision recorded in the application and security-configuration certificates; the device fails a certificate whose revision is below the software-revision floor its key provisioning installed. The default is 1. The flash-kernel and HSM-runtime certificates always carry revision 1. |
| `--debug-password=PASSWORD` | JTAG only: C29 debug password of the security configuration active on the device, as for `mkehx` (optional). |
| `--passphrase=PASS_PHRASE` | Passphrase for extended hex file |
| `--explain-only` | Print the human-readable report of the security configuration, carried in the binary or named by `--seccfg`, and exit without signing or writing an ehx. See *Consequences of a Security Configuration* on page 56. |
| `--explain-only-json` | As `--explain-only`, but emitting the report as a single JSON document for build pipelines and AI assistants. |
| `--allow-irreversible` | Acknowledge the irrecoverable or lockout consequences and write the ehx anyway. Review them with `--explain-only`. |
| `--ssu-mode=MODE` | Build the profile named by `--seccfg` as if its `ssu_mode` were MODE (1, 2 or 3), leaving the file unchanged. The consequence report records the mode given. |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

| mkehx-cp signing and encryption options | |
|---|---|
| `--sign-key-file=KEY_FILE` | Path to PEM-encoded private key file for signing |
| `--sign-pubkey-fp=FINGERPRINT` | SHA1 or SHA256 fingerprint of public key for PKCS#11-based signing. See Appendix D on page 99 for PKCS#11 setup. |
| `--pkcs11-pin=PIN` | PIN for PKCS#11 device access |
| `--keyset=KEYSET_FILE` | Path to a password-encrypted keyset holding the signing keys |
| `--keyset-password=PASSWORD` | Password protecting the keyset |
| `--keyrev=SET` | Manufacturer key set to sign with: `secondary` (the SMPK, the default) or `backup` (the BMPK). The numbers `1` and `2` select the same two sets. |
| `--encrypt` | Encrypt the CPU application images with the keyset's encryption key: SMEK, or BMEK with `--keyrev=backup`. Requires `--keyset`. See *Encrypted Application Images* on page 55. |

| mkkeys options | |
|---|---|
| `--keyset-password=PASSWORD` | Password protecting the new keyset. Required; without it the command refuses rather than writing an unprotected keyset. |
| `--key-algorithm=NAME` | Signing key algorithm: `rsa4k` (default), `secp256r1`, `secp384r1`, `secp521r1` or `brainpool512` |
| `--key-algorithm-backup=NAME` | Signing key algorithm of the Backup key (BMPK), from the same list. Defaults to the algorithm `--key-algorithm` selects. |
| `--keyset-name=NAME` | Name recorded inside the keyset. Defaults to the file name without its extension. |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

| mkkpcert options | |
|---|---|
| `--ti-fek=KEY_FILE` | TI Field Encryption Key public key of the silicon revision being provisioned. Required. |
| `--keyset=KEYSET_FILE` | Keyset holding the Manufacturer keys, with `--keyset-password`. Both signing keys are used: the Secondary signs the primary certificate, the Backup the secondary one. Required. |
| `--msv=VALUE` | Manufacturer specific value, up to six hex digits. When omitted, the field is left unprogrammed. |

| mkkpcert options (continued) | |
|---|---|
| `--keycnt=COUNT` | Manufacturer key sets programmed: 1 or 2 (default 2). |
| `--keyrev=SET` | Manufacturer key set the device treats as current: `secondary` (the SMPK, the default) or `backup` (the BMPK). The numbers `1` and `2` select the same two sets. A device programmed with one key set cannot be set to the other. |
| `--sr-sbl=COUNT`,<br>`--sr-hsmrt=COUNT`,<br>`--sr-app=COUNT`,<br>`--sr-ssu=COUNT` | Anti-rollback floor of the bootloader, the HSM runtime, the application and the security configuration. Each is a revision count, at most 32 (64 for the security configuration); each defaults to 1. |
| `--ext-otp=HEX` | Extended OTP payload, with `--ext-otp-index=BITS` and `--ext-otp-size=BITS` placing it in the 1664-bit region and `--ext-otp-protect=HEX` giving the protection mask, a 64-bit value the field carries twice, once for write protection and once for read protection (all zero by default). The offset and the size are multiples of 8; a size above the payload's own length pads it with zero bytes at its most significant end. |
| `--mpk-opt=VALUE`,<br>`--mek-opt=VALUE` | The Manufacturer key and encryption key option fields, 10 and 5 bits wide, as hexadecimal numbers. Both are reserved on F29H85x; when omitted, they are left unprogrammed. |
| `--keys=LIST` | Manufacturer key fields to program: `keys` (all four, the default) or a comma-separated list of `smpk`, `smek`, `bmpk`, `bmek`. A field left out is written as zeros and marked unprogrammed. Without `bmpk` there is no Backup key to sign with, so the certificate is the Secondary-signed one alone. |
| `--protect=LIST` | Fields whose OTP rows are closed to later writes and reads: `keys` (default), `all`, `none`, `swrev`, or a comma-separated list of `smpk`, `smek`, `bmpk`, `bmek`, `msv`, `keycnt`, `keyrev`, `sr-sbl`, `sr-hsmrt`, `sr-app`, `sr-ssu`, `ext-otp`, `mpk-opt`, `mek-opt`. `sr-app` and `sr-ssu`, and therefore `swrev` and `all`, are not yet supported. |
| `--override=LIST` | Fields the keywriter may program over eFuse rows that already hold a value, from the same set as `--protect`; none by default. Being OTP, such a write can only set further bits, never clear one. |

| mkkpcert options (continued) | |
|---|---|
| `--signing-algorithm=SCHEME` | RSA signature scheme of both certificates and of the signatures over the payload key: `pkcs1v15` (default) or `pss`. |
| `--aes256=KEY_FILE` | File holding the 32-byte payload key the certificate's secrets are encrypted under. One is generated for each certificate when this is not given. |
| `--sign-pubkey-fp=FINGERPRINT`, `--sign-pubkey-fp-backup=FINGERPRINT` | SHA-1 or SHA-256 fingerprints of the Secondary and Backup Manufacturer keys on a PKCS#11 token, which sign in place of the keyset's. Both keys come from the token or both from the keyset. See Appendix D on page 99. |
| `--pkcs11-pin=PIN` | PIN of the token holding the signing keys. |
| `--config-file=FILE` | Read options from a file and combine them with those given on the command line; an option on the command line wins over the same option in the file. |

## 4.1 Exit Status

Every **c2p-cli** command reports its outcome through the process exit status, so a build pipeline or an AI assistant can act on the result without parsing the message:

- 0: the command did what was asked
- 1: failure; the reason is in the error message on standard error
- 2: usage error, such as an unknown option, a missing one, a value out of range, two options that exclude each other, or an option that does not apply to the selected target
- 3: the security configuration carries irrecoverable or lockout consequences and `--allow-irreversible` was not given (see *Consequences of a Security Configuration* on page 56)
- 5 (`load`): the port, the probe or the target could not be reached, or the link to it failed during programming; a retry may succeed
- 6 (`load`): the target was reached but rejected the operation, for example a failed verify, a refused certificate or a C29 debug gate that refused the password
- 130: interrupted with Ctrl-C

For `load`, a wrong `--passphrase` and a `--port` of the other kind than the container's target (a serial port for a JTAG container, or a debug probe for a serial one) are usage errors, status 2. A `--port` value that C2Prog cannot use also reports status 2. This is the case for the following:

- port options that are not a valid JSON object
- port options on a serial port or a debug probe, which accept none for `load`
- a CAN port option other than `ta` and `sa`
- a CAN adapter prefix that C2Prog does not know, for example `canfoo:1`
- a `ta` or `sa` port option that is not valid

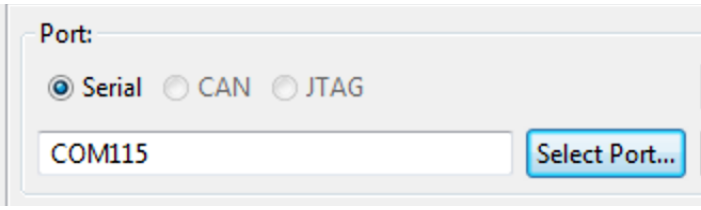
When the CAN address is stored in the container, the same error reports status 1.
Status 2 covers a wrong option or a wrong option value; correcting the command line fixes it. A `--seccfg-template` that matches no template in the installation belongs here: the name is a value out of range. Status 1 covers what fails after C2Prog accepts the command line: a file that is not there, an image that is truncated, a security configuration that fails validation, a key that does not match the fingerprint given for it. A `--config-file` is read like any other input file, so an option written wrong inside it reports 1.

# 5 Programming Workflows

This section describes programming a target over each supported physical interface.

## 5.1 Programming over Serial Link

The serial interface can be used with TI's SCI/UART bootloader as well as customer-specific bootloaders developed by CodeSkin. Please refer to the TI Boot ROM Reference Guides for information on how to configure your target for SCI programming. To operate with the C2Prog default settings, the serial link must support communication at 115200 baud and full-duplex transmission. Slower links and half-duplex operation are supported but will require custom settings.




C2Prog works most reliably with converters that use the FTDI chipset, for example the Parallax USB to Serial Adapter. The serial port of TI's evaluation hardware also works well with C2Prog.

### 5.1.1 Serial over TCP/IP

The serial protocol also runs over a TCP connection, for example with targets that are reached through a Lantronix serial-to-Ethernet or serial-to-WiFi converter. In place of a COM port, the port configuration then names a socket, `socket:<host>:<port>`, where `<host>` is the host name or IP address of the converter and `<port>` the port it listens on. For example, **socket:192.168.0.100:1001** reaches a converter at that address. The same string is used on the command line:

```
c2p-cli load --port=socket:192.168.0.100:1001 test.ehx
```

Sockets are not discoverable, so the port-selection dialog does not list them: type the configuration string into the **Port** field. C2Prog gives up after 15 seconds if the connection is refused or unanswered.

---

The baud rate of the selected target option applies to the link between the converter and the MCU, not to the TCP connection. Configure the converter to match the line settings of the target.

---

## 5.2 Programming over JTAG

C2Prog supports TI XDS emulators for flash programming and in conjunction with the GDB server (see Section 8.1 on page 75).

When you install C2Prog on Windows, an option is provided to also install TI's XDS drivers and EmuPack. Alternatively, you may point C2Prog to an existing installation of the EmuPack, for example within Code Composer Studio (CCS) or UniFlash.

The manual configuration is performed using the **c2p-cli** executable:

```
c2p-cli set ccs-base-path C:\ti\ccs<version>\ccs\ccs_base
```

If you are using CCS for code development, it may be desirable that C2Prog use the same EmuPack version. Referring to an existing installation of the EmuPack can also save disk space, and it keeps the probe firmware stable: two EmuPack versions on one machine make each tool upgrade or downgrade the XDS110 firmware as you switch between them (Section 5.2.2).

---

Using the agentic interface (Section 7.2), you can ask an AI assistant to configure this: give it the path to your EmuPack (or CCS **ccs_base** directory) and it stores the `ccs-base-path` and confirms that JTAG programming is available.

---

---

On non-Windows machines, manual configuration is required to enable programming over JTAG.

---

---

Select the correct external clock/crystal frequency: C2Prog cannot verify the frequency over JTAG, and a wrong frequency may damage the MCU.

---

### 5.2.1 Port configuration

The port configuration must be formatted as follows:

- `xds100v<v>:<sn>`
- `xds110:<sn>`
- `xds110-2w:<sn>`
- `xds2xx:<sn>`

where <v> must be equal to “1”, “2”, etc., and <sn> specifies the serial number of the emulator (only needed if several of the same type are connected).

The “2w” suffix selects 2-pin cJTAG (IEEE 1149.7) instead of 4-wire JTAG. In this mode the probe drives the target over TMSC and TCKC alone, using the OSCAN2 scan format, which is how boards that do not route TDI and TDO to the debug header are reached. The wiring decides which form is correct: a target wired for 4-wire JTAG does not respond to `xds110-2w`, and one wired for cJTAG does not respond to `xds110`: the probe then fails to open with `Error -1050`, which the EmuPack describes only as an invalid parameter.

Some examples for valid configuration strings include:

- `xds100v1`, `xds100v2`, `xds100v3`, `xds110`, `xds110-2w`
- `xds100v2:TIUDCI83`

The port-selection dialog scans for connected XDS emulators and lists each detected probe with its serial number, so the configuration string can be selected directly rather than typed; the **Scan Ports** button repeats the scan. When only one emulator of a type is present, both the plain (e.g., `xds110`) and serial-qualified (e.g., `xds110:<sn>`) forms are offered. The scan cannot tell how a target is wired, so an XDS110 is listed in both the 4-wire and the `-2w` form; pick the one that matches the debug header of your board.

---

Multiple XDS2xx emulators are distinguished by serial number only on Windows; on macOS and Linux a single XDS2xx is detected.

---

---

You can ask an AI assistant to list the connected debug probes; it reports each XDS emulator with its serial number and the port configuration string that selects it.

---

### 5.2.2 XDS110 Considerations

C2Prog can be configured to automatically upgrade/downgrade the firmware of XDS110 emulators. This feature is enabled by default but may be disabled using the **c2p-cli** executable as follows:

```
c2p-cli set xds110-auto-update off
```

Turning it off keeps the firmware that another tool, typically CCS, installed on the probe.

Should your XDS110 emulator become unresponsive after an upgrade/downgrade, open a command-line prompt, navigate to the `ccs_base\common\uscif\xds110` folder and execute the following sequence of commands to restore the firmware.

```
xdsdfu -m
xdsdfu -f firmware.bin
```

The location of the EmuPack used by C2Prog can be queried as follows:

```
c2p-cli get ccs-base-path
```

### 5.2.3 Changing an Emulator Serial Number

When several emulators of the same type are connected, each is selected by its serial number in the port configuration (Section 5.2). The serial number of a particular unit is reported by the port-selection dialog, and can also be read and changed with the EmuPack utility for the emulator type:

- XDS100 family: the `xds100serial` utility in the `ccs_base\common\uscif` directory.
- XDS110: the `xdsdfu` utility in the `ccs_base\common\uscif\xds110` directory.
- XDS2xx: the `xds2xx_conf` command in the `ccs_base\common\uscif\xds2xx` directory.

For an XDS110, open a command-line prompt in the `ccs_base\common\uscif\xds110` folder and switch the probe into DFU mode before assigning a new serial number, then reset it:

```
xdsdfu -m
xdsdfu -s NEWSERL1 -r
```

The `-s` option replaces the entire serial number (up to eight characters, no spaces); `-n` instead sets a four-character suffix and preserves any board prefix.

An XDS100 keeps its serial number in the FTDI EEPROM, where the **xds100serial** utility writes it.

An XDS2xx stores its serial number in the unit, and `xds2xx_conf get xds2xxu 0` reads it whether or not serial-number reporting is enabled. By default the probe does not report the serial number over USB, so the debug driver cannot tell several connected XDS2xx probes apart. The following invocation enables serial-number reporting for multi-emulator support:

```
xds2xx_conf set xds2xxu 0 EnableUSBSerial=true
```

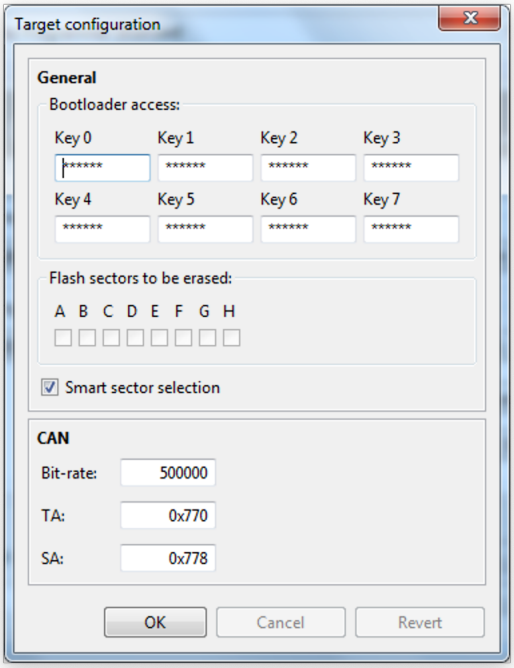
The **ReadMe.txt** next to the utility documents its full syntax.

## 5.3 Programming over CAN

C2Prog allows programming over CAN using Unified Diagnostic Services (UDS) as defined in ISO-14229/15765.

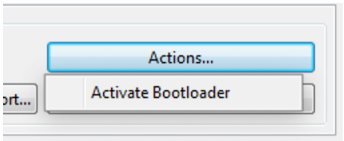
Reflashing over CAN works only in conjunction with a custom CodeSkin CAN bootloader resident on the target: the CAN boot mode of the C2000 boot ROM is too limited for C2Prog to build a programming session on.

A special convention is used to allow for the configuration of custom bit-timings. In lieu of a normal bit rate, e.g., ”250000”, it is possible to specify the BTR0 and BTR1 values directly as ”0x8000<BTR1><BTR0>”, where BTRx stands for the bit-timing registers of the SJA1000 (or compatible) CAN controller, assuming a clock frequency of 16 MHz. For example, a value of 0x80001C03 specifies 87.5% sampling at 125 kbit/s.




When configuring CAN identifiers in the **Configure…** dialog, they can be entered as hex numbers (with ‘0x’ or ‘$’ prefix) or decimal numbers. An ‘X’ postfix specifies an extended CAN identifier.

CodeSkin’s custom CAN bootloaders offer a mechanism to recover from unresponsive application code. This **Activate Bootloader** feature can be accessed from the **Actions…** menu.




Contact CodeSkin for custom CAN bootloading solutions.

### 5.3.1 CAN Adapters on Windows

On Windows, CAN hardware from Vector, Kvaser, Lawicel, NI, and PEAK-System is supported. The syntax for the port configuration is as follows:

- Lawicel: ”canlw:0” – only one port supported
- Kvaser: ”cankv:<n>” – where <n> is the port number (0 or 1)
- Vector: ”canvec:<n>” – where <n> is the port number (0 or 1)
- NI-XNET: ”canxnet:<n>” – where <n> is the port assigned to the adapter
- Peak CAN: ”canpk:<n>” – where <n> is the device number as configured in PCAN-View, e.g., 255; other channels on the same device are identified as ”255B”, ”255C”, etc.

### 5.3.2 CAN Adapters on Linux

On Linux, SocketCAN is used for accessing a wide range of supported CAN hardware. The port configuration is done according to the name of the device, i.e., the netdev name, as reported by `ifconfig -a` or `ip link show`, for example ”can0”.

SocketCAN interfaces must be configured (requires root privileges) from the command line before starting a programming session.

```
sudo ip link set can0 type can bitrate 500000
sudo ip link set up can0
```

# 6 Integrity and Security

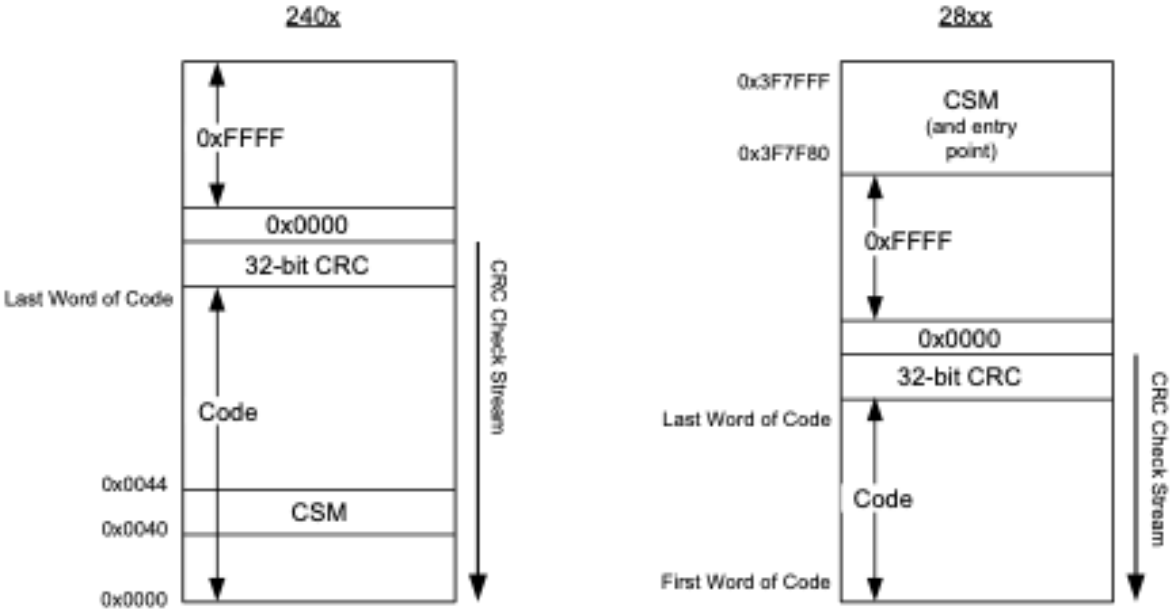
This section covers flash integrity verification and device security: the CRC checksum and code security module for F240x and F28x devices, and the certificate-based provisioning workflow for F29x devices.

## 6.1 F240x and F28x Devices

For F240x and F28x devices, C2Prog provides integrity verification and security unlocking mechanisms that operate directly on flash memory and the code security module (CSM).

### 6.1.1 CRC Checksum

C2Prog can be configured to automatically append a 32-bit CRC to the code being programmed into flash. This allows the embedded application to verify flash integrity at each power-up or even periodically during operation.




If the **Append CRC** box is checked, or the `--append-crc` command-line option is used, C2Prog will first parse the binary file and determine the lowest and highest addresses to be programmed. For a 240x MCU, this includes the CSM zone (4 keys); for a 28xx MCU, the CSM zone is ignored (including keys, reserved words, and program entry point). Next, the 32-bit CRC is calculated and appended at the top of the memory, i.e., at the two addresses above the highest address of the hex file determined before. In addition, a CRC delimiter, one zero word (0x0000), is placed above the two CRC words.

The range C2Prog covers is contiguous from the lowest to the highest programmed address. If the application leaves gaps between sections, for example code in one flash sector and constants in another, the words in those gaps are part of the CRC stream and are assumed to read 0xFFFF, the erased state. The runtime check must process the gap words as well; it cannot skip them.

---

Every sector the CRC range spans must be erased before programming, including a sector that holds only gap. Automatic sector selection erases only the sectors that carry data to be programmed, so a gap-only sector still holding stale data from an earlier build makes the on-device stream diverge from the value C2Prog computed, and the check never resolves to zero. Force-erase such sectors with manual sector selection (or the `--sector-mask` option).

---

The 32-bit CRC algorithm used has the following parameters:

- Polynomial: 0x04c11db7
- Endianness: big-endian
- Initial value: 0xFFFFFFFF
- Reflected: false
- XOR out with: 0x00000000

Test stream: 0x0123, 0x4567, 0x89AB, 0xCDEF results in CRC = 0x612793C3. Please refer to Appendix B on page 95 of this manual for a CRC32 implementation example.

In contrast to a typical data stream with the CRC transmitted at the end, the C2Prog CRC must be verified by processing the flash data starting with the CRC, i.e., one memory address below the CRC delimiter (0x0000). A successful data verify results in a CRC register value of zero (0x00000000).

A typical flash verification algorithm running at MCU power-up executes as follows:

1. Set a memory pointer to the highest possible program address.
2. Decrement the pointer until it points to the CRC delimiter (0x0000), skipping all 0xFFFF values.
3. Decrement the pointer by one more address (at which time it points to the first CRC word).
4. Initialize the CRC register to 0xFFFFFFFF.

5. Update the register with the value addressed by the memory pointer (CRC polynomial: 0x04C11DB7).
6. Decrement memory pointer.
7. Repeat 5-6 until the memory pointer reaches the lowest program address.
8. If, at this point, the register holds 0x00000000, then the data integrity has been successfully verified.

Sample Code for F28x with code in flash sector A:

```
#define FLASH_TOP (const uint16_t*)(0x3F7F7FL)
#define FLASH_BOT (const uint16_t*)(0x3F6000L)

const uint16_t* FlashPtr;
uint32_t CRCRegister;

FlashPtr = FLASH_TOP;
// search for CRC delimiter
while((*FlashPtr != 0x0000) && (FlashPtr > FLASH_BOT)){
  FlashPtr--;
}
// process stream, CRC first
CRCRegister = 0xFFFFFFFFL;
while(FlashPtr > FLASH_BOT){
  FlashPtr--;
  // each CRC32Step() shifts one byte into the CRC register
  CRCRegister = CRC32Step(((*FlashPtr >> 8) & 0xFF), CRCRegister);
  CRCRegister = CRC32Step(((*FlashPtr >> 0) & 0xFF), CRCRegister);
}
// at this point CRCRegister should be reading zero
```

The **Tools…> Firmware Info** dialog reports the image start address, that is, the lowest programmed address. `FLASH_BOT` must equal this value, since the calculation ends there and the register must read zero. `FLASH_TOP` is only a ceiling for the delimiter search: it must sit at or above the CRC delimiter, with every word between it and the delimiter erased (0xFFFF), so it is normally set to the top of the flash region and need not match the address at which the CRC was added.

### 6.1.2 Code Security Module (CSM)

C2Prog will unlock the code security module (CSM) using the keys provided with the `--keys` command-line option or **Configure…** dialog. In the case of 32-bit keys, each key is split into two 16-bit keys in big-endian (BE) order.

For older C2000 MCUs with the keys located at fixed locations in erasable flash memory, C2Prog will extract the keys from the firmware image if default keys of all 0xFFFF are provided.

If the MCU has the PSWDLOCK mechanism, specifying all 0x0000 in C2Prog (`--keys`, **Configure...**) will instruct C2Prog to unlock the CSM by reading the unprotected password locations in OTP.

---

The security zone is not protected until the corresponding PWDLOCK is set. Review the relevant TI documentation.

---

**Dual Code Security Module (DCSM)** In the case of MCUs with dual code security module (DCSM), C2Prog will automatically determine which zone must be unlocked to program a particular firmware image.

---

It is only possible to program one security zone at a time.

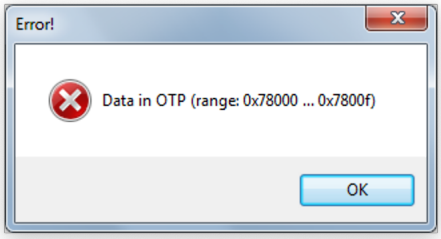
---

Do not set the PWDLOCK of a zone unless the sector assignment for that zone has been fully configured by either explicitly requesting, or not requesting, the zone for each sector. See TI technical documentation of GRABSECT and GRABRAM registers.

---

### 6.1.3 F28x OTP

C2Prog helps prevent unintentional writing to one-time programmable (OTP) memory. Unless the user generates the Extended Hex file with the `--allow-otp` option or checks the **Allow OTP Programming** box, C2Prog will not permit the programming of OTP locations.



When programming OTP, C2Prog is very strict. If the programming data includes a “1” for a location that has already been (factory) cleared to ”0”, C2Prog will report an error.

C2Prog automatically pads programming data with 0xFFFF to comply with the block sizes and alignment that the Flash API expects. This padding could result in the violation stated above.

---

All memory locations of an OTP block being programmed must be explicitly specified in the firmware image, taking into account bits that have already been (factory) cleared.

---

## 6.2 F29x Devices

F29x devices protect firmware through two mechanisms. *Code authentication* determines which images the device accepts. After key provisioning, every image must carry an X.509 certificate signed with the customer key, and the Hardware Security Module (HSM) verifies that certificate before it programs or launches an application. *Access control* determines what code on the device may access. The Safety and Security Unit (SSU) checks memory and peripheral accesses against the security configuration, which also sets flash write protection and debug access.

F29x devices move through the security states HS-FS (High Security-Field Securable), HS-KP (High Security-Key Provisioned) and HS-SE (High Security-Security Enforced). Provisioning follows these states:

1. In the HS-FS state, one can program and test the application directly from the C2Prog GUI or with `c2p-cli mkehx`. No signing is required. The flash bank mode can be set freely in this state, by a programming session or by key provisioning (Section 3.4.2). The security configuration can be developed and tested here as well, with `mkehx --seccfg` (Section 6.2.2).
2. Key provisioning writes the customer keys to OTP (Section 6.2.3). It takes three commands: `mkkeys` to generate the keys, `mkkpcert` to build the certificate, and `mkehx-kp` to package it. The device enters HS-KP at the next power-on reset. From this point on, every image programmed into the device must be signed (Section 6.2.4).
3. Programming the initial security configuration with `mkehx-cp` takes the device to HS-SE at the next power-on reset (Section 6.2.5). The application can be programmed in the same session.
4. In the HS-SE state, `mkehx-cp` provisions the application code and security configurations (Section 6.2.6).

On an HS-FS device, C2Prog either programs the certificates the firmware image carries or

generates placeholder certificates.

---

C2Prog supports programming in the HS-FS state on every F29x silicon revision. Key provisioning and code provisioning (steps 2 to 4) are only supported on silicon revisions B and more recent.

---

Extended Hex files for key-provisioned devices can only be generated with the command-line tool **c2p-cli**, for example as a CCS post-build step. The command signs every certificate the file needs, including those of the secondary bootloaders (the C2Prog flash kernels), the HSM runtime, and the security configurations.

### 6.2.1 Security States, Security Configuration, and Certificates

**Security States** Every F29x device leaves the TI factory in the HS-FS state. The state can only advance, from HS-FS to HS-KP and from HS-KP to HS-SE, and no transition can be reversed.

---

Each change of state takes effect at the next power-on reset.

---

In the **HS-FS** state no signature is checked, and the C2Prog flash kernel programs the flash from CPU1, without the HSM. However, JTAG access and flash programming are still subject to the active security configuration (*The Security Configuration* below). TI's factory configuration leaves JTAG debug of the C29 CPUs open and lets a flash kernel program the application flash. A stricter configuration can restrict both. In **HS-FS**, the HSM cannot be reached through its regular JTAG access port.

The flash bank mode and the security configuration become fixed in the later states. Use the HS-FS state to set the flash bank mode (Section 3.4.2) and to develop and test the security configuration (Section 6.2.2).

A device reaches the **HS-KP** state at the first power-on reset after the TI Keywriter has programmed the customer keys into OTP memory (Section 6.2.3). In this state the CPU can no longer write its own flash. The CPU cannot read the flash either, and the bank mode committed in HS-FS takes effect again once the device reaches HS-SE. All programming must go through the HSM, and the HSM accepts only images with certificates signed with the provisioned key. C2Prog programs an HS-KP device only with a file that carries at least a security configuration, which takes the device to HS-SE (Section 6.2.5). The CPU1 and CPU3 application can be programmed in the same session as the security configuration.

**HS-SE** is the state in which devices typically ship to the field. The boot ROM checks the flash image against the keys on every boot, the HSM programs all updates from signed images, and the active security configuration controls memory access, flash write protection, and debug access (Section 6.2.6).

**The Security Configuration** The security configuration (seccfg) consists of a set of data blocks, one per CPU, that hold the protection rules of the device: which memory each CPU and each of its execution contexts may access, flash write protection, debug access, boot options, and the update owners that may change these rules later. The blocks are always programmed together. An F29x part without a CPU3 carries no CPU3 block. Their location in flash depends on the bank mode.

The security rules are enforced by the Safety and Security Unit (SSU), the hardware unit of the F29x device that checks memory and peripheral accesses against the security configuration. In the HS-FS and HS-SE states, the SSU applies the security configuration stored in flash. In the HS-KP state, the boot ROM cannot read flash, and no stored configuration is applied.

The configuration assigns code to LINKs. A LINK is an execution context with its own access rights. The configuration also selects the SSU mode, called SSU Mode 1 to 3 or SSUMODE1 to SSUMODE3.

The mode selects how much of the configuration the SSU enforces. In SSUMODE1 the access-protection regions are disabled. SSUMODE2 and SSUMODE3 enforce them against all code, including the C2Prog flash kernel. SSUMODE3 additionally binds the debug gates, the write and erase protection, and the update owners that may replace these rules later.

The seccfg blocks are produced in one of two workflows. In the *combined* workflow, TI's SysConfig generates the blocks inside the application project, and the linker places them into the **.out** file with the code. In the *separate* workflow, a stand-alone **.lua** profile describes the configuration, which C2Prog subsequently compiles into the blocks (Section 6.2.9).

The device keeps two copies of the configuration in dedicated flash sectors. A new configuration goes into the inactive copy, and only the reset at the end of the session switches over to it. In the event that such an update is interrupted, the active copy stays intact and the device keeps its previous rules.

On an HS-FS device the configuration is written without signing. On an HS-KP device, programming the first signed configuration takes the device to HS-SE. On an HS-SE device, only a signed update through the configured seccfg update owner can change the configuration.

In SSU Mode 2 and 3, a later session can replace a security configuration only if the active configuration still lets the flash kernel run (page 59).

The agentic interface (Section 7.2) can answer questions on this topic directly from the manual (`search_docs`) and explain what a given security configuration does (`explain_seccfg`, `mkehx --explain-only`).

**Certificates** A certificate is an X.509 structure attached to an image. It names the target CPU and the load address, carries a hash of the image and a software revision number for anti-rollback, and is signed. The certificate proves who signed the image and that the image is unchanged. It may also specify other parameters, for example parameters related to encryption and debug access.

On an HS-FS device the boot ROM only checks that the certificate has the correct format. The signing key plays no role yet, and C2Prog generates placeholder certificates automatically. Once the device has keys, the HSM validates the certificate of every image (application code, secondary bootloader, HSM runtime, and security configuration) against the Manufacturer public key stored in OTP. C2Prog creates and signs these certificates when it builds the Extended Hex file (Section 6.2.4).

The Secondary Manufacturer key pair (SMPK), and optionally a Backup key pair (BMPK), are written into OTP during key provisioning. The BMPK is a spare: if the SMPK is ever compromised, the device can switch its root of trust to the BMPK. This works only if the BMPK was written during key provisioning.

### 6.2.2 Security Configuration in HS-FS

The HS-FS state is well suited to developing and testing a security configuration. In this state, C2Prog writes the configuration directly, without the HSM and without signing. An SSUMODE1 configuration can be reprogrammed without restrictions. After an SSUMODE2 or SSUMODE3 configuration, the next session works only if that configuration still lets the flash kernel run (page 59). Both workflows (Section 6.2.1) apply, without the signing steps.

In the combined workflow, the **.out** file carrying the application and its seccfg blocks is packaged with the `mkehx` command, or loaded in the C2Prog graphical interface, with the same target as for any other application:

```
c2p-cli mkehx --target=29H85xTU9,TU8-CPU1_UART --allow-seccfg my_app.out
```

A SysConfig project that configures the SSU emits these blocks into the application's **.out**. Programming them with C2Prog requires an explicit opt-in: the `--allow-seccfg` option on the command line, or the *Allow Security Configuration Programming* checkbox in the graphical interface. Without it, C2Prog refuses an image that carries a security configuration.

On HS-FS devices, `--allow-seccfg` also acknowledges non-default update owners in the configuration, as `--allow-irreversible` does on the signed routes.

In the separate workflow, the `--seccfg` option specifies a **.lua** profile, or `--seccfg-template` selects a built-in template. C2Prog compiles the profile into the per-CPU blocks and embeds them in an Extended Hex file:

```
c2p-cli mkehx --target=29H85xTU9,TU8-CPU1_UART --seccfg-template
    ↪ default_seccfg_ssumode1
```

The resulting ehx file is named after the profile unless an output path is given. `--allow-seccfg` is not needed with `--seccfg` or `--seccfg-template`. An application image may be specified with the security configuration. C2Prog then merges the security configuration blocks into it, such that the ehx file can program the application and the configuration in a single session:

```
c2p-cli mkehx --target=29H85xTU9,TU8-CPU1_UART --seccfg my_seccfg.lua
    ↪ my_app.out
```

An application image that also carries seccfg blocks, or other data in the seccfg flash sectors, is refused.

To build one profile for different SSU modes without editing it, use the `--ssu-mode` option. It builds the profile named by `--seccfg` with its `ssu_mode` field overridden and leaves the file unchanged:

```
c2p-cli mkehx --target=29H85xTU9,TU8-CPU1_UART --seccfg my_seccfg.lua --
    ↪ ssu-mode 1
```

The `--ssu-mode` option requires `--seccfg` or `--seccfg-template`. A security configuration contained in a binary cannot be rebuilt. The `mkehx-cp` command accepts the same option.

The Extended Hex files that `mkehx` creates program only HS-FS devices. C2Prog refuses them on a key-provisioned device, which must be programmed with the signed workflows of Sections 6.2.5 and 6.2.6.

---

Before programming a new security configuration, generate and carefully inspect the consequence report (page 56). If the report contains irrecoverable or lockout findings, `mkehx` stops with status 3 and writes no file. To write the file anyway, pass `--allow-irreversible`.

---

### 6.2.3 Key Provisioning

Key provisioning takes an F29x device from the HS-FS state to the HS-KP state. It programs the customer keys, together with the OTP fields the key-provisioning certificate defines, into the device's one-time programmable (OTP) memory. The keys become the root of trust for all later operations.

---

⚠ The flash bank mode can no longer be changed once the device has been key provisioned. Key provisioning commits the bank mode of the selected `--target`. Therefore, select the target for the bank mode in which the finished device is to run. A device that is to run both cores needs a dual-core target such as **29H85xTU9,TU8-CPU1+CPU3_UART**. See Section 3.4.2 on page 16 for details on F29x bank modes.

---

**Prerequisites** Key provisioning requires two files that must be obtained from Texas Instruments. Both are in TI's OTP Keywriter package for F29H85x devices. They are specific to the silicon revision. Revisions B and C, for example, use the `SR_20` build:

- **TI Field Encryption Key (FEK)**: the public key used to encrypt the secrets in the certificate. `mkkpcert` reads it through `--ti-fek`. In the package:

  **sbl_keywriter/scripts/cert_gen/common/tifek/f29h85x/SR_20/ti_fek_public.pem**

- **Keywriter image**: the HSM runtime that performs the OTP write on the device. `mkehx-kp` reads it through `--otp-kw`. In the package:

  **sbl_keywriter/keywr_bin/f29h85x/SR_20/otp_kw_f29h85x_hs_fs.hsmimage.bin**

Contact Texas Instruments support to obtain the package.

C2Prog can generate the Manufacturer keys and the key-provisioning certificate, and ships the HSM runtime that code provisioning uses. The TI CyberShield Toolkit is another source for generating the keys and the certificate; This route is described in Section 6.2.3.

**Generating the Manufacturer Keys** The Manufacturer keys are the Secondary and Backup signing keys (SMPK and BMPK) and the Secondary and Backup encryption keys (SMEK and BMEK, AES-256). The `c2p-cli mkkeys` command generates all four and writes them to a *keyset*, a password-encrypted key store on disk. The file uses CST's session format and is therefore compatible with the toolkit:

```
c2p-cli mkkeys --keyset-password=secret MyKeyset.session
```

The `--key-algorithm` option selects the signing algorithm, one of `rsa4k` (the default), `secp256r1`, `secp384r1`, `secp521r1` and `brainpool512`. The `--key-algorithm-backup` option specifies the algorithm for the Backup key. The encryption keys are AES-256 in every case.

The run prints the SHA-256 fingerprint of each signing key on standard error and writes nothing to standard output. This fingerprint is the value to pass as `--sign-pubkey-fp`.

---

⚠ The device holds a hash of the signing key in OTP. A device provisioned with a key whose private half is lost can never be programmed again. Therefore, at least one readable copy of the Manufacturer keys must be preserved, whether that copy is this keyset, a PKCS#11 token, or a backup of either. It is recommended that such a keyset be stored on offline media, separately from its password.

---

**Generating the Key-Provisioning Certificate** Once the Manufacturer keys are produced, the key-provisioning certificate must be generated.

The `mkkpcert` command of the **c2p-cli** command-line utility builds the key-provisioning certificate, the binary that `mkehx-kp` subsequently packages for the keywriter, from the Manufacturer keys stored in a keyset:

```
c2p-cli mkkpcert --ti-fek=ti_fek_public.pem --keyset=MyKeyset.session --
    ↪ keyset-password=secret --msv=0x1E22D --keycnt=2 --keyrev=secondary
    ↪ --sr-sbl=1 --sr-hsmrt=1 --sr-app=1 --sr-ssu=1 final_certificate.bin
```

The TI Field Encryption Key (FEK) public key, **ti_fek_public.pem**, is specific to the device silicon revision. Use the build at the path given under *Prerequisites* on page 44. The keywriter refuses a certificate built with the wrong public key.

The options below write the OTP fields that make up the device's root of trust:

- `--ti-fek` names the TI Field Encryption Key public key of the silicon revision being provisioned. The certificate's key material is encrypted to it, so only the HSM of a device of that revision can read it.
- `--msv` is the Manufacturer Specific Value, a hexadecimal number of your choosing, up to six hex digits on F29H85x. When the option is omitted, the field is left unprogrammed.
- `--keycnt` is the number of Manufacturer key sets programmed, 1 (Secondary only) or 2 (Secondary and Backup, the default). `--keyrev` selects which one the device treats as current, `secondary` for the SMPK or `backup` for the BMPK, and may not name a set

the key count leaves out. Revoking a compromised Secondary key means moving the device to the Backup set.

- `--sr-sbl`, `--sr-hsmrt`, `--sr-app` and `--sr-ssu` set the minimum software revisions of the anti-rollback policy of the F29x boot flow: an image whose software revision is below the minimum for its type is rejected. Each argument is a revision count, at most 32 (64 for `--sr-ssu`), which the certificate carries as that many consecutive one bits. Each defaults to 1. `--sr-sbl` applies to the secondary bootloader and also to the CPU1 and CPU3 application certificates that C2Prog generates. `--sr-hsmrt` applies to the HSM runtime and `--sr-ssu` to the security configuration. `--sr-app` applies to a TI application certificate type that C2Prog does not generate. Raising `--sr-sbl` or `--sr-hsmrt` above 1 leaves the device unable to accept the C2Prog flash kernel and HSM runtime, which always carry revision 1 (see the warning below).

- `--ext-otp` programs the customer extended OTP region. The value is given as hexadecimal digits.

- `--ext-otp-index` specifies the bit offset of the value within the 1664-bit region. The keywriter programs the region in whole bytes. Therefore, the offset and the size must be multiples of 8, and together they must stay inside the region.

- `--ext-otp-size` specifies the number of bits the value occupies and defaults to the length of `--ext-otp`. A size below that length is refused. A larger size pads the payload with zero bytes at its most significant end, because the payload is a big-endian number.

- `--ext-otp-protect` specifies the protection mask, a 64-bit value that the field carries twice, once for write protection and once for read protection.

- `--mpk-opt` and `--mek-opt` set the Manufacturer key and encryption key option fields, 10 and 5 bits wide. Both are reserved on F29H85x.

- `--keys` names the Manufacturer key fields to program, `smpk`, `smek`, `bmpk` and `bmek`, all four by default. A device provisioned without a Backup key has no second key set to revoke to, and its certificate is signed by the Secondary key alone.

- `--protect` names the fields whose OTP rows are closed to later writes and reads. It takes `keys` (the four Manufacturer key fields, the default), `all`, `none`, `swrev`, or a comma-separated list of `smpk`, `smek`, `bmpk`, `bmek`, `msv`, `keycnt`, `keyrev`, `sr-sbl`, `sr-hsmrt`, `sr-app`, `sr-ssu`, `ext-otp`, `mpk-opt` and `mek-opt`. If `keyrev` is left out of `--protect`, the device can still move to the Backup key set later. The values `sr-app` and `sr-ssu`, and therefore `swrev` and `all`, are not yet supported. With them, the keywriter fails with **No response to write keys**.

- `--override` takes the same names and lets the keywriter program a field whose eFuse rows already hold a value. Because the field is OTP, such a write can only set additional bits.
- `--signing-algorithm` selects the RSA signature scheme, `pkcs1v15` (the default) or `pss`.
- `--sign-pubkey-fp` and `--sign-pubkey-fp-backup` name the Secondary and Backup Manufacturer keys on a PKCS#11 token by fingerprint, with `--pkcs11-pin` for the token. The keys never leave the token. Both signing keys must come either from the token or from the keyset.

Each signing key signs twice, once for the payload key and once for its own certificate. A token that requests the PIN for every signing operation should therefore be configured with a PIN policy that covers the whole session.

---

The F29H85x keywriter only accepts `pkcs1v15`. It refuses a certificate signed with `pss`.

---

Make sure that you carefully review the `mkkpcert` consequence report. A protected OTP field can never be changed again on that device. Key provisioning itself is irreversible: it moves the device out of HS-FS, and from then on it accepts only images signed with these keys. The software revision fields set an anti-rollback floor, and a certificate below the floor for its type no longer authenticates. `--swrv` sets the revision of the application and security-configuration certificates. The C2Prog flash kernel and HSM runtime always carry revision 1, so an `--sr-sbl` or `--sr-hsmrt` floor above 1 leaves the device unable to accept them.

---

**Using the TI CyberShield Toolkit Instead** This route is optional. It is intended for workflows that are already built around the toolkit.

CST produces the key-provisioning certificate (**final_certificate.bin**) that the `mkehx-kp` command takes, building it from the Manufacturer keys of a keyset.

CST refers to the keyset as a *session* and uses the same file format as `mkkeys`. Therefore, either tool can open a file written by the other. The CST command-line tool is **spt**. Refer to the TI CyberShield Toolkit documentation for its sub-commands and options.

Pass the **final_certificate.bin** CST writes to C2Prog as shown under *Creating the Extended Hex File* below.

**Creating the Extended Hex File** Once the certificate binary has been generated, by `mkkpcert` or by CST, use the **c2p-cli** command-line utility to create an Extended Hex file:

```
c2p-cli mkehx-kp --target=29H85xTU9,TU8-CPU1_UART --otp-kw
    ↪ otp_kw_f29h85x_hs_fs.hsmimage.bin final_certificate.bin
```

The `mkehx-kp` command packages the bootloader, HSMrt keywriter image and certificate payload with their programming instructions into an Extended Hex file (**final_certificate.ehx**).

The `--target` option specifies the device type, the bank mode, and the communication protocol. In this example, **_UART** refers to the UART boot mode. Consult the TI Boot ROM documentation for your specific F29x device to configure the target hardware to boot in UART mode.

The `--otp-kw` option specifies the TI Keywriter HSM runtime image, **otp_kw_f29h85x_hs_fs.hsmimage.bin**. The OTP Keywriter package holds one build per silicon revision, for example under

```
sbl_keywriter\keywr_bin\f29h85x\SR_20\otp_kw_f29h85x_hs_fs.hsmimage.bin
```

Pass the full paths if the files are not in the working directory:

```
c2p-cli mkehx-kp --target=29H85xTU9,TU8-CPU1_UART --otp-kw C:\ti\
    ↪ otp_keywriter_f29h85x_SR_20_<version>\sbl_keywriter\keywr_bin\
    ↪ f29h85x\SR_20\otp_kw_f29h85x_hs_fs.hsmimage.bin C:\keys\
    ↪ final_certificate.bin
```

For key provisioning via JTAG, use the corresponding **_JTAG** target.

---

⚠ Key provisioning also configures the device bank mode. Make sure that you are using the correct `--target` parameter. Once the device is key provisioned, the bank mode can no longer be modified.

---

---

⚠ Create the first key-provisioning file with the `--dry-run` option. A dry run goes through the complete workflow without writing keys to OTP. However, it does commit the bank mode. Once the dry run succeeds, create the file again without the option to provision the keys.

---

The `mkehx-kp` command bundles the TI Keywriter HSMrt image with the SBL and the key-provisioning certificate. The resulting Extended Hex file is therefore subject to the licensing conditions under which you obtained the TI Keywriter HSMrt image from TI, both when you distribute it and when you use it.

**Reviewing the Certificate** Key provisioning is a one-time operation. Therefore, review the certificate carefully before programming it. The `--explain-only` option decodes the certificate payload, prints a consequence report, and exits without writing an Extended Hex file. `--explain-only` requires only the certificate. No target selection, Keywriter image, or device connection is needed.

```
c2p-cli mkehx-kp --explain-only final_certificate.bin
```

The report classifies each OTP field the certificate programs as *flexible* or *restricted*. A restricted field is write- or read-protected in OTP and can never be changed after provisioning. A flexible field leaves room for later updates, such as switching to the backup key set or raising an anti-rollback floor.

The report also states the SHA-512 hashes of the signing public keys (the SMPK, and the BMPK in a dual-key chain). These keys become the device's root of trust once provisioned. Verify the hashes against your key inventory before programming. Each key is also listed with its SHA-1 fingerprint, which is the value to pass as `--sign-pubkey-fp` when signing application code or a security configuration with `mkehx-cp` (see *Application Code Provisioning* below).

A certificate that carries a single key set, or one that activates the backup key set and thereby revokes the primary key, is flagged at the irrecoverable level. The key values themselves are encrypted for the device and are not shown. The Keywriter validates them on-chip.

`mkehx-kp` has no `--allow-irreversible` option and does not stop on irreversible findings. Review the report before programming.

When `--explain-only` is combined with `--dry-run`, the report describes the dry run. It lists no irreversible items, names the fields that a run without `--dry-run` would program into OTP, and lowers every warning to the caution level. Both review forms exit with status 0 after printing the report, or with status 1 when the file holds no key-provisioning certificate.

For build pipelines and AI assistants, `--explain-only-json` emits the report as one JSON document on standard output. It uses the same schema as the security-configuration report

(see *The JSON Report* on page 57), with `report` set to `c2p-kp-explain`. The following fields are specific to key provisioning:

- `kp_present`: false when the file holds no key-provisioning certificate.
- `dry_run`: true for the report of a dry run.
- `images`: the SHA-256 of the certificate payload, which ties an archived report to the exact bytes it describes.

**Programming the Device** The key provisioning Extended Hex file can be programmed using either the C2Prog graphical interface or the command-line utility.

The session ends with a reset, but the device enters HS-KP only at the next power-on reset. Until then it still reports HS-FS, and C2Prog cannot tell it apart from a device without keys. Power-cycle the device before starting another session. The HSM converts a device to HS-SE only when the device has started in HS-KP.

After the power cycle, every image programmed into the device must be signed with the provisioned key. The next step is the initial security configuration, which can carry the application as well (Section 6.2.5).

### 6.2.4 Signing Methods

From the HS-KP state onward, every image programmed into the device must be signed. The X.509 certificates that authenticate an image are signed with the private keys established during key provisioning. C2Prog generates and signs these certificates when it creates the Extended Hex file.

The `mkehx-cp` command accepts three signing methods:

For testing with PEM-encoded private key files:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-key-file smpk.pem
    ↪  binary.out
```

For production with PKCS#11-based signing:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 binary.
    ↪ out
```

The `--sign-pubkey-fp` option specifies the SHA-1 (or SHA-256) fingerprint of the public key corresponding to the private signing key stored in the PKCS#11 device. The `--pkcs11-pin` option provides the PIN for accessing the security token.

For development or testing with a key held in a keyset:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --keyset MyKeyset.
    ↪ session --keyset-password secret binary.out
```

A *keyset* is the password-encrypted key store described on page 44, holding the Secondary and Backup Manufacturer signing and encryption keys. `--keyset` names the file and `--keyset-password` the password protecting it.

A keyset written by CST is stored in the CST session directory. Refer to the TI CyberShield Toolkit documentation for its location, and give `--keyset` the full path.

By default the Secondary Manufacturer key (SMPK) is used for signing. To sign with the Backup key (BMPK) instead, add `--keyrev backup`.

The three signing methods are interchangeable across every `mkehx-cp` invocation in this section, including the security-configuration and application-code commands below. The remaining examples use the PKCS#11 form for brevity. Substitute `--sign-key-file` or `--keyset` as appropriate for your key storage.

---

The file-based signing methods, a PEM private-key file (`--sign-key-file`) or a keyset (`--keyset`), are intended for development and testing only. In production environments, the PKCS#11 approach (`--sign-pubkey-fp`) must be used, with the signing key generated in and confined to a hardware security module or security token. Refer to Appendix D on page 99 for information on configuring PKCS#11 infrastructure and security tokens.

---

---

You can ask an AI assistant to help assemble a `mkehx-cp` command: it can list the available target IDs and look up the signing methods and option flags documented in this section.

---

### 6.2.5 Initial Security Configuration

Programming a security configuration takes the device from the HS-KP state to the HS-SE state. C2Prog programs the security configuration blocks of every CPU in a single operation. On an HS-KP device, a file that carries only part of the set, or only application images, is refused.

The file can carry the CPU1 and CPU3 application as well. C2Prog then programs the application images first and the security configuration last. If an image fails to program, the device stays in HS-KP and the session can be repeated.

Program the application in this session. An application programmed in the HS-FS state carries placeholder certificates, which fail authentication once the device is HS-SE. On an HS-SE device, the debug port opens only when the CPU1 application certificate permits it. Every certificate generated by C2Prog permits it. A device that reaches HS-SE without an authenticated application therefore refuses a JTAG connection until an application is programmed over UART.

For a device being provisioned for the first time, the recommended configuration is the built-in **ti_default_seccfg_bankmode_0_ssumode1** (**...bankmode_2...** for a bank-mode-2 device). These templates carry the per-CPU blocks of the security configurations Texas Instruments ships with the F29 SDK (**source/defseccfgbin**) with access-protection regions disabled, no write/erase protection of application flash, and the flash update owners set to the HSM. Programming one establishes the device's baseline security state and brings it to HS-SE, after which the device accepts signed application code. The application's own, tighter security configuration can still be provisioned later, once the application is being brought up (Section 6.2.6).

Security-configuration Extended Hex files are created with `mkehx-cp`. A built-in configuration is selected by name with the `--seccfg-template` option. A production configuration is supplied as a custom **.lua** profile with `--seccfg` (Section 6.2.9). The `mkehx-cp` command can be given an application binary as well, resulting in a single Extended Hex file that programs the application and the security configuration in one session:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1+CPU3_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 --
    ↪ seccfg-template ti_default_seccfg_bankmode_2_ssumode1 application.
    ↪ out
```

Without an application image, the file carries the security configuration alone:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 --
    ↪ seccfg-template ti_default_seccfg_bankmode_0_ssumode1
```

This command generates **ti_default_seccfg_bankmode_0_ssumode1.ehx**. Both files can be programmed using either the C2Prog graphical interface or the command-line utility.

The `--seccfg-template` argument is a template name without a path. The **.lua** extension is optional. The **ti_default_seccfg_...** templates are specific to a bank mode, like the SDK files they carry. The chosen template must therefore match the bank mode of the `--target`. The other built-in templates, such as **default_seccfg_ssumode1**, serve every bank mode, because C2Prog derives their flash placement from the `--target`.

Alternatively, Texas Instruments ships default security configurations as compiled **.out** images with the F29 SDK, one per flash bank mode, for example:

```
C:\ti\f29h85x-sdk_<version>\source\defseccfgbin\
    ↪ default_seccfg_bankmode_0_ssumode1.out
```

---

The bank mode compiled into the **.out** file must match the one selected by the `--target` option (Section 3.4.2). The file name states which mode it carries.

---

Because the seccfg blocks are compiled into the **.out** file, the file is passed to `mkehx-cp` as the image, without the `--seccfg` option:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ default_seccfg_bankmode_0_ssumode1.out
```

This command generates **default_seccfg_bankmode_0_ssumode1.ehx**. The built-in **ti_default_seccfg_...** templates carry the same files byte-for-byte and require no SDK installation. Their use is therefore recommended.

Power-cycle the device when the session has ended. The reset at the end of the session leaves the device in HS-KP. The device enters HS-SE at the next power-on reset. The device then boots the application in flash and accepts signed application code.

### 6.2.6 Provisioning in HS-SE

In the HS-SE state, two things can be provisioned: the application code and the security configuration. In this state Extended Hex files are produced with signed certificates.

**Combined Workflow (CCS and SysConfig)** The linker places the SysConfig-generated blocks into the **.out** file as the **.TI.bound:CPUn_Cfg** sections, located at the seccfg flash addresses. `mkehx-cp` detects the configuration by those addresses and produces one Extended Hex file that carries both the application and its security configuration (see *Application Code Provisioning* below).

**Separate Workflow (Lua Profile)** The **.lua** profile is compiled and signed into an Extended Hex file, either on its own or with an application image:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 --
    ↪ seccfg my_seccfg.lua
```

If only the security profile is specified, the application code is then provisioned separately with a further `mkehx-cp` run. The security configuration has its own Extended Hex file and can be reviewed and signed off independently of application releases, as a safety-certification process may require. Section 6.2.9 on page 63 describes how a profile is

written and verified, including with an AI assistant and **c2p-agent**.

Alternatively, the security configuration compiled from a **.lua** profile can be packaged together with application code:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 --
    ↪ seccfg my_seccfg.lua application.out
```

**Application Code Provisioning** Once the device is in the HS-SE state, application firmware can be provisioned. The `mkehx-cp` command accepts ELF (**.out**) and Intel hex (**.hex**) images:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ application.out
```

An application binary that carries no security configuration can be combined with one using `--seccfg` or `--seccfg-template`, as shown under *Separate Workflow* above. The two options are mutually exclusive. A binary that already carries a security configuration, or other data inside the seccfg flash sectors, is refused. In both workflows, the consequence review applies: inspect the configuration with `--explain-only`, and acknowledge irrecoverable or lockout findings with `--allow-irreversible` (see *Consequences of a Security Configuration* on page 56).

For dual-core applications, use the appropriate multi-core target configuration:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1+CPU3_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ application.out
```

The resulting Extended Hex file contains the signed application code and the certificates the HSM needs to authenticate it.

Every application and security-configuration certificate records a software revision, which is set with `--swrv` and defaults to 1. The flash-kernel and HSM-runtime certificates C2Prog adds to the file always carry revision 1. Key provisioning sets a minimum software revision for each image type (the `--sr-*` options under *Key Provisioning*), and the device rejects a certificate whose revision is below the minimum for its type. Raising a minimum retires the images signed below it. The application certificates are checked against the `--sr-sbl` minimum, the security-configuration certificates against `--sr-ssu`.

The seccfg flash addresses depend on the flash bank mode, which `--target` selects (see Section 3.4.2 on page 16). A binary whose security configuration was built for a different bank mode, or that carries only some of the per-CPU seccfg blocks, is refused.

**JTAG Code Provisioning** Code provisioning can also be performed via JTAG by selecting a target configuration with the `_JTAG` suffix instead of `_UART`:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_JTAG --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ application.out
```

JTAG programs faster than UART. The security configuration and the security state can restrict or disable JTAG access.

---

⚠ If a security configuration makes the device unreachable over JTAG, use the UART boot mode to recover it. On an HS-SE device in SSU Mode 3, an application image cannot be loaded over UART. However, if the configuration in force names the HSM as the seccfg update owner, a file that holds only an SSU Mode 1 security configuration can be loaded over UART. This returns the device to SSU Mode 1 (page 59).

---

**Encrypted Application Images** The `--encrypt` option of the `c2p-cli mkehx-cp` command encrypts the CPU1 and CPU3 application images, so the Extended Hex file carries only their ciphertext. During code provisioning, the device's HSM decrypts each image and programs the plaintext into flash.

The encryption key is taken from the keyset that also provides the signing key. By default, the Secondary Manufacturer Encryption Key (SMEK) is used. With `--keyrev backup`, the Backup Manufacturer Encryption Key (BMEK) is used. The key must match the encryption key installed during key provisioning.

The encryption key can only be taken from a keyset. A PEM file or a PKCS#11 token holds a signing key only. Therefore, `--encrypt` requires the `--keyset` option.

To encrypt both images of a dual-core application:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1+CPU3_JTAG --keyset MyKeyset.
    ↪ session --keyset-password secret --encrypt application.out
```

Each image is encrypted with AES-256 in CBC mode under a fresh random initialization vector, padded with 0xFF to a whole number of 16-byte blocks. Its certificate records the initialization vector, the SHA-512 hash of the ciphertext and the SHA-512 hash of the plaintext. The HSM checks the ciphertext before decrypting and the plaintext after. The key is not written to the Extended Hex file. A security configuration in the same file is signed but not encrypted.

The HSM erases the CPU's application flash when it accepts the certificate, and it can check the key only once the image is programmed. An image encrypted with a key other than the one key provisioning installed fails at verification (**Verification failed**) and leaves no valid application for that CPU. When the CPU is CPU1, the device also refuses JTAG connections, because debug access is granted by the CPU1 application certificate. Program a correctly encrypted file over UART to recover. In SSU Mode 3, the HSM refuses an application image loaded over UART. If the configuration in force names the HSM as the seccfg update owner, first load a file that holds only an SSU Mode 1 security configuration over UART, then the correctly encrypted file (page 59). With any other owner, UART cannot recover the device.

### 6.2.7 Reviewing a Security Configuration

C2Prog reports the consequences of a security configuration before it writes the Extended Hex file. Both `mkehx` and `mkehx-cp` produce the report, as text for a reviewer and as JSON for build pipelines and AI assistants.

**Consequences of a Security Configuration** Some security configurations limit the device permanently. `mkehx-cp` checks every configuration before it writes the Extended Hex file and ranks each finding by severity: irrecoverable, lockout, caution or info.

When a finding is at the irrecoverable or lockout level, `mkehx-cp` stops. It writes no Extended Hex file and exits with status 3 (Section 4.1). To write the file anyway, pass `--allow-irreversible`. Findings at the caution level do not stop the command.

`mkehx` applies the same check to a configuration for an HS-FS device (Section 6.2.2).

To see the findings before signing, use `--explain-only`. It prints a report and exits without signing and without writing a file. No signing key is required. Therefore, a profile or an application build can be reviewed on a machine without key material:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_JTAG --explain-only --seccfg
    ↪ my_seccfg.lua
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_JTAG --explain-only
    ↪ my_application.out
```

The report covers the CPU1 image, which holds the device-wide settings: debug access, write/erase protection, update owners and the RAM-open policy. It lists what the configuration does, every field that cannot be changed once it is programmed, and the warnings, most severe first.

The report also lists the access-protection (APR) table of each CPU whose configuration the file carries. This table is the memory protection that the core applies to its LINKs at run time. Application code without a security configuration produces no report.

Update-owner and write/erase-protection settings are binding only in SSUMODE3.

---

⚠ A configuration flagged irrecoverable or lockout may permanently remove the device's ability to be re-provisioned or debugged. Review the report with `--explain-only` before passing `--allow-irreversible`.

---

**The JSON Report** For build pipelines and AI assistants, `--explain-only-json` writes the same report as one JSON document on standard output. Progress and diagnostic messages go to standard error.

Both explain options exit with the status that the signing run would return: 3 when it would need `--allow-irreversible`, 1 when it would refuse the configuration, and 0 otherwise. A pipeline can therefore gate on the status alone and keep the document as a record of what was approved:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_JTAG --explain-only-json app.
    ↪ out | jq .summary
```

The document is a versioned audit record. It has these fields:

- `report`: the document type, `c2p-seccfg-explain`.
- `report_version`: the version of the field set.
- `seccfg_present`: false for application code without a security configuration. No further fields follow in that case.
- `summary`: one line with the SSU mode and the number of findings.
- `valid`: true when the configuration passes validation.
- `validation_errors`: the individual validation findings.
- `does`: what the configuration applies.
- `irreversible`: the fields that cannot be changed once programmed.
- `warnings`: the findings, most severe first.
- `cores`: the APR table of each reviewed CPU, as structured data (described below).
- `reflash_paths`: one object per transport (described below).
- `way_back`: the way-back line of the text report.
- `images`: the SHA-256 of each seccfg image reviewed, keyed by CPU.
- `tool`: the version of the tool that produced the report.
- `target`: the selected target.

- `acknowledgment_required`: true when writing the file would need `--allow-irreversible`.
- `generation_refused`: true when a signing run would refuse the profile for leaving no reflash path.

Each entry of `does`, `irreversible` and `warnings` holds a `title` and a `detail`. A warning also has a `severity` of `IRRECOVERABLE`, `LOCKOUT`, `CAUTION` or `INFO`. The `images`, `tool` and `target` fields tie an archived report to the exact bytes it approved.

Each object in `reflash_paths` has these fields:

- `transport`: `UART` or `JTAG`.
- `verdict`: `OK`, `BLOCKED`, `NOT_ENFORCED` for an SSUMODE1 image, or `UNKNOWN` when the kernel's needs were not available.
- `reason`: the explanation of the verdict.
- `notes`: additional remarks, such as the SSUMODE3 UART note.

`cores` is keyed by CPU (`CPU1`, `CPU2`, `CPU3`). CPU1 is always reported. CPU2 or CPU3 is reported when the file carries its configuration and that configuration has at least one active APR. Each entry has these fields:

- `apr_active`: the number of rows in use.
- `apr_locked_rows`: the number of rows locked until the next reset, unused fill rows included.
- `apr_committed_unused`: the number of unused fill rows committed for the power-on cycle.
- `aprs`: one object per active row.

Each object in `aprs` has these fields:

- `index`: the row's position among the core's 64 logical APRs.
- `start`: the first byte the row covers, as a number.
- `end`: the last byte the row covers, as a number.
- `owner_link`: the id of the LINK that owns the row.
- `code_region`: true when the owning LINK may execute from the row.
- `locked`: true when the row is locked until the next reset.
- `committed`: true when the row is committed for the power-on cycle.
- `read`: the ids of the LINKs the row grants read access to, as an array of numbers.
- `write`: the ids of the LINKs the row grants write access to, as an array of numbers.
- `api_link`: the id of the row's API LINK. Present only on a row that sets one.

An empty `read` or `write` array means the row grants no access beyond what its owner holds.

The field names are fixed for `report_version` 1. A field keeps its meaning, and new fields are only added alongside the existing ones. Therefore, a freedom-from-interference check can read the protection map directly from the report.

The following command asserts that no region owned by LINK2 grants write access to LINK5. Replace the LINK ids with those of your own design. `jq -e` exits with a non-zero status when the assertion is false, which fails the build step:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_JTAG --explain-only-json --
    ↪ seccfg my_seccfg.lua | jq -e '[.cores.CPU1.aprs[] | select(.
    ↪ owner_link == 2 and (.write | index(5)))] | length == 0'
```

### 6.2.8 Keeping a Device Reachable

A security configuration controls the transports and debug gates through which C2Prog reaches the device. In SSU Mode 2 and 3 these settings decide whether a later session can program it at all.

**Keeping UART and JTAG Available** To replace a security configuration, C2Prog loads its flash kernel onto the device over UART or JTAG and runs it there. If a configuration prevents both, it can never be replaced.

The following profile settings keep both transports available. Most of them are defaults and only need to be left unchanged. In SSU Mode 2 and 3, the `flasher` block must be added.

For UART:

- Leave `boot.default_bootmode` at `true`, the default. The boot ROM then follows the boot pins and can enter its UART loader.
- A custom boot configuration (`boot.default_bootmode = false`) must list a UART boot mode in `boot.bootdef_codes`.

For JTAG:

- Leave emulation boot enabled (`boot.emu_boot_en = true`). If the profile does not set it, C2Prog sets it (see `recovery_safe` below).
- In SSU Mode 3, do not set `debug.c29.mode = "disable"`. Keep the default `"password"` mode. A profile that sets its own `debug.c29.password` needs that password in every later JTAG ehx file. Keep at least one zone gate on the default password (see *C29 Debug Password* on page 62).

For both, in SSU Mode 2 and 3:

- Add `flasher = { link = "LINK1" }`. The kernel runs as LINK1, and this block grants LINK1 the kernel's three RAM regions and the peripherals the kernel uses (listed below).
- Also grant LINK1 all other peripherals with `periphs_rw = { "*" }` on the LINK1 system link. This configuration has been verified on silicon. It also covers kernel accesses that are not in the list below.
- Leave `update_owners` unset. The default owner then matches the command that builds the file. For `mkehx-cp`, the default is the HSM (`cpu = "CPU-HSM", link = "LINK2", enable = true`), which writes the configuration on an HS-SE device. For `mkehx`, the default is CPU1 (`cpu = "CPU1", link = "LINK1", enable = true`). On an HS-FS device, the flash kernel writes the configuration from CPU1.
- Leave `ramopen_lock` at `false`, the default.

The kernel uses the following peripherals. A denied access to any of them stops the kernel during start-up, on UART and on JTAG alike:

- System control and watchdog: `CPU_SYS_REGS`, `PER_CFG_REGS_WD_REGS`.
- CPU timers: `CPUTIMER0_REGS`, `CPUTIMER1_REGS`, `CPUTIMER2_REGS`.
- GPIO: `GPIO_DATA_REGS`, `GPIO_DATA_READ_REGS`.
- UART: `UARTA_REGS`.
- Error aggregator: `ERR_AGG_REGS`.
- Error signaling modules: `ESM_CPU1_REGS`, `ESM_CPU2_REGS`, `ESM_CPU3_REGS`, `ESM_SYS_REGS`, `ESM_SAFETY_AGG_REGS`.
- Clock checkers: `DCC1_REGS`, `DCC2_REGS`, `DCC3_REGS`.
- HSM communication: the HSM mailbox and the CPU1 HSM IPC channels.

The LINK1 wildcard must be written with braces, as a list. In the following profile fragment, LINK2 (the application's link) drives two peripherals, and LINK1 receives the wildcard:

```
system_links = {
   { name = "LINK2", is_link2 = true, periphs_rw = { "EPWM1_REGS", "
      ↪ ESM_SYS_REGS" } },
   { name = "LINK1", is_link1 = true, periphs_rw = { "*" } },
},
```

With the braces, LINK1 may use every peripheral and LINK2 may use its two. `EPWM1_REGS` and `ESM_SYS_REGS` are shared by both links, and the flash kernel keeps access to everything it needs.

Without the braces (`periphs_rw = "*"`), the wildcard does not give LINK1 any peripheral that another link names. `EPWM1_REGS` and `ESM_SYS_REGS` then belong to LINK2 alone. The kernel needs `ESM_SYS_REGS`, so C2Prog refuses to generate the profile and names the missing peripheral:

```
CPU1 flasher: the kernel requires ESM_SYS_REGS on LINK1, but the resolved
   ↪ configuration does not grant it (a wildcard yield, an except entry, or
   ↪  isolated_peripherals removed it)
```

The report lists each peripheral the wildcard left out under the caution **peripheral wildcard yields**.

Before programming a profile, run `--explain-only` and check the **REFLASH PATHS** lines. Each transport gets a verdict of **OK** or **BLOCKED**. A blocked line says what stops the kernel. The closing way-back line says whether the configuration can still be replaced. In this example for an HS-FS device, JTAG is closed and UART still works:

```
REFLASH PATHS (can this device still be reprogrammed?):
  [OK] UART -- A UART-capable boot mode is reachable and LINK1 holds
                 every region and peripheral the kernel needs.
  [BLOCKED] JTAG -- C29 device debug is disabled (DEBUG_CFG C29DBGEN = 0x4)
     ↪ ,
                 which is enforced in SSU Mode 3, so no debug probe can
                 attach.
  Way back: The seccfg update owner is LINK1 (enabled) and a surviving
           reflash path runs a kernel as that link, so the inactive slot
           can still be written.
```

Next, test the configuration on an HS-FS device. Program it with `mkehx` in the SSU mode of the production device (Section 6.2.2), and run a second programming session over each transport you need.

On an HS-SE device in SSU Mode 3, the HSM refuses application images that are loaded over UART. However, if the configuration in force names the HSM (CPU-HSM / LINK2) as the seccfg update owner, a file that holds only a security configuration can still be loaded over UART. Such a file can return the device to SSU Mode 1. After that, the application can be loaded over UART again. If a profile names no owner, `mkehx-cp` makes the HSM the seccfg update owner.

If the seccfg update owner is CPU1 (CPU1 / LINK1), a security configuration that is loaded over UART does not take effect. In this case, JTAG is the only way to reprogram the device. Therefore, keep JTAG available on such a device. An HS-FS device does not have this limit.

In the following example, **profile.lua** is the security profile of the device, and `--ssu-mode 1` builds it for SSU Mode 1. The resulting file holds no application. It is loaded with the device in UART boot mode, as follows:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --keyset=MyKeyset.
   ↪ session --keyset-password=secret --seccfg=profile.lua --ssu-mode 1
   ↪ recover_mode1.ehx
c2p-cli load --port=COM7 recover_mode1.ehx
```

In addition, C2Prog refuses or flags the following configurations:

- A configuration with neither a UART boot mode nor emulation boot is refused with status 1. `--allow-irreversible` does not override this.

- A profile that blocks every path in SSU Mode 2 or 3 is refused when it is generated, with status 1. To program such a lockout on purpose, set `allow_no_reflash = true` in the profile.
- A configuration that blocks every path gets a lockout warning and needs `--allow-irreversible`. The same applies to a configuration for `mkehx-cp` in SSU Mode 3 that leaves UART as the only path and does not name the HSM as the seccfg update owner.

Setting `recovery_safe = false` disables both the automatic `emu_boot_en` setting and the refusal at generation. This setting is intended for profiles that must reproduce a fixed byte image.

The verdicts are based on the requirements of the C2Prog flash kernel. The UART verdict assumes a key-provisioned device, where the configuration must grant the kernel its RAM. The `flasher` block provides this grant. The JTAG verdict assumes that the kernel runs as LINK1 while RAMOPEN is open. RAMOPEN is a boot state in which LINK1 may use the RAM into which the boot ROM loads a kernel. Setting `ramopen_lock = true` closes it. RAMOPEN covers no peripherals. Therefore, the peripheral grants above are needed for JTAG as well.

For a profile that reserves flash for its own bootloader (`custom_bootloader`), the report notes that the requirements of this bootloader are not checked.

**C29 Debug Password** In SSU Mode 3, four debug gates control JTAG access to the CPUs: the C29 gate (`debug.c29`) and one gate per zone (`debug.zone1` to `debug.zone3`). A gate in `"password"` mode opens when a debugger presents that gate's password over the secure access port (SEC-AP) of the device. A CPU debug port opens only while the C29 gate and at least one zone gate are open.

C2Prog presents the passwords each time it connects over JTAG, before it selects CPU1. For the C29 gate, it presents the password stored in the ehx file. If the ehx file contains none, it presents the SDK default password, 16 bytes of `0x55`. For the zone gates, it always presents the default password. A profile without its own passwords contains the default password, as does every configuration in SSU Mode 1 and 2.

For JTAG programming in SSU Mode 3, keep at least one zone gate in `"password"` mode with the default password. When no zone gate opens for the default, the consequence report shows the JTAG path as **BLOCKED**.

The password is stored in the ehx file when the file is created, like the CSM keys of an F28x device:

- Command line: the `--debug-password` option of `mkehx`, `mkehx-cp` and `mkehx-kp`.

- GUI: the **C29 debug password (SSU Mode 3)** field in the **Configure…** dialog of a JTAG target.

The password is specified as 32 hex digits, in the same format as the profile's `debug.c29.password`. Only JTAG targets accept a password. An empty GUI field selects the default password. For example:

```
c2p-cli mkehx --target=29H85xTU9,TU8-CPU1_JTAG --debug-password
    ↪ =00112233445566778899AABBCCDDEEFF application.out
```

Specify the password of the security configuration that is active on the device when the ehx file is loaded. Therefore, an ehx file that programs a profile with a new password must specify the old password. Subsequent ehx files specify the new one.

If the device refuses the password in the ehx file, C2Prog presents the default password next. An ehx file that specifies a password therefore also loads onto a device whose active configuration uses the default. If the device refuses both, the load fails. The error message states which passwords the device refused, whether a zone gate opened, and the `DEBUG_ENABLE_STATUS` value the device reported.

It is recommended that an ehx file containing a debug password be protected with `--passphrase` when it leaves the production environment.

### 6.2.9 Authoring a Custom Security Configuration

A security configuration is described by a high-level **.lua** profile. The profile lists the application's protection links, the memory and peripherals each one owns, and the device-level debug, boot, update-owner and RAM-open policy. C2Prog compiles it into the per-CPU seccfg images and signs them into an Extended Hex file with `mkehx-cp --seccfg` (see the separate workflow in Section 6.2.6). Verify and explain a profile before signing it.

---

⚠ On an HS-KP device, programming a security configuration takes the device to HS-SE in every SSU mode, and this transition cannot be undone. In SSUMODE3, the configured seccfg update owner is the only path for changing the configuration afterwards. A production profile can therefore lock the device permanently. With `update_owners.seccfg.enable = false`, the device can never be re-provisioned. Combined with a disabled debug gate, JTAG access is also removed permanently. In SSUMODE1 and SSUMODE2, the seccfg sectors remain reprogrammable. Committed or disabled fields are therefore recoverable, unless the device is later moved to SSUMODE3 with these fields still set. Verify a profile with `explain_seccfg` before signing it. Its warnings are ranked by severity and take the SSU mode of the profile into account.

---

A production configuration is specific to the application. It names the protection links, the memory and peripherals each link owns, and the debug and update-owner policy for that design. C2Prog ships no ready-made production configurations. Its templates, such as **default_seccfg_ssumode1**, are development baselines that disable access protection. The authoring tools are part of the agentic interface (Section 7.2 on page 70). The **c2p-agent** seccfg tools require no signing key, no files on disk, and no device. Therefore, a profile can be drafted and revised before any key material is involved. The workflow is as follows:

- **Learn the grammar.** The `seccfg_schema` tool returns the authoritative field grammar with a minimal skeleton. The engine accepts only those fields and reports a misspelled or unknown field as an error.
- **Write the profile for the application.** Describe the protection links, the memory and peripherals each one owns, the sandbox zones, and the device-level debug, boot and update-owner policy. The **default_seccfg_ssumode1** template is a useful reference for the development (SSUMODE1) case.
- **Generate and verify.** The `generate_seccfg` tool compiles a profile, then validates and decodes the resulting images in the same call. It reports register-level findings, which are used to revise the profile. The `verify_seccfg` tool does the same for a standalone image and can compare it against a known-good reference.
- **Explain before flashing.** The `explain_seccfg` tool reports what the configuration does, which bits are permanent at the profile's SSU mode, and ranked irrecoverable or lockout warnings. Run it on the final profile as a last check.
- **Sign and program.** Package the profile with `mkehx-cp --seccfg` as shown in Section 6.2.6, then program the Extended Hex file over UART or JTAG.

---

To build a profile interactively, connect an assistant to **c2p-agent** (Section 7.2 on page 70) and describe the application in plain language: the CPUs in use, the code and data each protection domain needs, the peripherals each one drives, and how strict the debug and update-owner policy should be. A first request might be: *"Using the c2prog seccfg tools, draft an SSUMODE3 profile for CPU1 with a control link that owns the ADC and ePWM registers and a separate communications link, then generate and explain it."* The assistant reads the grammar with `seccfg_schema`, drafts a **.lua** profile, and calls `generate_seccfg` to compile and validate it. You then review the register-level findings together and refine the description until it validates cleanly. Finally, have it run `explain_seccfg` and walk you through the irreversible fields before you sign the profile with `mkehx-cp`.

---

Keep a UART recovery boot mode available. Should a configuration leave the device unreachable over JTAG, UART boot is the recovery path (see the warning under *JTAG Code*

*Provisioning*). On an HS-SE device in SSU Mode 3, a UART load can return the device to SSU Mode 1 only if the HSM is the seccfg update owner. With any other owner, JTAG must stay available.

**Comparing Against a Reference Image** A profile sometimes has to reproduce a configuration that already exists, for example one generated with TI SysConfig. The `verify_seccfg` tool of the agentic interface (Section 7.2 on page 70) compares generated seccfg images against such reference images byte for byte. The images under test are passed as `cpu1`, `cpu2` and `cpu3`, and the reference images as `ref_cpu1`, `ref_cpu2` and `ref_cpu3`. Each image must be one complete 2048-byte seccfg block, base64-encoded. A block of any other length is refused. The differing offsets show where the generated configuration deviates from the reference. Correct the profile and regenerate it until the images match.

---

The comparison is intended to run inside an assistant conversation. The assistant reads the reference file, encodes it, and passes it as a tool argument, so no base64 is prepared by hand. The reference is a file holding the raw block, for example extracted from a SysConfig-built **.out** with `c29objcopy -O binary --only-section=.TI.bound:CPU1_Cfg`. A request like *"Generate the seccfg from my_seccfg.lua and compare it against ref/cpu1.bin; list the differing bytes and the profile fields they trace to"* makes the assistant generate the images, call `verify_seccfg` with the reference, and work through the reported offsets.

---

The comparison is reported as `reference_diff`, an object keyed by CPU with one entry per reference supplied:

- `identical`: true when the image and its reference agree byte for byte.
- `byte_diff_count`: the number of bytes that differ.
- `diffs`: one entry per differing byte, at most 128. Each entry has these fields:
  - `offset`: the position in the image, as a hexadecimal string (`0x0e4`).
  - `got`: the byte value in the image under test.
  - `expected`: the byte value in the reference.
- `truncated`: true when more bytes differ than `diffs` lists.

The field names are stable across releases: a name keeps its meaning, and new fields are only added beside the existing ones.

To confirm that a regenerated configuration still matches the approved one, compare the SHA-256 values under `images` in the consequence report with those of the report archived at approval (Section 6.2.7 on page 57). No byte comparison is needed.

### 6.2.10 Isolating Parts of an Application

A security configuration can also isolate parts of an application from each other. Each protection link owns code, data and peripherals of its own, and the hardware blocks accesses across the boundary, with no operating system involved. The hardware checks addresses. Therefore, the part of the application that belongs to a link is defined by the object files that the linker places into its regions. The object files can hold hand-written C, code generated from a PLECS or Simulink model, or a vendor library. No changes to the compiler or code generator are required, and the build can remain an ordinary Code Composer Studio or makefile project.

The example below gives generated control code and hand-written communication code separate links. Any other grouping of the object files works the same way. In the profile, each entry in `modules` is one protection link, with its memory in kB and the peripheral register blocks it may reach (the full grammar is in Section 6.2.9):

```
modules = {
    { name = "Control", code_flash_kb = 32, rw_data_kb = 8,
      periphs_rw = { "ADCA_REGS", "ADCA_RESULT_REGS" },
      fixed_addresses = { codeAPR_Flash = 0x10010000 } },
    { name = "Comms", code_flash_kb = 16, rw_data_kb = 4,
      periphs_rw = { "GPIO_DATA_REGS" },
      fixed_addresses = { codeAPR_Flash = 0x10020000 } },
},
```

A link can access only the peripherals listed in its `periphs_rw`. For example, **Comms** lists no ePWM registers. Therefore, a fault in the communication code cannot write the PWM registers. The `seccfg_peripherals` tool of the agentic interface lists the valid register-block names. A name outside that list stops generation.

The `fixed_addresses` entry pins a region to a chosen start address. Without it, C2Prog selects the addresses, and they must be copied from the consequence report (shown below) into the linker command file. With it, the profile and the linker command file specify the same addresses, `0x10010000` and `0x10020000` in this example. The **SECTIONS** directive of the linker command file places the object files of each link into its region:

```
MEMORY
{
    CONTROL_CODE : origin = 0x10010000, length = 0x8000
    COMMS_CODE : origin = 0x10020000, length = 0x4000
}
SECTIONS
{
    .control_text : { model.obj(.text) ctl_task.obj(.text) } > CONTROL_CODE
    .comms_text : { comms.obj(.text) canif.obj(.text) } > COMMS_CODE
}
```

To add a source file to either link, list its object file in the **SECTIONS** directive. The security configuration only needs to change when a region has to grow. The placement can be confirmed with the consequence report, which lists every region with its address range and owner:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART --explain-only --seccfg
    ↪ my_seccfg.lua
```

```
- CPU1 APR12: 0x10010000-0x10017fff (32 kB), owner LINK4, code region;
    ↪ read: none; write: none.
- CPU1 APR13: 0x10020000-0x10023fff (16 kB), owner LINK5, code region;
    ↪ read: none; write: none.
```

The ranges must match the **MEMORY** directive. A region that grants no read or write to another link is private to its owner. A build script reads the same table from `--explain-only-json` under `cores`, where each row carries `start`, `end`, `owner_link`, `read` and `write` (Section 6.2.7 on page 57).
Partitioning is optional. With a single protection link, the device-level debug, boot, update-owner and write/erase policy still applies to the whole application.

**The Multi-Core Variant** If the design already uses several cores, the partitioning can follow the cores, for example with control on CPU1 and connectivity on CPU3. Each core is built and linked on its own. Therefore, no per-object placement and no pinned addresses are needed. The profile describes each core in its `cores` table. The isolation between the cores is provided by the security configuration alone:

```
cores = {
    CPU1 = { flash_load_size_kb = 4, system_links = { --[[ control ]] } },
    CPU3 = { flash_load_size_kb = 4, system_links = { --[[ comms ]] } },
},
```

An application on CPU2 is declared the same way. CPU2 owns no flash of its own, so its code and constant data are placed in RAM, and its flash footprint is reserved on CPU1 with `secondary_flash_kb`. Each core has its own security configuration image with its own access-protection table, and the consequence report covers every core the payload carries.

# 7 Automation and Agentic Interfaces

C2Prog can be driven by other software: over JSON-RPC for CI/CD pipelines, through the Model Context Protocol for AI agents, and via a DLL for integration into custom applications.

## 7.1 JSON-RPC Interface

The C2Prog functionality can be accessed via `JSON-RPC`, a lightweight remote procedure call protocol.

The executable for launching the JSON-RPC server is **c2p-server**.

```
c2p-server --rpc-port=<port>
```

For example

```
c2p-server --rpc-port=8080
```

By default the server binds the loopback interface (`127.0.0.1`) and accepts connections only from the local machine. To reach it from another host, such as a CI/CD runner, name the address of the interface it should listen on:

```
c2p-server --rpc-port=8080 --bind-address=192.168.1.20
```

A JSON-RPC client can then connect to the C2Prog server and issue remote procedure requests.

---

⚠ The server has no authentication: whoever reaches the port can program the connected target. Bind a single address on a trusted network. Avoid using the wildcard `0.0.0.0` on a machine with a route to an untrusted network, and keep the port behind a firewall.

---

### 7.1.1 Load command

The `load` method initiates a programming session.

```
{
  "method": "load",
  "params": {
    "file":"<ehx file path>",
    "port":"<port>",
    "passphrase":"<passphrase>"
  },
  "jsonrpc": "2.0",
  "id": 0
}
```

For example:

```
{
  "method": "load",
  "params": {
    "file":"c:\test.ehx",
    "port":"COM5",
    "passphrase":"very secret"
  },
  "jsonrpc": "2.0",
  "id": 0
}
```

The server response to the `load` command is formatted as follows:

```
{
  "result": {
    "info":"",
    "error":"<error description>",
    "progress":0.0
  },
  "jsonrpc": "2.0",
  "id": 0
}
```

A non-empty `error` string signals that the `load` command failed.

### 7.1.2 Get Status command

The `get_status` method can be used to query the status of an ongoing flashing session.

```
{
  "method": "get_status",
  "jsonrpc": "2.0",
  "id": 0
}
```

The response to `get_status` is identical to the `load` method response:

```
{
  "result": {
    "info":"<status description>",
    "error":"<error description>",
    "progress":<[0.0-1.0]>
  },
  "jsonrpc": "2.0",
  "id": 0
}
```

The programming session is complete once `progress` reaches `1.0`. If an error occurs, it is indicated by means of the `error` field.

### 7.1.3 Shutdown command

The server can be shut down using the `shutdown` method.

```
{
  "method": "shutdown",
  "jsonrpc": "2.0",
  "id": 0
}
```

### 7.1.4 Version command

The `version` method returns the C2Prog version in the `info` field of the response.

```
{
  "method": "version",
  "jsonrpc": "2.0",
  "id": 0
}
```

### 7.1.5 Request errors

A request that is not valid JSON is answered with the JSON-RPC error `-32700`, a `load` without `file` or `port` with `-32600`, and an unknown method with `-32601`. A failure during programming is not a JSON-RPC error: it is reported in the `error` field of the response.

## 7.2 Agentic Interface (MCP)

C2Prog can be exposed to AI agents through the `Model Context Protocol (MCP)`, an open standard supported by a range of agent hosts. The server is host-agnostic: any MCP-compatible client, such as `ClaudeCode` and other MCP-enabled tools, can launch it and call C2Prog functionality as tools.

The executable is **c2p-agent**. It provides two subcommands.

### 7.2.1 Running the server

The `mcp` subcommand runs the MCP server over *stdio*, exchanging newline-delimited JSON-RPC messages with the agent host that launches it.

```
c2p-agent mcp
```

The agent host spawns this process; its standard output carries only protocol messages, while diagnostics are written to standard error. The tools default to read-only operation; the exceptions are storing a configuration setting such as the TI EmuPack path, and the MCU diagnostics tools that halt, reset or write an attached target. A tool that can change

state says so in its description and is marked as a writing tool over MCP.

The server exposes tools for target and device inspection, serial-port and JTAG-probe detection, EmuPack configuration, JTAG-adapter maintenance guidance, live MCU diagnostics through the GDB tooling (Section 8), and retrieval from a searchable copy of this manual (covering supported devices, the security and provisioning model, Extended Hex files, error codes, and the command-line, JSON-RPC and DLL interfaces). No network access or additional installation is required: the documentation is packaged with C2Prog and searched locally.

These tools are meant for an AI agent. A connected host discovers them and their parameters over MCP, so they are not listed individually here; refer to the tool descriptions presented by your agent host.

### 7.2.2 Configuring the agent host

Any MCP-compatible host can use the server; it is registered by pointing the host's MCP configuration at the launch command `c2p-agent mcp`. The `register` subcommand emits or applies this registration. In every case the launch command is this executable's own resolved path, so the binary that is registered is exactly the one that ran `register`. Registration is idempotent: an entry that already names this executable is left unchanged.

Run `c2p-agent register` with no option to register every supported host that is installed on the computer. A host is considered installed when its configuration directory exists. For each installed host, the command writes the server entry into the host's user-scope configuration. Therefore, the server is available in every project, and no host CLI is required. For Claude Code, Codex and OpenCode, the command also installs a skill describing the tools. Before a configuration file is changed, it is copied to a file with the additional extension **.c2prog.bak**.

```
c2p-agent register
```

The `--host` option selects a host by the name given in the list below, and it can be repeated. A host selected with `--host` is registered even if its configuration directory does not exist yet. The options `--claude`, `--codex` and `--opencode` are short forms of `--host claude`, `--host codex` and `--host opencode`. The `--dir` option overrides the configuration directory of the selected host and requires exactly one host.

```
c2p-agent register --host cursor --host vscode
```

The `--status` option lists every host with its configuration file, whether it is installed, and whether its entry launches this **c2p-agent**, another C2Prog installation, or nothing. It writes nothing. The `--status-json` option prints the same report as a JSON document.

In the graphical interface, **Help > Set up AI agent integration…** shows this status and registers the selected hosts. On Windows, the installer offers a **Set up AI agent integration**

option that runs `c2p-agent register`.

The supported hosts are as follows:

- **Claude Code** (`claude`): the entry is written to **~/.claude.json**, the user-scope location Claude Code reads for every project. **Code Composer Studio** uses Claude Code as its built-in AI assistant, so the same registration applies. To use it in CCS, enable the assistant via **Help > Get Started with AI**, select Claude Code, and open **View > Claude Code Chat**.
- **Codex** (`codex`): the entry is written as a `[mcp_servers.c2prog]` table in **~/.codex/config.toml** (or under `$CODEX_HOME`), and the skill is installed under **~/.codex/skills**. Existing comments and other servers in the file are preserved.
- **OpenCode** (`opencode`): the configuration directory is **~/.config/opencode** (or `$XDG_CONFIG_HOME/opencode`). The entry is written under the `mcp` key in **opencode.json**, and the skill is installed under **skills** in the same directory.
- **Claude Desktop** (`claude-desktop`): the entry is written under the `mcpServers` key in **claude_desktop_config.json**. The file is located in **%APPDATA%\Claude** on Windows, in **~/Library/Application Support/Claude** on macOS, and in **~/.config/Claude** (or `$XDG_CONFIG_HOME/Claude`) on Linux. For Claude Desktop installed from the Microsoft Store, the file is located in the app's package folder under **%LOCALAPPDATA%\Packages**.
- **Cursor** (`cursor`): the entry is written to **~/.cursor/mcp.json** under the `mcpServers` key.
- **VS Code** (`vscode`), in GitHub Copilot agent mode: the entry is written under the `servers` key in the **mcp.json** file of the default user profile. The file is located in **%APPDATA%\Code\User** on Windows, in **~/Library/Application Support/Code/User** on macOS, and in **~/.config/Code/User** on Linux. A file that contains comments is left unchanged, and the entry to add by hand is printed instead. For a single workspace, `c2p-agent register --vscode` prints the entry for a **.vscode/mcp.json** file in the workspace folder, plus an equivalent `code --add-mcp` command.
- **Gemini CLI** (`gemini`): the entry is written under the `mcpServers` key in **settings.json** in the **~/.gemini** directory. If the environment variable `GEMINI_CLI_HOME` is set, the **.gemini** directory is located in the directory it names instead.

Other settings and servers in these files are preserved. For any other host, `c2p-agent register --print` outputs a `mcpServers` entry and the path of the skill file to install.

A host that is already running continues to use the server it started. Restart the host, or reconnect its MCP servers, to load a newly registered server. The output of `c2p-agent`

`register` names the step for each host. Similarly, C2Prog can be updated while a host is running **c2p-agent**. In this case, the host continues to use the previous version until it is restarted.

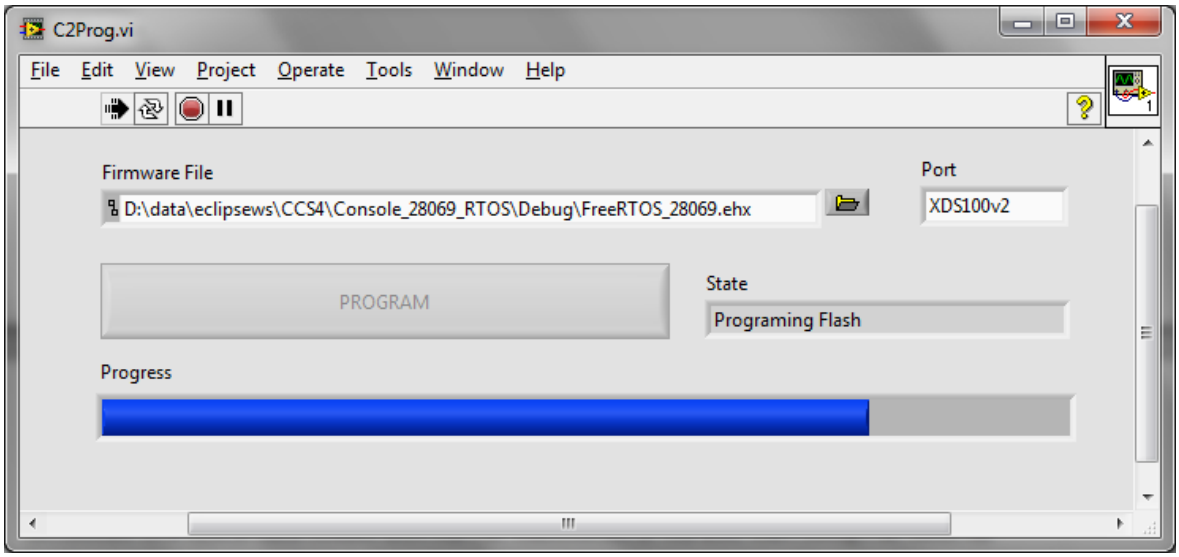
---

The agentic interface is a minimal first release and is subject to change without notice. Firmware programming and erase are not yet exposed as tools.

---

## 7.3 DLL Interface

The C2Prog programming functionality is exported via a Dynamic-Link Library (DLL) for use by other applications, such as end-of-line (EOL) test stands or field support tools[2]. With the "c2p" C2Prog DLL, flash programming can be added to custom applications built with virtually any toolchain, including NI LabVIEW.




Below is a list of the function calls that are exposed by the C2Prog DLL. Please refer to Appendix C on page 96 for a more detailed description of the API.

- **c2pInitializeLibrary**: initializes the C2Prog environment
- **c2pProgram**: starts programming session (programs flash)
- **c2pGetStatus**: retrieves information about the progress of the flash programming
- **c2pGetErrorDescription**: retrieves a textual description (string) for a given error code
- **c2pCloseProgrammingSession**: terminates a programming session

[2]The DLL interface is only available with the professional license of C2Prog.

The function **c2pProgram** can be called as “blocking” or “non-blocking”. In blocking mode, the function will not return until the programming has completed (successfully, or with an error). In non-blocking mode, the function returns immediately, and the progress of the programming (and its success) must be polled using the **c2pGetStatus** call. This allows the application using the C2Prog DLL to display progress information while the programming takes place.

The C2Prog DLL is implemented as a wrapper around the **c2p-cli** executable. Therefore, the DLL must be able to locate **c2p-cli**. On Windows, the DLL will read the entry in the Windows registry that the C2Prog installer creates.

An alternate mechanism for specifying the location of the **c2p-cli** executable is the use of a **c2p.config** configuration file, located in the same directory as the DLL itself.

The **c2p.config** file must contain a single line with a quoted string. Several options exist for how to provide the path:

1. Specification of an absolute path
2. Specification of a relative path
3. Specification of a registry key, which contains the absolute path (Windows only)

The example below illustrates the first option, specifying an absolute path. Note how the path has to be provided as a quoted string.

```
"/home/user/opt/c2prog_v2.x"
```

Alternatively, a relative path can be specified by using the ”$” prefix. The path is then interpreted to be relative to the ”c2p.config” file (and, hence, the C2Prog DLL). This is shown in the next example.

```
"$/C2Prog"
```

The third option specifies a Windows HKCU registry key that contains the absolute path to the **c2p-cli** binary. The special character ”@” is used to identify this option, as shown below. The DLL will then read the value named ”InstallPath” at the provided registry path.

```
"@\SOFTWARE\MyApp\C2Prog"
```

The DLL binaries and header file are located in the `{C2Prog}\api` folder. Examples of how to use the API from different programming languages can be found in `{C2Prog}\api\examples`.

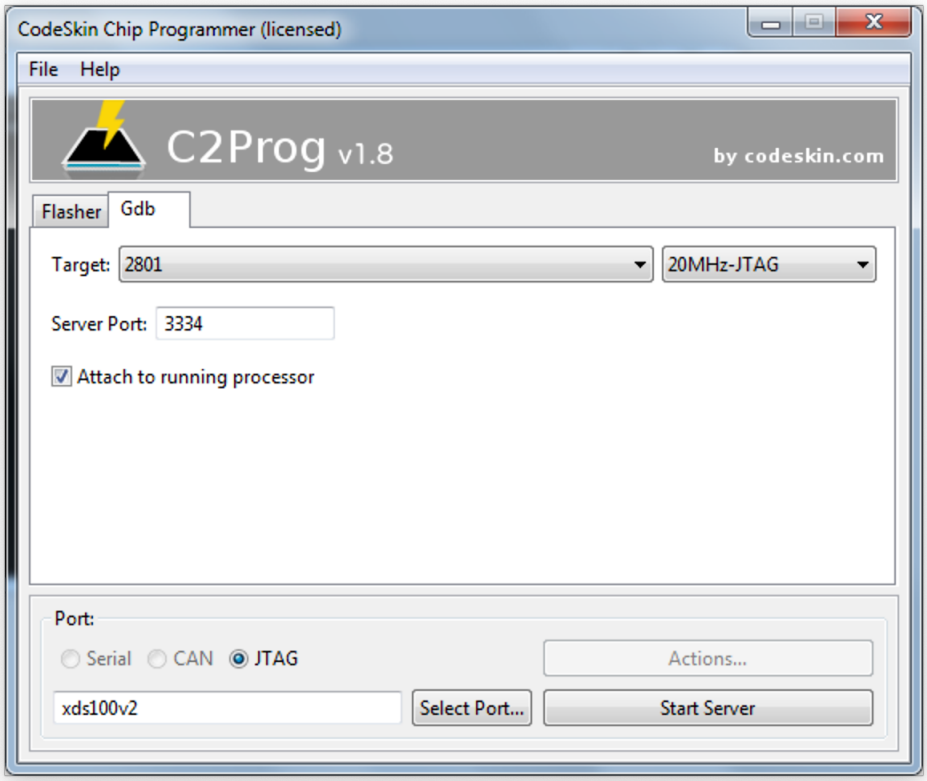
# 8 Debugging Tools

## 8.1 GNU Debug Server

C2Prog includes a GDB server. It attaches to a target through a JTAG debug probe and serves the GDB Remote Serial Protocol on a TCP port, so a protocol-speaking client can inspect and control the MCU: memory, registers, run control, and breakpoints, including hardware breakpoints in flash. It backs the **c2p-gdb** scripting client (Section 8.2) and the **c2p-agent** MCU diagnostics tools.

A few notes about the server implementation:

- C28x: addresses are word addresses as printed in the **.map** file; width is bytes per value (2 or 4); the least addressable unit size (`lau_size`) is inferred from the target. Each 16-bit word transfers little-endian. For example, PARTIDH of an F28P65x is read at address 0x5D00A with width 4.
- C29x: addresses are byte addresses. For example, RESC of an F29x device is read at address 0x30200040 with width 4.
- The server accepts multiple connections. This allows issuing a blocking 'continue' command and subsequently accessing the MCU, while it is running, through a second connection.




Standard GDB builds do not include the C28x and C29x CPU architectures, so a stock **gdb** cannot debug these targets at the source level; the intended clients are **c2p-gdb** and other tools that work at the protocol level.

### 8.1.1 Running the server from the command line

The command-line server **c2p-gdb-server** attaches to a target through a JTAG debug probe and serves the GDB remote protocol on a TCP port.

| **c2p-gdb-server options** | |
|---|---|
| `--ccs-base-path=<path>` | Path to the CCS or UniFlash EmuPack |
| `--server-port=<port>` | TCP port to serve on (default 3334) |
| `--non-intrusive` | Attach without halting the target |
| `--single` | Exit once the first session closes |
| `--core=<n>` | CPU to attach to on a multi-core device (C29: 1, 2 or 3) |
| `--help` | Display help, including the exit status table |
| `--version` | Print the version |

The last two arguments are the target configuration and the debug probe. The first command attaches to an F28P65x (C28x). The second command attaches to CPU3 of an F29x device (C29x):

```
./c2p-gdb-server --ccs-base-path=<ccs_base> --non-intrusive 28P650DK9,DK8,
   ↪ DK7,DK6,SK7,SK6_JTAG xds110
./c2p-gdb-server --ccs-base-path=<ccs_base> --non-intrusive --core=3 29
   ↪ H85xTU9,TU8-CPU1+CPU3_JTAG xds110
```

An unknown target name makes the server list the available ones.

The debug probe argument has the form `<probe>[:<serial>][:<options>]`:

- `<probe>`: the probe type, for example `xds110`, `xds100v2` or `xds200`.
- `<serial>`: the serial number of the probe, when several are connected, for example `xds110:CL850001`.
- `<options>`: a JSON object with one of the two fields below.
- `{"core":N}`: selects CPU N, as `--core` does.
- `{"ap":N}`: selects access port N. The F29x access ports are listed in Section 8.1.2.

A command line that passes `--core` carries no `<options>` object.

The probe argument `sim` selects a built-in simulator. It needs no probe and no EmuPack, but `--ccs-base-path` is still required and accepts any path. Its memory reads back what was written to it:

```
./c2p-gdb-server --ccs-base-path=. --non-intrusive 29H85xTU9,TU8-CPU1_JTAG '
   ↪ sim:{"ap":1}'
```

In a Unix shell, single quotes keep the JSON intact.

Without `--non-intrusive`, the server halts the target when it attaches. It also halts a running target when a client connects while no other client is connected. A client that disconnects leaves the target in the state it is in, and the server removes the breakpoints when the last client disconnects. When the server exits, the probe releases the target, and a halted target runs again. The server writes a line to standard error when it halts the target on attach, and another when it exits while the target is halted.

Once the server is listening, it writes a single JSON ready record to standard output; log and diagnostic messages go to standard error. A script that spawns the server can wait for this line and read the bound port from it:

```
{"v":1,"event":"ready","tool":"c2p-gdb-server 2.4.0","port":3334,
 "target":"29H85xTU9,TU8-CPU1_JTAG","emu_port":"xds110","non_intrusive":true
    ↪ ,"core":1,
 "lau_size":1}
```

The record carries the stream version `v`. Within a version, new fields may be added in the future. `core` is present only when the server was started with `--core`. `lau_size` is the size of the target's least addressable unit in bytes: 2 on C28x devices, 1 on C29x devices.

The exit status is as follows:

- 0: the server ran and shut down cleanly
- 1: failure; the reason is in the error message
- 2: usage error, such as an unknown option or target, or an `<options>` object that is not one of the two forms listed for the debug probe argument
- 5: connection error; the probe was not found or the target refused to attach
- 16: EmuPack error; the emulator plugin or the EmuPack is missing or unusable
- 17: the server port is already in use
- 130: interrupted with Ctrl-C

### 8.1.2 Monitor commands

The server implements the following `monitor` commands. In a GDB session, type e.g., `monitor status`; in a **c2p-gdb** script, call **gdb.remote_command('status')**.

| monitor commands | |
|---|---|
| `status` | Target and session state |
| `status-json` | The same state as one JSON object |
| `reset` | Reset the target |
| `halt` | Halt the target; fails while the CPU is in the error state, which a `reset` ends |
| `resume` (or `run`) | Resume the target, stepping over a breakpoint at the PC first; fails when the CPU goes to the error state |
| `run-free` | Let the target run without debug intervention |
| `set <key> <value>` | Set a device-specific parameter |
| `get <key>` | Read a device-specific parameter |

`status-json` reports the same state as `status`, as one JSON object for a script to parse:

| status-json fields | |
|---|---|
| `report` | Document identity: `c2p-gdb-status` |
| `report_version` | Document version; within a version, fields are only added |

| status-json fields (continued) | |
|---|---|
| `state` | `halted`, `running`, `breakpoint`, `free-running`, `disconnected` or `error` |
| `sessions` | Number of open connections |
| `non_intrusive` | Whether the attach left the target running |
| `thread` | Current thread |
| `target` | Debug topology of the target definition, such as `c280`; several device families share one, for example the 28P55x targets use `28p650` |
| `configuration` | Target configuration the server was started with, such as **28027,23,21_JTAG** |
| `breakpoints` | Software breakpoints the server holds patched into the target |
| `software_breakpoints` | Whether the server patches a trap instruction for `Z0`; false on F29x devices, where every `Z0` takes a comparator |
| `hardware_breakpoints_supported` | Whether this device offers address comparators |
| `hardware_breakpoints_used` | Comparators the server currently holds |
| `hardware_breakpoints` | Comparators the device has, where the count is known |
| `hardware_breakpoints_free` | Comparators still free for breakpoints and watchpoints, where the count is known |
| `watchpoint_kinds` | Kinds of data access the device can watch, as a list: `write`, `read` and `access` on an F29x CPU and a C28x CPU with an ERAD, empty where there are none |
| `watchpoint_comparators` | Comparators one watchpoint of each kind occupies, as an object such as `{"write": 1, "read": 1, "access": 2}` |
| `watchpoints_used` | Watchpoints the server holds |
| `watchpoint_comparators_total` | Comparators watchpoints draw on, where known |
| `watchpoint_comparators_free` | Those still free |
| `watchpoint_comparators_shared` | Whether every hardware breakpoint draws on the same comparators |
| `target_description` | Whether a target description is served |
| `memory_map` | Whether a memory map is served |
| `registers` | How many registers the description names |
| `core` | Selected CPU, on a device with selectable CPUs |
| `ap` | Selected access port, on such a device |

| status-json fields (continued) | |
|---|---|
| `stop_reason` | Why the target stopped: `breakpoint`, `watchpoint`, `interrupt` or `step`; absent while it runs and when the server did not see it stop |
| `watch_kind` | Kind of the watchpoint behind a `watchpoint` stop |
| `watch_address` | First address of that watchpoint |
| `watch_pc` | Address of the instruction that made the access, where the device records it |

The breakpoint fields are explained in Section 8.1.4, the watchpoint and stop fields in Section 8.1.5.

The keys for `set` and `get` depend on the selected target.

The F29x debug logic has one access port per CPU and two secure access ports (SEC-AP): one for the HSM, the security processor of the device, and one for the C29x CPUs. A debugger presents debug passwords over a SEC-AP (see *C29 Debug Password* on page 62). The emulation boot registers select the boot mode that the next reset uses while a debug probe is connected. RAMOPEN is the boot state in which the flash kernel may use the RAM that the boot ROM loads it into (see Section 6.2.8). F29x devices provide:

| F29x monitor keys | | |
|---|---|---|
| `core` | set, get | CPU: 1, 2 or 3 |
| `ap` | set, get | Access port: 0 HSM SEC-AP, 1 C29 SEC-AP, 2 CPU1, 3 HSM Cortex-M4, 4 CPU2, 5 CPU3 |
| `bootmode` | set | Boot mode for the next reset, written to the emulation boot registers: `ram`, `ram-1` or `wait` |
| `bootmode` | get | The emulation boot register value, as 8 hex digits |
| `emubooten` | get | 1 when the emulation boot registers are armed for the next reset, otherwise 0 |
| `ramopen` | get | 1 while RAMOPEN is active, otherwise 0 |
| `ramopen` | set | 1 opens RAMOPEN, 0 closes it |
| `debug-password` | set | C29 debug password to present: 32 hex digits, as in the profile's `debug.c29.password`; an empty value selects the default |
| `debugstatus` | get | Presents the debug passwords over the C29 SEC-AP, then returns the `DEBUG_ENABLE_STATUS` register as 8 hex digits |
| `num-hw-breakpoints` | get | Hardware breakpoint comparators of the selected CPU; 0 on a security access port |

| F29x monitor keys (continued) | | |
|---|---|---|
| watchpoints | get | Watchpoint kinds of the selected CPU with their comparator cost: write:1,read:1,access:2; empty on a security access port |
| num-watch-comparators | get | Comparators watchpoints draw on: 8, the pool the hardware breakpoints use; 0 on a security access port |
| watch-hit | get | The watchpoint that fired since the last run started, as <kind>,<address>,<pc> in hex; empty when none fired |
| script-revision | get | Revision of the target script |

To select a CPU, use set core; the server translates the CPU number to its access port. ap selects an access port directly, which is how the two security access ports are reached. Once the server has attached, both can be changed while it runs. The switch takes effect on the next access, and a running target keeps running. Access port 3, the HSM Cortex-M4, does not connect on an HS-FS device.

The server attaches through the access port selected at startup: the port of the target's CPU, unless --core or the probe options select another. When that port is closed, the attach fails with Connect:95 before any monitor command can run.

A CPU access port is closed when the active security configuration of the device is in SSU Mode 3 (see *The Security Configuration* in Section 6.2.1) and sets its own C29 debug password. To open it, start the server on the C29 SEC-AP, access port 1:

```
./c2p-gdb-server --ccs-base-path=<ccs_base> 29H85xTU9,TU8-CPU1_JTAG 'xds110
   ↪ :{"ap":1}'
```

Then run a script such as **unlock.lua** against it. Replace the password with the one in the profile's debug.c29.password:

```
gdb.remote_command('set debug-password 00112233445566778899AABBCCDDEEFF')
print(gdb.remote_command('get debugstatus'))
gdb.remote_command('set core 1')
```

```
./c2p-gdb script --port 3334 unlock.lua
```

get debugstatus presents the configured password. A password the device refuses closes the C29 debug gate, so get debugstatus then presents the default password. Bit 0 of the returned value is set when the C29 debug gate is open. After the script, the server stays attached to CPU1 for the next client.

F28x devices provide set bootmode. The value, flash or ram, is the boot mode the target script prepares the device for each time the target is reset. The 280013x, 280015x, 28002x, 28003x, 2838x, 28P55x and 28P65x targets support both values. The 2802x, 2803x, 28004x,

2806x, 2807x and 2837x targets support `flash` only.

The 28002x, 28003x, 28004x, 2838x, 28P55x and 28P65x targets also provide `get watchpoints`, `get num-watch-comparators` and `get watch-hit`, as on F29x devices. The `watch-hit` value carries no PC: `<kind>,<address>`.

With an XDS probe, the probe adapter provides further keys:

| XDS probe monitor keys | | |
|---|---|---|
| `supported-reset-types` | get | The reset types the debug driver offers |
| `reset-type` | set | Reset type `monitor reset` uses, by its number in that list |
| `block-reset` | set, get | Block device resets: 0 off, 1 once, 2 stay |

---

Switching `core` or `ap` clears the breakpoints and watchpoints the server holds: each patched instruction is restored and each hardware comparator is released on the CPU it was set on before the switch takes effect.

---

### 8.1.3 Target description and memory map

The server serves a GDB target description over `qXfer:features:read:target.xml`. GDB and the **c2p-gdb** scripts read it to address registers by name: PC, ACC, XAR0 and so on for C28x; D0, A0, PC and so on for C29x. One description exists per CPU family and covers every JTAG-capable F28x device and every F29x device. The `g` and `G` packets transfer the registers the description names, in ascending register number. A device without a description reports both packets unsupported.

A memory map is served over `qXfer:memory-map:read`, generated from the selected target's flash layout and the RAM regions its definition declares. It is advertised in `qSupported` only when the target has one.

### 8.1.4 Interrupting and breakpoints

To break into a running target, send the interrupt character (Ctrl-C, 0x03) while the continue is outstanding; **gdb.interrupt** sends it from a script and waits for the reply. The target halts and the continue completes with SIGINT, on the same connection.

On F28x devices, a software breakpoint (`Z0`) patches the trap instruction ESTOP0 over the target's own. The original instruction is restored when the breakpoint is cleared, when the session ends, and when the server shuts down, so no trap remains in the application. The server reads the patch back after writing it, so a write that flash or a protected region

silently dropped does not become a breakpoint that is not there. A continue or step that starts at a breakpoint executes the original instruction once and leaves the breakpoint armed, so a loop through the address stops on every pass. When that instruction leads to another breakpoint, the continue stops there. A breakpoint cleared while the CPU is stopped on it, a PC write there, and the end of the session leave the CPU ready to execute the original instruction.

On F29x devices, every `Z0` occupies an ERAD comparator, in RAM as in flash, and application memory is never modified. The instructions of this device are 16, 32 or 48 bits long, while its trap instruction is 16 bits. A trap patched over a longer instruction leaves the rest of that instruction to be decoded as instructions of their own.

Memory reads report the original instruction at a breakpoint address, so a disassembly or a comparison against the programmed image still matches. A memory write to a breakpoint address updates the instruction that the restore later puts back; the breakpoint stays armed.

A hardware breakpoint (`Z1`) occupies one of the device's address comparators and works at any code address including flash. Comparators are scarce: two on a C28x CPU, eight per CPU on F29x devices, and a set beyond the last free comparator fails with an error. On F28x devices with an ERAD (F28002x, F28003x, F28004x, F2838x, F28P55x and F28P65x), a hardware breakpoint beyond the second occupies an ERAD comparator instead, so up to ten are available, shared with the watchpoints (see Section 8.1.5). The `hardware_breakpoints` and `hardware_breakpoints_free` fields of `status-json` include those comparators. A C28x CPU applies a comparator change at its next halt and run, so the server halts a running target briefly around the change. Where the device offers them, a `Z0` whose trap patch fails its readback is placed as a hardware breakpoint instead, so a breakpoint in flash-resident code works without the client doing anything differently. Hardware breakpoints follow the same cleanup as patched traps: they are released when cleared, when the session ends, when the server shuts down, and before a `core` or `ap` switch.

Two mechanisms provide the comparators. On F28x devices the TI debug driver places them through the device's analysis unit. On F29x devices the debug driver implements none, so the server places them itself: the target script programs the device's ERAD bus comparators over the debug connection, claiming each one through ERAD's ownership handshake. Comparators the application owns (ERAD is also an application-visible peripheral, used for profiling) are left alone and reduce what the debugger can take; the pool state is visible in `status-json` as described above.

Where the server patches no trap, `Z0` is placed as a hardware breakpoint if the device has comparators, and reported unsupported if it has none. `status-json` then reports `software_breakpoints` as false. A step (`s`) on a running target fails.

### 8.1.5 Watchpoints

A watchpoint halts the target when it accesses a range of memory. A write watchpoint (`Z2`) halts on a store, a read watchpoint (`Z3`) on a load, and an access watchpoint (`Z4`) on either. The F29x CPUs and the C28x CPUs that have an ERAD (F28002x, F28003x, F28004x, F2838x, F28P55x and F28P65x) support all three kinds. Watchpoints occupy comparators of the CPU's ERAD, the Embedded Real-time Analysis and Diagnostic module:

- a write watchpoint occupies one comparator, which compares the address of every data write
- a read watchpoint occupies one comparator, which compares the address of every data read
- an access watchpoint occupies two comparators, one of each

On F29x devices, these are the eight comparators the hardware breakpoints use. On C28x, the first two hardware breakpoints come from the CPU's analysis unit. Watchpoints use the eight ERAD comparators, which they share with any further hardware breakpoints. In `status-json`, `watchpoint_comparators` states the cost per kind, and `watchpoint_comparators_total`, `watchpoint_comparators_free` and `watchpoint_comparators_shared` describe the pool.

One comparator covers a range whose length is a power of two addressable units (bytes on C29x devices, 16-bit words on C28x devices) and whose start address is a multiple of that length. A client states the length in bytes, as in the `Z` packets, so on C28x it is an even number. A range of any other shape is refused with an error. An access halts the target when any unit it reads or writes falls in the range.

The stop reply names the watchpoint: `watch:<address>`, `rwatch:<address>` or `awatch:<address>`. In addition, `status-json` reports `stop_reason` as `watchpoint`, and the watchpoint in `watch_kind` and `watch_address`. On F29x devices, `watch_pc` holds the address of the accessing instruction. On F28x devices the ERAD records no such address, and the field is absent. The PC of the halted CPU lies some instructions after the access. When a call, branch or return follows the access, the PC is at its target, which can be in another function. On F28x devices, where `watch_pc` is absent, the halted PC therefore does not identify the accessing code. Accesses close together produce a single stop, and `watch_pc` then names the last of them. The `state` field reads `halted` after a watchpoint stop.

On F28x devices with an ERAD clock gate (all but the F28004x), an ERAD whose clock is off ignores the debugger. The first watchpoint or ERAD breakpoint turns the ERAD clock on when the application has not, and the last one turns it off again. The ERAD is claimed for the debugger with the first watchpoint or ERAD breakpoint and handed back with the last. An ERAD the application has claimed is left alone, and a watchpoint is then refused.

Watchpoints follow the same cleanup as hardware breakpoints: they are released when cleared, when the session ends, when the server shuts down, and before a `core` or `ap` switch.

---

Watchpoints on other F28x devices are not supported, and the server reports them as unsupported. A client that offers them anyway single-steps the target and re-reads the value after every instruction, which is slow over JTAG.

---

## 8.2 GDB Client

C2Prog includes a GDB client **c2p-gdb** that can be used to interact with a GDB server, such as the C2Prog GDB server, Segger J-Link GDB Server, or `OpenOCD`.

**c2p-gdb** accepts the following primary commands and options:

| | |
|---|---|
| `c2p-gdb --help` | Display help |
| `c2p-gdb --version` | Print the version |
| `c2p-gdb script [options] <filename>` | Run Lua script |
| `c2p-gdb load [options] <filename>` | Load binary file |

| **c2p-gdb options** | |
|---|---|
| `--endian=[big]/[little]` | Target endianness (default little) |
| `--lausize=<lau size>` | Size of the least addressable unit in bytes (default: the server's target, otherwise 1) |
| `--url=<server url>` | Server URL (default 127.0.0.1) |
| `--port=<server port>` | Server port (default 3334) |
| `--nvic=<nvic base>` | NVIC base address; `load` only, ARM only |
| `--eval=<lua source>` | Run the given Lua source instead of a file (short form `-e`); `script` only |
| `--json` | Write one JSON result document on standard output; `script` only |
| `--server-cmd="<cmd>"` | GDB server launch command |
| `--server-start-delay=<delay>` | Startup delay for a server that announces no ready record [ms] (default 1000) |
| `--server-ready-timeout=<ms>` | How long to wait for a **c2p-gdb-server** ready record [ms] (default 30000) |

The following command connects to port 3333 of a GDB server and executes the script contained in **my_script.lua**:

```
./c2p-gdb script --port 3333 my_script.lua
```

Scripts are written in the Lua programming language using **gdb** methods to interact with the GDB server. See the **<c2prog-installation-path>/scripts** folder for examples. The first script resets an F28x device, configures boot from flash, and starts the application:

```
-- reset target
gdb.restart()
-- configure boot from flash
gdb.write_memory(0xD00, {0x55aa, 0x0b}, 2)
-- start application; the script ends while it runs
gdb.remote_command('resume')
```

The second script reads the latched reset causes (RESC) of an F29x device without stopping it:

```
local resc = gdb.read_memory(0x30200040, 1, 4)[1]
print('RESC 0x%08X' % resc)
```

The next command connects to port 3333 of a GDB server and programs a target with the binary file **my_firmware.elf**. Providing the `--nvic` option for ARM processors runs the application automatically after it has been programmed.

```
./c2p-gdb load --port 3333 --nvic 0x8000000 ~/Desktop/my_firmware.elf
```

With the `--server-cmd` option it is possible to launch the GDB server automatically for the duration of the interaction with the GDB client. A bare **c2p-gdb-server** in the command is found next to the **c2p-gdb** executable. The first command reads the part ID of an F28P65x, the second reports how an F29x device started; both leave the application running:

```
./c2p-gdb script --json --server-cmd="c2p-gdb-server --ccs-base-path=<
   ↪ ccs_base> --single --non-intrusive 28P650DK9,DK8,DK7,DK6,SK7,SK6_JTAG
   ↪ xds110" -e "gdb.emit(gdb.read_memory(0x5D00A, 1, 4))"
./c2p-gdb script --json --server-cmd="c2p-gdb-server --ccs-base-path=<
   ↪ ccs_base> --single --non-intrusive 29H85xTU9,TU8-CPU1_JTAG xds110"
   ↪ scripts/status_29h85x.lua
```

When the command names **c2p-gdb-server**, the client waits for its ready record and starts the session as soon as the server is listening; `--server-ready-timeout` bounds the wait. For a third-party server (OpenOCD, J-Link), which prints no ready record, the fixed `--server-start-delay` applies:

```
./c2p-gdb load --server-cmd="~/opt/openocd/bin/openocd -s ~/opt/openocd/
   ↪ share/openocd/scripts -f board/stm32g431.cfg" --server-start-delay
   ↪ 1000 --port 3333 --nvic 0x8000000 ~/Desktop/my_firmware.elf
```

### 8.2.1 Driving the client from a program

For build pipelines and AI assistants, `c2p-gdb script` also reads the script from standard input (file name –) or from the command line (`--eval`):

```
./c2p-gdb script --port 3334 --eval "gdb.emit(gdb.status())" --json
echo "gdb.emit({pc = gdb.read_register('PC')})" | ./c2p-gdb script --port
   ↪ 3334 - --json
```

With `--json`, standard output carries exactly one JSON document; everything the script prints goes to standard error. JSON is not meant for direct reading; have an AI assistant (Section 7.2) or a pipeline consume it. Without `--json`, the script's prints are the output.

For a program that reads the document, its fields are:

- `report`: the document name, `c2p-gdb-result`
- `report_version`: the document version; within a version, fields are only added
- `tool`: the producing tool and its version
- `exit_code`: the exit status
- `layer`: `client` when `exit_code` is a **c2p-gdb** status, `server` when it is the status of a **c2p-gdb-server** that failed to start
- `error`: the error message, when the run failed
- `traceback`: the script lines that led to a script error, one per line, when there is more than one
- `result`: the value the script passed to **gdb.emit**

The exit status reports the failure class:

- 0: success
- 1: failure; the reason is in the error message
- 2: usage error, such as a missing script or an unknown option
- 5: connection error; the GDB server could not be reached
- 6: target error; the remote rejected an operation such as a memory or register access
- 7: script error; the Lua script raised an error
- 130: interrupted with Ctrl-C

When a **c2p-gdb-server** launched with `--server-cmd` fails to start, the client exits with the server's status from the list in Section 8.1.1, and the `--json` document sets `layer` to `server`. Codes 1, 2, 5 and 130 name the same failure class in both lists; every other code appears in one list only.

### 8.2.2 Script API

Scripts access the target through the **gdb** table. Addresses and values are integers, tables are indexed from 1, and a width is the number of bytes per value (1, 2 or 4, default 1). The third column states what a call does when the operation fails: it either raises a Lua error, which a script catches with `pcall`, or returns false or nil.

| gdb target access methods | | |
|---|---|---|
| **Method** | **Result** | **On failure** |
| **gdb.read_memory(addr, n, width)** | Table of n values | raises |
| **gdb.write_memory(addr, t, width)** | Writes the values in table t | raises |
| **gdb.read_register(reg)** | Value of a register, by name or number | raises |
| **gdb.write_register(reg, v)** | Writes a register, by name or number | raises |
| **gdb.registers()** | Table of register names and their numbers | |
| **gdb.status()** | The `status-json` document, as a table | raises |
| **gdb.remote_command(cmd)** | Output of a monitor command, as a string | raises |
| **gdb.set_breakpoint(addr)** | Sets a software breakpoint | raises |
| **gdb.clear_breakpoint(addr)** | Clears a software breakpoint | raises |
| **gdb.set_watchpoint(addr, n, kind)** | Sets a watchpoint over n bytes; kind is `'write'` (default), `'read'` or `'access'` | raises |
| **gdb.clear_watchpoint(addr, n, kind)** | Clears a watchpoint set with the same arguments | raises |

| gdb run control and file methods | | |
|---|---|---|
| **Method** | **Result** | **On failure** |
| **gdb.run_to(addr, timeout)** | Runs to an address and returns whether it was reached within timeout ms (default 0, no limit); the target is halted afterwards | raises |
| **gdb.restart()** | Resets the target; true on success | false |
| **gdb.step()** | Executes a single instruction of a halted target; true on success | false |
| **gdb.async_cont()** | Starts a run without waiting, unless a run of the script is outstanding; true on success | false |

| gdb run control and file methods (continued) | | |
|---|---|---|
| **Method** | **Result** | **On failure** |
| **gdb.cont(timeout)** | Runs and blocks until the target stops or timeout ms pass (default 0, no limit); true when it stopped | false |
| **gdb.interrupt()** | Halts a running target and waits for the reply to an outstanding run; true on success | false |
| **gdb.is_running()** | Whether a run this script started is still outstanding | |
| **gdb.read_object(obj, annex)** | A qXfer object such as the target description, as a string | nil |
| **gdb.query_target()** | Configures a generic ARM target for **gdb.load_binary** | raises |
| **gdb.load_binary(file)** | Loads a binary file into the target and returns its entry address | nil |
| **gdb.create_binary(file, t, width)** | Writes an ELF file from table t, which maps each start address to a table of values; true on success | false |
| **gdb.flash_erase(addr, n)**, **gdb.flash_write(addr, t)**, **gdb.flash_done()** | Programs flash through the vFlash packets; true on success | false |
| **gdb.emit(value)** | Sets the `result` field of the `--json` document | raises |

A failed memory or register access raises a Lua error; when the script does not catch it, the run exits with status 6.

An error message starts with the position of the script line that raised it, for example `script:3: No such register: NOPE`. An error raised in a module loaded with **require** starts with the module's file name instead.

After a run that a watchpoint ended, **gdb.status()** reports `stop_reason` as `watchpoint` and the address of the accessing instruction in `watch_pc` (see Section 8.1.5). A shipped script, **watchwrite_29h85x.lua**, runs an F29x device until a variable is written and reports that address.

**gdb.cont** leaves the target running when its timeout passes: the run stays outstanding, **gdb.is_running** reports it, and **gdb.interrupt** ends it. The calls after the interrupt work as before. To start the application and end the script, use **gdb.remote_command('resume')**. To wait for an address, use **gdb.run_to**. **gdb.is_running** tracks only runs started by the script itself; **gdb.status().state** reports the state of the target. **gdb.load_binary** needs a prior

**gdb.query_target** call. The **gdb.flash_*** methods need a GDB server that implements the vFlash packets, such as OpenOCD; **c2p-gdb-server** does not.

**Script environment** Scripts run in Lua 5.4 with its standard libraries. **c2p-gdb** adds the following:

- The `%` operator formats a string: `'PC 0x%08X' % pc`, or `'%s = %d' % {name, value}` for several values. A table with string keys fills named placeholders: `'%(pc)08X' % {pc = pc}`.
- **sleep(seconds)** returns after the given number of seconds. It waits in a loop, which keeps one processor core of the host busy.
- **print** writes each string or number argument on a line of its own and skips arguments of other types. With `--json`, the output goes to standard error.
- **require** searches the directory of the script (the current directory for `--eval` and standard input), then **~/c2p**, then the **scripts** folder next to **c2p-gdb**. `require 'status_29h85x'` therefore runs a shipped recipe.

### 8.2.3 Diagnostics recipes

The **scripts** folder of the installation contains worked diagnostics recipes: reading memory and variables while the application keeps running, halting and inspecting the CPU, querying the reset and boot state, dumping a peripheral register block, and running to an address. They are named **<purpose>_<device>.lua**, and the shared helpers a recipe pulls in with **require** are named **utils_<device>.lua**.

On F28x devices, **status_28x.lua** reports the part identification registers and the lock state of each code security zone. **watch_28x.lua** samples a variable while the application runs. Neither script halts, resets or writes to the target.

**status_28x.lua** covers the F28x device families that **c2p-gdb-server** offers as JTAG target configurations. For any other target configuration it reports the run state only.

Every script opens with a comment block. Its first lines are fields:

- `Purpose:` what the script reports or does.
- `Devices:` the device families it runs on.
- `Effect: read-only`, or a list of `halts`, `resets` and `writes`.
- `Invocation:` a **c2p-gdb** command line that runs it.

The helpers named **utils_<device>.lua** carry only `Purpose:` and `Devices:`. Below the fields, the header describes the parameters to edit. Read that header for the details of a script; the scripts in the folder can be newer than this manual.

Each script prints a readable report and, under `--json`, emits the same information as the result document. A script with the effect `read-only` leaves the target running.

A recipe runs from **c2p-gdb** as a script file, or by name through **require**. The **c2p-agent** tool `run_gdb_script` runs one with the script `require 'status_29h85x'` for an F29x device, or `require 'status_28x'` for an F28x device.

### 8.2.4 Error messages

Messages such as `Connect:95` come from the debug probe or the target and pass through **c2p-gdb-server** and **c2p-gdb** unchanged. The C2Prog error codes of Section 9 do not cover them.

The table refers to the error aggregator of an F29x device. The error aggregator records, per CPU, the address, the error type and the program counter of each read, write or fetch that the security configuration denied. Its registers start at address 0x6008C000.

| debug session error messages | | |
|---|---|---|
| **Message** | **Meaning** | **Next step** |
| `Connect:95` in a server error message; the server exits with status 5 | The probe reached no open debug port. The probe may be disconnected or unpowered. On an F29x device, the security configuration may keep the selected CPU access port closed | Check the probe with the **c2p-agent** tool `list_ports`. On an F29x device, attach through access port 1 and present the debug password (Section 8.1.2) |
| `Connect:106` with `Device is held in reset`, after `set core` | The selected CPU is held in reset. CPU3 of an F29x device is held in reset in bank mode 0 (Section 3.4.2) | Select a CPU that runs. In a **c2p-gdb** script, send `set core` through `pcall` so that the script continues |
| `Error -1170` | The access port does not exist on the part or cannot be reached | Check the port number against Section 8.1.2 |
| `Cannot access register` on a write | The device refused the register write | Check the debug access that the security configuration grants the CPU |
| `Cannot access memory at address` on a peripheral read; exit status 6 | The device refused the access. Either the security configuration does not grant the selected CPU the peripheral, or the peripheral belongs to another CPU | Read through the access port of the CPU that owns the peripheral (`set core`). On an F29x device, read every register of the error aggregator on that CPU |

Read peripheral registers one at a time. On an F29x device in SSU Mode 2, a read of several peripheral registers in one call can return zero-filled values.

# 9 Error Codes

If an error occurs during programming, either a single error code or a pair of primary/secondary codes is displayed.

---

You can ask an AI assistant what an error code means; it resolves a primary C2Prog error code to its description and points you to the relevant entry in this section.

---

```
Connecting with target...
-Flash API version: 200
 OK.
Erasing flash... [] OK.
Programming...  failed (write error: 31 @ 3f6000)!
```

```
-Chip ID: 0x009F
-Chip Rev: 2
-Flash API version: 100
 OK.
Erasing flash... [] OK.
Programming...  failed (write error: 9/31 @ 0x3f6020)!
```

A single number, or the primary code of a pair, must be interpreted based on the type of MCU that is being programmed.

In the case of 240x, 280x, 2802x, 2803x, 2805x, 2806x, 2823x, and 2833x devices, the value corresponds to the error code reported by the TI flash API. Please refer to the relevant technical documentation for details.

For all other processors and custom bootloaders licensed from CodeSkin, the single number, or primary code of a pair, indicates the following:

- 1: Unknown error
- 2: Feature not supported
- 3: Unlock error
- 4: Invalid size or alignment
- 5: Buffer overflow
- 6: Invalid address
- 7: Illegal sector
- 8: Erase error
- 9: Write error
- 10: Flash pump error
- 11: Flash FSM error
- 12: OTP ECC write error
- 13: OTP data write error

The secondary code of a pair identifies the flash API-specific error code. When the primary code reads “Flash FSM error”, the secondary code corresponds to the value of the FSM status. Please review the TI technical documentation for more details or contact CodeSkin for assistance.

# Appendices

## A License

C2Prog is governed by the licensing agreement as stated in the file `C2Prog-License.pdf` available for download at `https://c2prog.com/license` and also located in the `data` folder of the C2Prog installation.

BY USING THIS SOFTWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. PLEASE READ IT CAREFULLY.

Except where otherwise noted, all documentation and software included in the C2Prog package is copyrighted by CodeSkin, LLC.

Copyright (C) 2006-2026 CodeSkin, LLC. All rights reserved.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

NO COVENANTS, WARRANTIES OR INDEMNITIES OF ANY KIND ARE GRANTED TO THE USER OF THIS SOFTWARE. CODESKIN AND ITS LICENSORS DO NOT WARRANT THAT THE SOFTWARE WILL OPERATE UNINTERRUPTED OR THAT IT WILL BE FREE FROM DEFECTS OR THAT IT WILL MEET YOUR REQUIREMENTS.

C2Prog uses several open source software packages. For the list of libraries used and their license conditions, see **Help > About C2Prog**.

### A.1 License Tiers

Three license tiers determine which interfaces are unlocked on a given machine:

- **Basic**: no license file. The graphical interface and command-line ehx generation (`mkehx` and its variants) are available.
- **Professional**: node-locked license that additionally enables command-line programming (`c2p-cli load`), the DLL and the JSON-RPC server.
- **Integration**: redistribution license, which also signs ehx files so that a recipient can verify their origin.

An AI assistant with the agentic interface registered (Section 7.2) reports the tier active on the machine.

# B 32-bit CRC Algorithm

A basic implementation of a 32-bit CRC algorithm, optimized for code space, is provided in the listing below.

```
// 32-bit CRC lookup table (poly = 0x04c11db7)
uint32_t CRC32Lookup[16]={
0x00000000L, 0x04c11db7L, 0x09823b6eL, 0x0d4326d9L,
0x130476dcL, 0x17c56b6bL, 0x1a864db2L, 0x1e475005L,
0x2608edb8L, 0x22c9f00fL, 0x2f8ad6d6L, 0x2b4bcb61L,
0x350c9b64L, 0x31cd86d3L, 0x3c8ea00aL, 0x384fbdbdL
};

uint32_t CRC32StepNibble(uint16_t nibbleIn, uint32_t crc){
  uint16_t index;
  index = (uint16_t)(crc >> 28);
  crc = ((crc << 4) | (uint32_t)(nibbleIn)) ^ CRC32Lookup[index];
  return(crc);
}

uint32_t CRC32Step(uint16_t byteIn, uint32_t crc){
  uint16_t nibble;

  // first nibble
  nibble = (byteIn >> 4) & 0x0F;
  crc = CRC32StepNibble(nibble, crc);

  // second nibble
  nibble = (byteIn) & 0x0F;
  crc = CRC32StepNibble(nibble, crc);
  return(crc);
}
```

# C C2Prog API

The following functions are provided by the C2Prog DLL. All API calls return status information c2pStatus of type int. Use **c2pGetErrorDescription** to obtain an error description.

c2pStatus **c2pInitializeLibrary()**

Initializes the programming environment. Must be called by the application prior to using any other API functions.

Returns:

- zero (0), if call successful,
- error code, otherwise; use **c2pGetErrorDescription** for a description of the error

c2pStatus **c2pProgram**
(char* fileName, char* password, char* protocol, char* port, short wait)

Initiates flash programming. Only one flash programming session can be active at any time.

Parameters:

- fileName: Full path and name of ehx file
- password: Password to decrypt ehx file. If no password is used, provide an empty string (”“).
- protocol: Reserved for future use. The string ”default” is recommended.
- port: Name of communication port
- wait: If set to 0, the function launches the programming session and returns immediately. Otherwise, the call blocks until the programming session terminates.

Returns:

- zero (0), if call successful,
- error code, otherwise; use **c2pGetErrorDescription** for a description of the error

Example:

```
c2pProgram("D:\\data\\Firmware.ehx", "", "default", "XDS100v2", 1);
```

c2pStatus **c2pGetStatus**
(int *state, double *progress, char* stateInfo, int infoStringMaxLen)

Obtains status information while a programming session is active. This function is typically used after a non-blocking call to c2pProgram to display progress, status, and error information. If a fault occurs during programming, the function returns an error code and the value of state indicates in which state the error occurred.

Parameters:

- state
  0. Idle
  1. Active
  2. Done
  3. Failed
- progress: Completion rate (0.5 = 50%, 1.0 = 100%)
- stateInfo: String describing the state; memory allocated by the caller
- infoStringMaxLen: Number of bytes allocated by caller to stateInfo string

Returns:

- zero (0), if call successful,
- error code, otherwise; use **c2pGetErrorDescription** for a description of the error

c2pStatus **c2pCloseProgrammingSession()**

Stops active programming session.

Returns:

- zero (0), if call successful,
- error code, otherwise; use **c2pGetErrorDescription** for a description of the error

c2pStatus **c2pGetErrorDescription**
(c2pStatus error, char* errorDescription, int errorStringMaxLen)

Obtains description for an error code.

Parameters:

- error: Status returned by any of the API calls
- errorDescription: String describing the error; memory allocated by the caller
- errorStringMaxLen: Number of bytes allocated by caller to the errorDescription string

Returns:

- zero (0) if call successful,
- error code, otherwise

c2pStatus **c2pGetProgressInfo**
(int *state, double *progress, char* stateInfo, int infoStringMaxLen)

**This function is deprecated and only provided for backward compatibility**.

Obtains status information while a programming session is active. This function is typically used after a non-blocking call to c2pProgram to display progress, status, and error information. If a fault occurs during programming, the function returns an error code and the value of state indicates in which state the error occurred.

Parameters:

- state
  0. Idle / Done
  1. Running
- progress: Completion rate (0.5 = 50%, 1.0 = 100%)
- stateInfo: String describing the state; memory allocated by the caller
- infoStringMaxLen: Number of bytes allocated by caller to stateInfo string

Returns:

- zero (0), if call successful,
- error code, otherwise; use **c2pGetErrorDescription** for a description of the error

# D PKCS#11 Infrastructure Setup

C2Prog supports key management through PKCS#11 (Public Key Cryptography Standards #11), a standard interface to cryptographic hardware such as Hardware Security Modules (HSMs) and smart cards.

C2Prog uses asymmetric cryptography (RSA/ECC) for code signing operations.

While C2Prog can use key files in PEM format, a PKCS#11 device keeps the private key in hardware: the key never leaves the device, signing happens on the device, and key management can be centralized.

This appendix explains how to configure and use PKCS#11 devices with C2Prog.

---

A C2Prog Professional License is required to activate PKCS#11 functionality.

---

## D.1 Required Utilities

The management and configuration of the PKCS#11 infrastructure in conjunction with C2Prog requires the following two command-line utilities:

- **OpenSSL**: A cryptographic toolkit with command-line utilities for key generation, key management, and cryptographic operations.
- **pkcs11-tool**: A utility from the OpenSC project that provides command-line access to PKCS#11 devices. This tool enables listing available slots and tokens, importing keys, querying device information, and testing PKCS#11 operations.

### D.1.1 Installing OpenSSL

OpenSSL is required for key generation and fingerprint extraction operations.

**Windows -** OpenSSL binaries for Windows can be downloaded from `https://slproweb.com/products/Win32OpenSSL.html`. After installation, add the OpenSSL binary directory to your system PATH environment variable. Alternatively, OpenSSL can be installed via package managers such as Chocolatey:

```
choco install openssl
```

**macOS -** OpenSSL is typically pre-installed on macOS. If not available, or if a newer version is required, it can be installed using Homebrew:

```
brew install openssl
```

macOS ships with LibreSSL, which differs slightly from OpenSSL; the Homebrew version provides the standard OpenSSL implementation.

**Linux -** Most Linux distributions include OpenSSL by default. If not installed, use your distribution's package manager.

To verify the installation, run:

```
openssl version
```

### D.1.2 Installing pkcs11-tool

The **pkcs11-tool** utility is part of the OpenSC project and provides command-line access to PKCS#11 devices.

**Windows -** Download and install OpenSC from `https://github.com/OpenSC/OpenSC/releases`. After installation, add the OpenSC binary directory to your system PATH environment variable. Alternatively, OpenSC can be installed via package managers such as Chocolatey:

```
choco install opensc
```

**macOS -** Install OpenSC using Homebrew:

```
brew install opensc
```

**Linux -** Install it using your distribution's package manager.

To verify the installation, run:

```
opensc-tool --version
```

## D.2 Public Key Fingerprint

The public key's SHA1 or SHA256 fingerprint is used by C2Prog to reference a private key for PKCS#11 operations (`--sign-pubkey-fp` option). The fingerprint is a unique identifier computed from the public key and allows C2Prog to locate the corresponding private key within PKCS#11 devices without exposing the private key itself.

For example, in the shell command below, `8507…ff73` is the SHA1 fingerprint of the public key corresponding to the private key to be used for the `mkehx-cp` command:

```
c2p-cli mkehx-cp --target=29H85xTU9,TU8-CPU1_UART
--sign-pubkey-fp 85074169edce9ff564f1350d125f304fded0ff73
--pkcs11-pin 123456 test_29h85x_c29_cpu1.out
```

### D.2.1 Computing Fingerprints from PEM Files

The method for computing the public key fingerprint depends on the type of PEM file available.

**From a Public Key PEM File -** If you have a PEM file containing a public key, the fingerprint can be computed directly:

```
openssl pkey -pubin -in public_key.pem -outform DER | openssl dgst -sha1 -
    ↪ hex
```

**From a Private Key PEM File -** If you have a PEM file containing a private key, the public key must first be extracted, then the fingerprint computed from that public key:

```
openssl pkey -in private_key.pem -pubout -outform DER | openssl dgst -sha1 -
    ↪ hex
```

**From a Smart Card or HSM -** In case of a private key that is already stored in a smart card or HSM, the public key fingerprint can be obtained by first extracting the public key from the device, then computing its fingerprint. See device-specific sections below for detailed instructions.

## D.3 PKCS#11 Providers

A PKCS#11 provider is a platform-specific shared library (DLL on Windows, dylib on macOS, or SO on Linux) that implements the PKCS#11 standard interface. Each hardware device or software HSM typically provides its own PKCS#11 module. C2Prog loads these modules to communicate with cryptographic devices.

When a public key fingerprint is specified in a C2Prog command, the software will sequentially load all registered PKCS#11 provider modules and search through all available slots and tokens until a matching key is found. This allows C2Prog to work with multiple devices without explicit device selection.

You can register one or more PKCS#11 providers by configuring the `pkcs11-providers` setting. Multiple providers are specified as a semicolon-separated list of absolute paths. This example configures C2Prog for both YubiKey and OpenSC-compatible devices:

```
c2p-cli set pkcs11-providers "C:\Program Files\OpenSC Project\OpenSC\
    ↪ pkcs11\opensc-pkcs11.dll;C:\Program Files\Yubico\Yubico PIV Tool\bin
    ↪ \libykcs11.dll"
```

To view the currently configured providers:

```
c2p-cli get pkcs11-providers
```

Provider paths must be absolute. Relative paths and environment variables are not supported. Each provider library must be accessible and properly installed before C2Prog can use it.

## D.4 YubiKey

The YubiKey from Yubico is a recommended hardware security key for storing cryptographic keys used with C2Prog. YubiKey devices support the PIV (Personal Identity Verification) standard, providing secure key storage with PIN protection.

The default YubiKey PIV PIN is `123456`. It is strongly recommended to change this PIN before deploying YubiKeys in production environments.

### D.4.1 Required Software

The necessary PKCS#11 module driver and management tools can be installed through Yubico's official software packages:

- **YubiKey Manager** (ykman): Command-line tool for configuring YubiKey devices.
- **Yubico PIV Tool**: Provides the PKCS#11 module (libykcs11) and command-line utilities for PIV operations.

**Windows -** Download and install the **YubiKey Manager** from `https://www.yubico.com/support/download/yubikey-manager/`. This package includes the necessary command-line tools. Also download and install **Yubico PIV Tool** from `https://developers.yubico.com/yubico-piv-tool/`. This will install the required PKCS#11 module.

**macOS -** Install using Homebrew:

```
brew install ykman
brew install yubico-piv-tool
```

C2Prog for macOS is compiled for Intel architecture (x86_64) and uses Rosetta 2 on Apple Silicon Macs due to TI XDS emulator driver limitations. On Apple Silicon Macs, the Intel version of Homebrew (which installs to `/usr/local`) must be used to ensure compatibility with C2Prog.

On Apple Silicon Macs, install Intel Homebrew:

```
arch -x86_64 /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
    ↪ Homebrew/install/HEAD/install.sh)"
```

Then install packages using the Intel version:

```
arch -x86_64 /usr/local/bin/brew install yubico-piv-tool
```

**Linux -** Install it using your distribution's package manager.

### D.4.2 Configuring the Yubico PKCS#11 Provider

After installing the Yubico PIV Tool, locate the PKCS#11 module library and register it with C2Prog. The default installation paths are:

**Windows:**

```
C:\Program Files\Yubico\Yubico PIV Tool\bin\libykcs11.dll
```

**macOS (Intel Homebrew):**

```
/usr/local/lib/libykcs11.dylib
```

**Linux:**

```
/usr/lib/x86_64-linux-gnu/libykcs11.so
```

To verify the provider path, insert a YubiKey device and use **pkcs11-tool** to list available slots (replace the path with your OS-specific path from above):

```
pkcs11-tool --module "C:\Program Files\Yubico\Yubico PIV Tool\bin\
    ↪ libykcs11.dll" --list-slots
```

The keys stored in a YubiKey can be listed as follows (replace the slot number with the one identified above):

```
pkcs11-tool --module "C:\Program Files\Yubico\Yubico PIV Tool\bin\
    ↪ libykcs11.dll" --slot 0 --list-objects
```

Once the path has been verified, register the provider with C2Prog:

```
c2p-cli set pkcs11-providers "C:\Program Files\Yubico\Yubico PIV Tool\bin\
    ↪ libykcs11.dll"
```

### D.4.3 Generating Keys on YubiKey

Keys can be generated directly on the YubiKey device, ensuring the private key never exists outside the secure element. The YubiKey PIV implementation provides four key slots suitable for code signing operations:

- Slot 9a: PIV Authentication

- Slot 9c: Digital Signature
- Slot 9d: Key Management
- Slot 9e: Card Authentication

For signing operations, slot 9c (Digital Signature) is recommended as it is specifically designated for digital signature purposes. However, any of these four slots can be used in conjunction with C2Prog as long as the proper policies are set (see below).

Keys can be generated using the **ykman** command-line tool:

```
ykman piv keys generate --algorithm RSA4096 --pin-policy ALWAYS --touch-
    ↪ policy NEVER 9c public_key.pem
```

This command generates the key pair on the device and exports the public key to **public_key.pem**. The private key remains secured within the YubiKey.

The `--pin-policy` option controls when PIN verification is required. For best security set it to `ALWAYS`.

The `--touch-policy` option controls when a physical touch of the key is required to unlock it. For use with C2Prog, a touch requirement is not practical; set it to `NEVER`.

After generating the key, obtain the public key fingerprint:

```
openssl pkey -pubin -in public_key.pem -outform DER | openssl dgst -sha1 -
    ↪ hex
```

This fingerprint can then be used with C2Prog's `--sign-pubkey-fp` option to reference the private key stored in the YubiKey.

#### D.4.4 Importing Existing Keys to YubiKey

Existing keys can be imported to the YubiKey using the **ykman** command-line tool:

```
ykman piv keys import --pin-policy ALWAYS --touch-policy NEVER 9c
    ↪ private_key.pem
```

---

⚠ Importing a key to a YubiKey does not erase the original key file. After confirming that the imported key works, securely delete the original private key file or move it to secure offline storage. The key cannot be extracted from the YubiKey once imported.

---

### D.4.5 Exporting Public Key and Computing Fingerprint

If a key pair has already been generated or imported into a YubiKey slot, the public key can be exported to compute its fingerprint for use with C2Prog.

To export the public key, for example from slot 9c:

```
ykman piv keys export 9c public_key.pem
```

Once the public key has been exported to a PEM file, compute the SHA1 fingerprint:

```
openssl pkey -pubin -in public_key.pem -outform DER | openssl dgst -sha1 -
    ↪ hex
```

The output will display the fingerprint in hexadecimal format, for example:

```
85074169edce9ff564f1350d125f304fded0ff73
```

This fingerprint can then be used with C2Prog's `--sign-pubkey-fp` option to reference the private key stored in the YubiKey.

## D.5 Nitrokey HSM 2

The Nitrokey HSM 2 is a USB hardware security module that provides secure cryptographic key storage and operations. It implements the PKCS#11 standard and supports various cryptographic algorithms including RSA and ECC for code signing operations.

---

The default Nitrokey HSM 2 SO-PIN (Security Officer PIN) is `3537363231383830` and the default User PIN is `648219`. It is strongly recommended to change these PINs before deploying Nitrokey HSM 2 devices in production environments.

---

### D.5.1 Required Software

The necessary PKCS#11 module driver and management tools are provided by the OpenSC project. Refer to the sections above for OpenSC and **pkcs11-tool** installation instructions for your platform.

### D.5.2 Configuring the OpenSC PKCS#11 Provider

After installing OpenSC, locate the PKCS#11 module library and register it with C2Prog. The default installation paths are:

**Windows:**

```
C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc-pkcs11.dll
```

**macOS (Intel Homebrew):**

```
/usr/local/lib/opensc-pkcs11.so
```

**Linux:**

```
/usr/lib/x86_64-linux-gnu/opensc-pkcs11.so
```

To verify the provider path, insert the device and use **pkcs11-tool** to list available slots (replace the path with your OS-specific path from above):

```
pkcs11-tool --module "C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc
    ↪ -pkcs11.dll" --list-slots
```

The keys stored in a Nitrokey HSM 2 can be listed as follows (replace the slot number with the one identified above):

```
pkcs11-tool --module "C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc
    ↪ -pkcs11.dll" --slot 0 --list-objects
```

Once the path has been verified, register the provider with C2Prog:

```
c2p-cli set pkcs11-providers "C:\Program Files\OpenSC Project\OpenSC\
    ↪ pkcs11\opensc-pkcs11.dll"
```

### D.5.3 Initializing the Nitrokey HSM 2

Before first use, the Nitrokey HSM 2 must be initialized. This process sets up the device and establishes the SO-PIN (Security Officer PIN) and User PIN.

To initialize the device with default PINs:

```
sc-hsm-tool --initialize --so-pin 3537363231383830 --pin 648219
```

For production use, initialize with custom PINs:

```
sc-hsm-tool --initialize --so-pin <new-so-pin> --pin <new-user-pin>
```

The SO-PIN is required for administrative operations such as unblocking the User PIN or reinitializing the device. The User PIN is required for normal cryptographic operations including key generation and signing.

### D.5.4 Generating Keys on Nitrokey HSM 2

Keys should be generated directly on the Nitrokey HSM 2 device, ensuring the private key never exists outside the secure element. The Nitrokey HSM 2 also supports importing existing keys, but the process requires additional tools and setup and is not covered here.

Use **pkcs11-tool** to generate an RSA key pair, for example with 4096 bits:

```
pkcs11-tool --module "C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc
    ↪ -pkcs11.dll" --login --pin 648219 --keypairgen --key-type RSA:4096
    ↪ --label "C2Prog Signing Key" --id 01
```

For ECC keys:

```
pkcs11-tool --module "C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc
    ↪ -pkcs11.dll" --login --pin 648219 --keypairgen --key-type EC:
    ↪ secp256r1 --label "C2Prog Signing Key" --id 01
```

The `--label` option gives the key a human-readable name, and `--id` assigns a unique identifier in hexadecimal format. The private key remains secured within the Nitrokey HSM 2.

### D.5.5 Exporting Public Key and Computing Fingerprint

After generating a key, export the public key and obtain its fingerprint:

```
pkcs11-tool --module "C:\Program Files\OpenSC Project\OpenSC\pkcs11\opensc
    ↪ -pkcs11.dll" --login --pin 648219 --read-object --type pubkey --id
    ↪ 01 --output-file public_key.der
```

Compute the SHA1 fingerprint from the exported public key:

```
openssl dgst -sha1 -hex public_key.der
```

The output will display the fingerprint in hexadecimal format, for example:

```
a3b2c1d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
```

This fingerprint can then be used with C2Prog's `--sign-pubkey-fp` option to reference the private key stored in the Nitrokey HSM 2.