C2Prog User Manual

Version 2.3

June 14, 2026

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

C2Prog is a comprehensive suite of programming tools designed specifically for Texas Instruments C2000™ microcontrollers (MCUs). The toolset provides versatile firmware programming capabilities through multiple communication interfaces, making it suitable for both development environments and field deployment scenarios.

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)

This multi-interface approach makes C2Prog particularly valuable for field deployment, where traditional JTAG access may be unavailable or impractical.

Some salient features of C2Prog are:

- Ease of use and reliable operation
- Support for multiple communication interfaces and protocols
- Smart flash erase with automatic sector optimization, or manual sector selection for fine-grained control
- Automatic 32-bit CRC generation for flash integrity verification during MCU bootup
- “Extended Hex” file format for comprehensive firmware distribution, encapsulating all programming settings including the secondary bootloader
- Firmware password protection
- Code signing and certificate generation
- Fast, reliable serial communication protocol that works seamlessly 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 Debug (GDB) server stub functionality

- 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
- Flexible and modular design enabling customer-specific solutions
- Lightweight application footprint
- Cross-platform compatibility (Windows, macOS, Linux, Embedded Linux) with binaries for Intel and ARM architectures

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

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

2 Quick Start

2.1 Installation

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

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

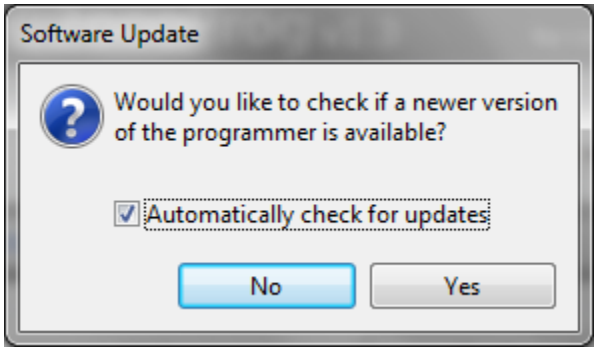
For Windows, an installer executable is provided. C2Prog can also be uninstalled from the control panel like other Windows software.

For macOS, the C2Prog application is distributed as a disk image. Simply mount the image and copy the application to your 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 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:

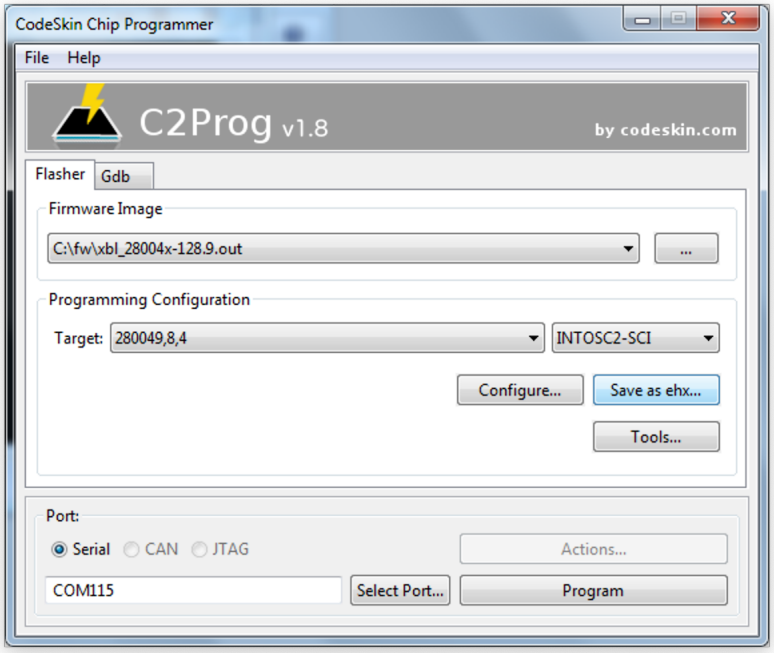
```
c29objcopy -O ihex .\Debug\test.out .\Debug\test.hex
```

ARM Tools:

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

2.3 Programming (over RS-232)

This section demonstrates the use of C2Prog in conjunction with TI's SCI bootloader (F28x device). 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.



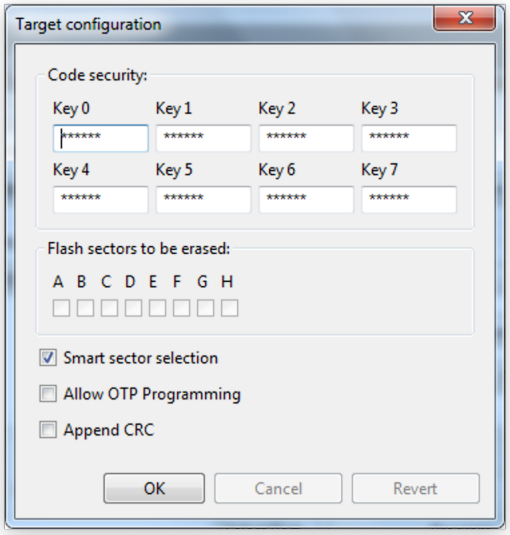
Activate the **Flasher** tab and select the **Firmware Image** (binary file) using the **…** popup menu.

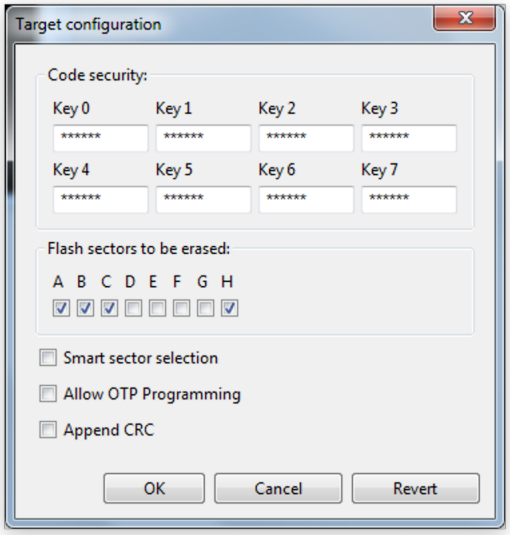
Raw binary files do not define target-specific programming information such as the MCU type to be flashed, oscillator frequency, communication protocol to be used, etc. You must therefore configure this information in the **Programming Configuration** section by choosing the appropriate MCU part name and options (e.g., clock frequency and communication interface—see Section 3.10 on page 30). 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). In the case of 32-bit keys, each key is split into two 16-bit keys in big-endian (BE) order. Note that unlike other fields, keys are not remembered as defaults.

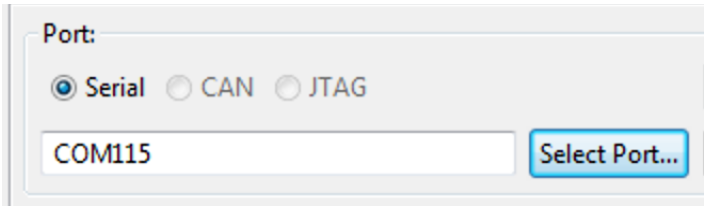
Now you must select which flash sectors should be erased prior to programming. This is done either manually by checking the individual boxes or by choosing **Smart Sector Selection**. The smart sector feature automatically detects which sectors require erasing by parsing the contents of the binary file.





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.

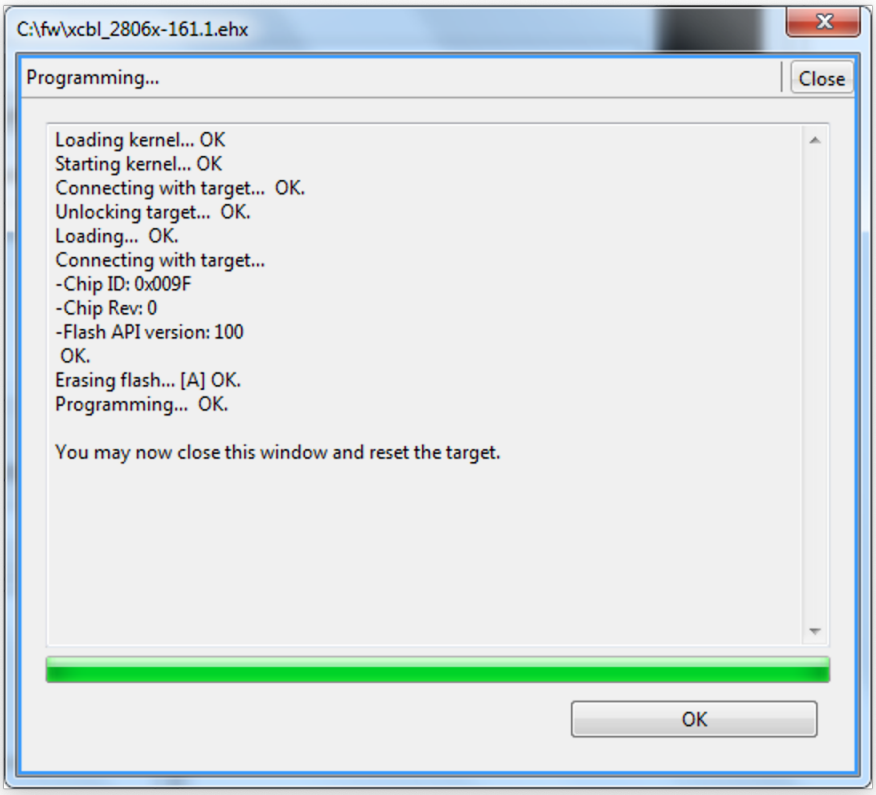
Once all programming configurations are completed, configure the COM port by either typing its name directly into the text field or clicking the **Select Port…** button. Valid entries for the serial port on Windows are COM1, COM2, etc.



The **Scan Ports** button in the port selection dialog automatically scans for all available serial ports. Please note that on some computers this feature can take considerable time to execute, 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 Detailed Description

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.

3.1.1 F29x Bank Modes

F29x devices support configurable flash bank modes that determine how flash memory is distributed between CPU cores. Understanding and properly configuring bank modes is critical for dual-core applications and must be done before key provisioning the device.

The C29 architecture supports multiple bank mode configurations:

- **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.

C2Prog automatically configures the appropriate bank mode based on the target selection. The examples below use the 29H850TU device for illustration, but the same bank mode conventions apply to other F29x targets supported by C2Prog:

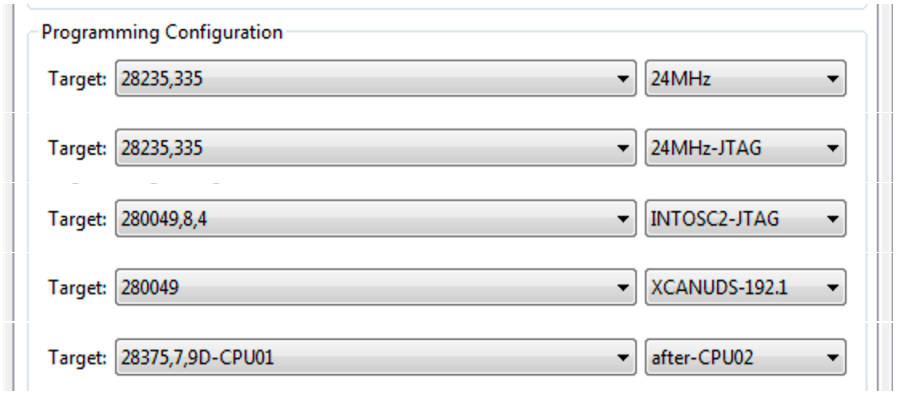
- **29H850TU-CPU1**: Programs flash with code for CPU1 in bank mode 0. Use this target for single-core applications where CPU1 requires access to all available flash memory.
- **29H850TU-CPU1+CPU3**: Programs flash with code for CPU1 and, optionally, CPU3 in bank mode 2. Use this target for dual-core applications where both CPUs execute from their respective flash regions.
- **29H850TU-CPU3**: Programs CPU3 only in bank mode 2. Use this target when updating only the CPU3 application code while preserving existing CPU1 code.

For devices in the HS-FS (High Security-Field Securable) state, C2Prog automatically configures the bank mode to match the selected target at the conclusion of the programming session. The device will be configured to boot in the appropriate bank mode on subsequent resets.

⚠ Once a device has been key provisioned and transitioned beyond the HS-FS state, the bank mode can no longer be changed. It is therefore imperative to configure the correct bank mode by flashing appropriate code while the device is still in the HS-FS state, before performing key provisioning operations.

3.2 Communication Protocols

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. The communication protocol is selected using dropdown fields in the **Programming Configuration**. Shown below are some examples:

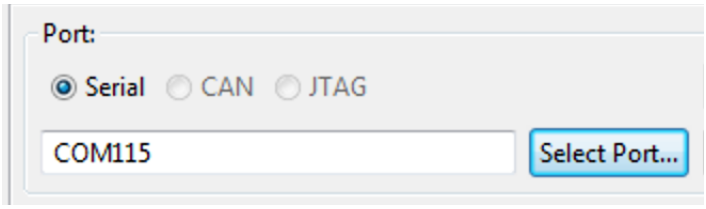


A few comments about target options:

- 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.

3.3 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 utilize the FTDI chipset. A good choice, for example, is the Parallax USB to Serial Adapter. The serial port of TI's evaluation hardware is also proven to work well with C2Prog.

3.4 Programming over JTAG

C2Prog supports TI XDS emulators for flash programming and in conjunction with the GDB server (see Section 3.12 on page 34).

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.

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

It is extremely important that the correct external clock/crystal frequency is selected, as C2Prog has no means of verifying the frequency when using JTAG. Selecting the wrong frequency may damage the MCU.

3.4.1 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 specifies use of the 2-pin cJTAG mode.
Some examples for valid configuration strings include:

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

The serial number of a particular unit can be determined using a command-line utility specific to 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`

3.4.2 XDS110 Considerations

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

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

The firmware update process is not always 100% reliable. Should your XDS110 emulator become unresponsive, 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.

The XDS110 firmware update will take a few moments during which the C2Prog UI will be unresponsive. Please allow the process time to complete.

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

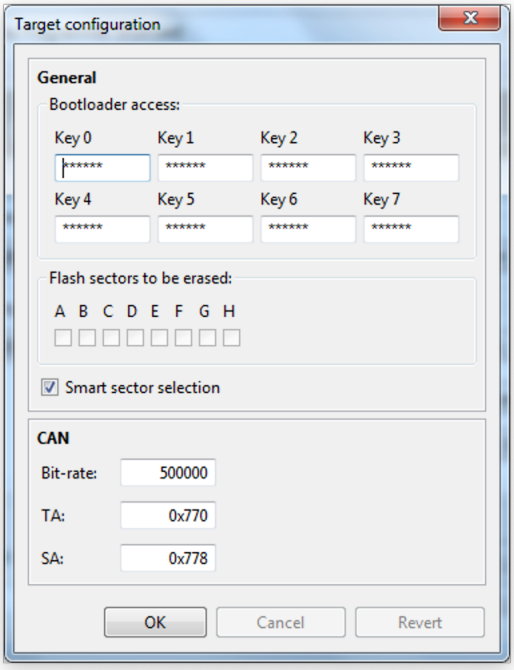
The location of the EmuPack used by C2Prog can be queried as follows:

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

3.5 Programming over CAN

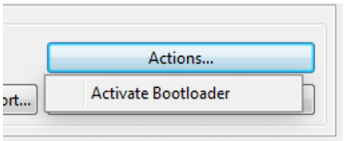
C2Prog allows programming over CAN using the ”Unified Diagnostics Protocol” as defined in ISO-14229/15765 (requires custom CodeSkin CAN bootloader).

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.



We encourage our users to contact CodeSkin for custom CAN bootloading solutions.

3.5.1 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.

3.5.2 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.

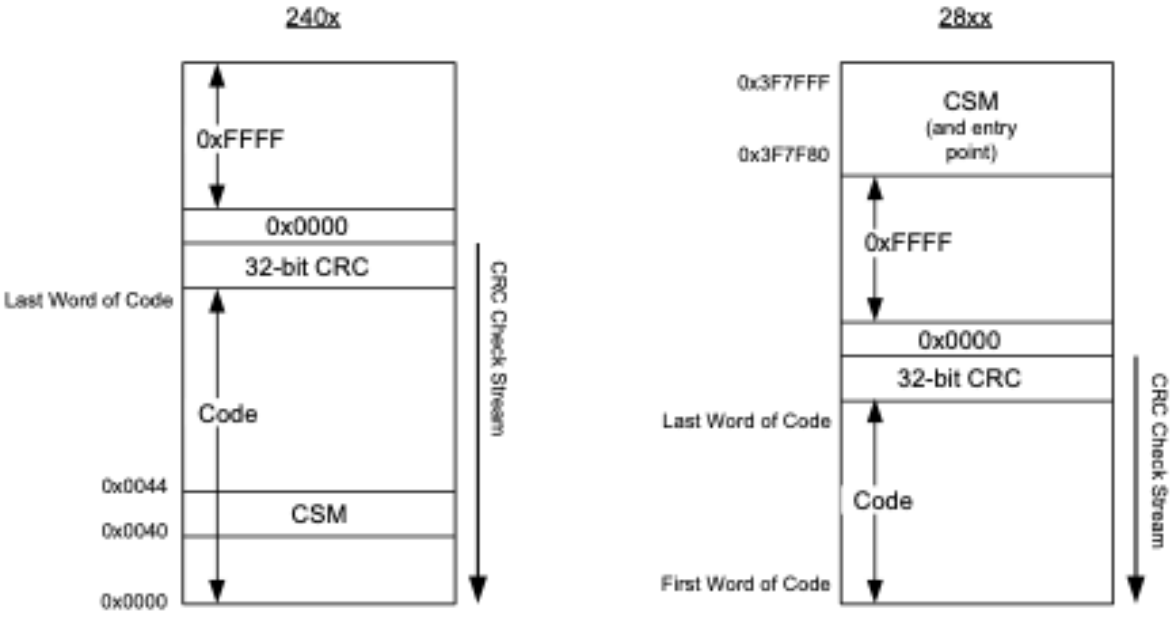
```
sudo ip link set can0 type can bitrate 500000
sudo ip link set up can0
```

3.6 Integrity and Security - 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).

3.6.1 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 "−−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 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 39 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 = CRC32Step1(((*FlashPtr >> 8) & 0xFF), CRCRegister);
  CRCRegister = CRC32Step(((*FlashPtr >> 0) & 0xFF), CRCRegister);
}
// at this point CRCRegister should be reading zero
```

3.6.2 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 will not be truly protected until the corresponding PWDLOCK is set. Please make sure that you review the relevant documentation from TI.

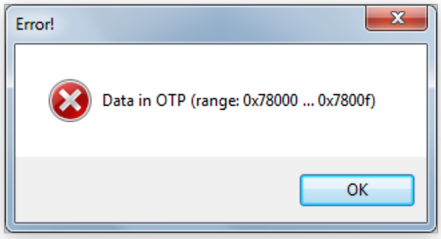
Dual Code Security Module (DCSM) In the case of MCUs with dual code security module (DCSM), C2Prog will automatically determine which zone must be unlocked to program a particular firmware image.

 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.

3.6.3 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.

It is also important to understand that 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.

It is imperative that all memory locations of a given OTP block to be programmed are explicitly specified in the firmware image, taking into account bits that have already been (factory) cleared.

3.7 Integrity and Security - F29x Devices

F29x devices implement a certificate-based security architecture through the Hardware Security Module (HSM). Unlike F240x and F28x devices that use direct flash programming and simple key mechanisms, F29x devices require X.509 certificates for all programming operations to ensure code integrity and security.

F29x devices support multiple security states (HS-FS → HS-KP → HS-SE) with increasing levels of security enforcement. The programming workflow and certificate requirements depend on the target security state and the specific security configuration needed for the application.

C2Prog automatically handles certificate requirements for F29x devices in the HS-FS (High Security-Field Securable) state. If the firmware image already includes proper certificates, those certificates are used for programming. Otherwise, C2Prog will create a default certificates to allow the device to boot.

In case of secured devices, the extended hex files for programming must be generated from the command line (**c2p-cli** , e.g as a CCS post-build step). In this case, C2Prog will also produce and sign the necessary certificates. More details are provided below.

3.7.1 Key Provisioning

Key provisioning is the process of transitioning an F29x device from the HS-FS (High Security-Field Securable) state to the HS-KP (High Security-Key Provisioned) state. This process programs cryptographic keys and security configuration into the device's one-time programmable (OTP) memory, establishing the root of trust for all subsequent operations.

⚠ Before performing key provisioning, ensure the correct flash bank mode has been configured by programming appropriate application code while the device is still in the HS-FS state. Once key provisioned, the bank mode cannot be changed. See Section 3.1 on page 12 for details on F29 bank modes.

Prerequisites Key provisioning requires the following components that must be obtained directly from Texas Instruments:

- **TIFS Collateral Package**: The TI Foundational Security (TIFS) collateral for F29H85x devices contains the necessary documentation, tools, and examples for generating security certificates and key payloads.
- **Key Generation Tools**: TI provides tools and detailed instructions for generating the cryptographic keys and assembling the final certificate binary that will be programmed into the device OTP.

Contact Texas Instruments support or refer to the TI Foundational Security documentation for F29x devices to obtain the TIFS collateral package and key generation instructions.

Key Generation and Certificate Creation Follow the procedures documented in the TI TIFS collateral to generate your cryptographic keys and create the final certificate binary. The TI documentation provides detailed instructions for:

- Generating the required RSA or ECC key pairs
- Creating the X.509 certificate structure
- Configuring security policies
- Assembling the final certificate payload

The output of this process is a binary file containing the complete key provisioning payload, typically named **final_certificate.bin**.

Generating the Certificate with the TI Cybershield Toolkit Texas Instruments distributes the TI Cybershield Toolkit (CST) as part of the TIFS collateral (available from `https://github.com/TexasInstruments/TI-CyberShield-Toolkit`). Within the C2Prog workflow CST has a single, narrow role: generating the Secondary and Backup Manufacturer signing keys and producing the one-time key-provisioning certificate (**final_certificate.bin**) that the `mkehx-kp` command consumes. All remaining steps — creating Extended Hex files, code provisioning, security-configuration provisioning, and application signing — are handled by C2Prog itself, which signs directly with the keys held in the CST session and requires no further CST involvement (see *Code Provisioning* below).

Before generating the certificate you need the TI Field Encryption Key (FEK) public key, **ti_fek_public.pem**. This file is *not* part of the CST download; it is supplied with the TI OTP Keywriter package and is specific to the device silicon revision. For an SR_10 device, for example, it is found at:

```
otp_keywriter_f29h85x_<version>\sbl_keywriter\scripts\cert_gen\common\tifek\
   ↪ f29h85x\SR_10\ti_fek_public.pem
```

Ensure the silicon-revision directory (`SR_10`, `SR_20`, …) matches the `--devSrVer` option used below.

CST stores its keys in a password-encrypted, on-disk *session*. Generating the certificate is therefore a two-step process: first generate the Manufacturer key pairs into a named session, then generate the certificate against that session.

First, create the session and generate the keys:

```
cst --device f29h85x genkeys -s MySession -p secret
```

This creates a session named `MySession` (protected by the supplied password) containing freshly generated RSA-4096 Secondary and Backup Manufacturer key pairs.

Then generate the key-provisioning certificate against that session:

```
cst --device f29h85x --session MySession --password secret gencert -t
   ↪ ti_fek_public.pem --msv 0x1E22D --smpk --smek --bmpk --bmek --sr_sbl 1
   ↪  --sr_hsmRT 1 --sr_app 1 --sr_ssu 1 --keycnt 2 --keyrev 1 -d f29h85x
   ↪ --devSrVer SR_10
```

For the `genkeys` sub-command the `-s` and `-p` options must appear *after* the sub-command name. For `gencert` (below) the `--session` and `--password` options instead appear *before* the sub-command name. The session must already exist when `gencert` is run.

By default CST writes its output, on Windows, to `C:\Users\<user>\ti\f29h85x\`

`certificates`, producing **final_certificate.bin** together with the individual **primary_cert.bin** and **secondary_cert.bin** components; use the `-o <dir>` option to write elsewhere. The **final_certificate.bin** binary is then passed directly to C2Prog as shown under *Creating the Extended Hex File* below. Refer to the TI Cybershield Toolkit documentation for the complete set of `gencert` options (write-protection flags, extended OTP, silicon-revision selection, and so on).

⚠ For enhanced security in production environments, the private keys used for signing should be stored in a hardware security module (HSM) or security token rather than as files on disk. C2Prog supports PKCS#11 infrastructure for accessing keys stored in hardware devices. Refer to Appendix D on page 44 for instructions on configuring PKCS#11 providers and importing keys to security tokens such as YubiKey devices.

Creating the Extended Hex File Once the certificate binary has been generated, use the **c2p-cli** command-line utility to create an Extended Hex file (see 3.8) suitable for programming. By default the certificate is written to CST's output directory — on Windows `C:\Users\<user>\ti\f29h85x\certificates` — so run the command from that directory or pass the full path to **final_certificate.bin**:

```
c2p-cli mkehx-kp --target=29H850TU-CPU1_UART --otp-kw otp_kw_f29h85x_hs_fs
    ↪ .hsmimage.bin final_certificate.bin
```

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

The `--target` parameter specifies both the device type 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` parameter specifies the TI Keywriter HSM runtime image. Note that TI provides different versions of keywriter binaries. You need to specify the version that is compatible with the silicon version of the chip to be provisioned.

⚠ The `--target` parameter must match the bank mode configuration that was established when the device was last programmed in the HS-FS state. For example, if the device was configured for dual-core operation using the **29H850TU-CPU1+CPU3** target, key provisioning must use a corresponding dual-core target such as **29H850TU-CPU1+CPU3_UART**. Refer to Section 3.1.1 on page 12 for details on F29x bank modes and their relationship to target selection.

 For initial testing, it is strongly recommended to use the `--dry-run` option when creating the Extended Hex file. This allows you to verify the complete key provisioning workflow without permanently committing keys to OTP memory. Once the dry run succeeds, regenerate the Extended Hex file without this option to perform the actual key provisioning.

 The `mkehx-kp` command bundles the TI Keywriter HSMrt image with the SBL and key provisioning certificate. Due to the inclusion of the TI runtime image, the resulting Extended Hex File is subject to the same licensing conditions under which you obtained the TI Keywriter HSMrt image from TI. You must comply with all applicable license terms when distributing or using this file.

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

After successful key provisioning, the device needs to be power-cycled to transition from the HS-FS state to HS-KP. The device now requires properly signed application code for all subsequent programming operations.

The immediate next step after key provisioning is to provision the device with security configurations.

3.7.2 Code Provisioning

After successful key provisioning, the device requires signed application code for all programming operations. Code provisioning is the process of programming signed firmware images to F29x devices in the HS-KP (High Security-Key Provisioned) or HS-SE (High Security-Secure) states.

Code provisioning requires X.509 certificates that authenticate the code and prove it was signed by an authorized entity. The certificates must be signed using the private keys established during key provisioning. C2Prog automatically generates and signs all required certificates upon creation of the Extended Hex file.

Generating Extended Hex Files for Code Provisioning Extended Hex files for code provisioning are created using the **c2p-cli** utility with the `mkehx-cp` command. Three signing methods are available:

For testing with PEM-encoded private key files:

```
c2p-cli mkehx-cp --target=29H850TU-CPU1_UART --sign-key-file smpk.pem
   ↪ binary.out
```

For production with PKCS#11-based signing:

```
c2p-cli mkehx-cp --target=29H850TU-CPU1_UART --sign-pubkey-fp 85074169
   ↪ edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 binary.out
```

The `--sign-pubkey-fp` parameter specifies the SHA1 (or SHA256) fingerprint of the public key corresponding to the private signing key stored in the PKCS#11 device. The `--pkcs11-pin` parameter provides the PIN for accessing the security token.

For development or testing with a key managed in a TI Cybershield Toolkit (CST) session:

```
c2p-cli mkehx-cp --target=29H850TU-CPU1_UART --cst-session MySession.
   ↪ session --cst-session-password secret binary.out
```

A *session* is a CST-specific concept: a password-encrypted, on-disk key store that holds the Secondary and Backup Manufacturer signing keys. The `--cst-session` parameter specifies the path to the **.session** file, and `--cst-session-password` provides the password used to protect it.

By default, CST stores session files on Windows in `C:\Users\<user>\AppData\Local\tisecprov\tisecprov\sessions`; supply the full path to the file (for example `C:\Users\<user>\AppData\Local\tisecprov\tisecprov\sessions\MySession.session`), or copy the **.session** file into your working directory first.

By default the Secondary Manufacturer key (SMPK) is used for signing; to sign with the backup key (BMPK) instead, add `--cst-session-keyrev 2`.

C2Prog decrypts the session in memory and uses the key directly — it is never written to disk. This applies only to software (disk-based) CST sessions; for a signing key stored on a PKCS#11 token, use the `--sign-pubkey-fp` method instead (see Appendix D).

These 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 `--cst-session` as appropriate for your key storage.

The file-based signing methods — a PEM private-key file (`--sign-key-file`) or a CST session file (`--cst-session`) — 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 44 for information on configuring PKCS#11 infrastructure and security tokens.

Security Configuration Provisioning The first step after key provisioning is to provision the device with a security configuration. Texas Instruments provides default security configuration files that should be used as the starting point for most applications. They can be found in the F29 SDK, e.g. at `C:\ti\f29h85x-sdk_<version>\source\defseccfgbin`.

For example **default_seccfg_bankmode_0_ssumode1.out**.

These files establish the basic security policies and access controls for the device.

Generate the Extended Hex file for the security configuration:

```
c2p-cli mkehx-cp --target=29H850TU-CPU1_UART --sign-pubkey-fp 85074169
    ↪ edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ default_seccfg_bankmode_0_ssumode1.out
```

This command generates **default_seccfg_bankmode_0_ssumode1.ehx**, which can then be programmed using either the C2Prog graphical interface or the command-line utility.

After successfully programming the security configuration, the device transitions from the HS-KP state to the HS-SE state. At this point, the device is ready to accept signed application code.

Application Code Provisioning Once the device is in the HS-SE state, application firmware can be provisioned. The mkehx-cp command supports provisioning with ELF files that include both security configurations and CPU code:

```
c2p-cli mkehx-cp --target=29H850TU-CPU1_UART --sign-pubkey-fp 85074169
    ↪ edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 application.out
```

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

```
c2p-cli mkehx-cp --target=29H850TU-CPU1+CPU3_UART --sign-pubkey-fp
    ↪ 85074169edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456
    ↪ application.out
```

The resulting Extended Hex file contains the signed application code and all necessary certificates for authentication by the device's Hardware Security Module (HSM).

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=29H850TU-CPU1_JTAG --sign-pubkey-fp 85074169
   ↪ edce9ff564f1350d125f304fded0ff73 --pkcs11-pin 123456 application.out
```

JTAG programming offers advantages during development, including faster programming speeds. However, JTAG access may be restricted or disabled depending on the security configuration or state of the device.

⚠ Should the device enter a security configuration in which a connection via JTAG is not possible, use the UART boot mode for recovery purposes.

3.8 Extended Hex Files

The programming configuration, secondary bootloader, HSM runtime, and contents of the binary file 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 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. Thus, it is the ideal format for distributing programming files while also preventing unauthorized use.

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 options, as below:

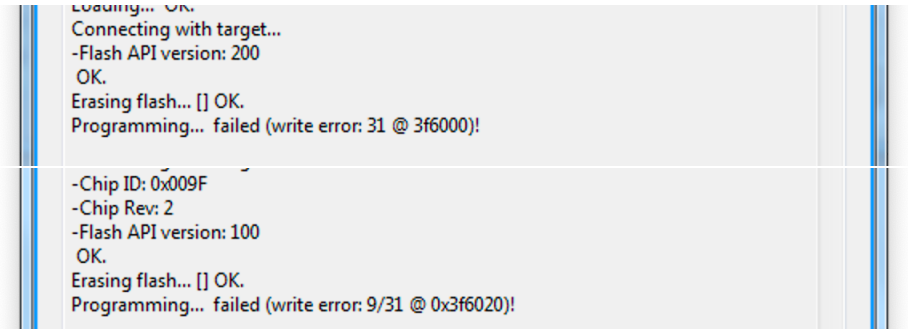
```
c2p-cli mkehx --target=28335_30MHz test.out
```

The target name 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
```

3.9 Error Codes

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



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.

3.10 Command Line Options

C2Prog can be launched from a command prompt (shell) with command-line options. This feature is available to facilitate the creation of ehx files as part of the code generation (for example, as a ”final build step” in Code Composer Studio™). Users with a ”professional” or

”integration” C2Prog license can also program MCUs via the command line and extract hex files from ehx files.

The corresponding executable is **c2p-cli**.

c2p-cli accepts the following primary commands and options:

c2p-cli --help	Display help
c2p-cli --version	Display version information
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 load [options]	Program ehx file (licensed users only)
c2p-cli extract [options]	Extract hex file from ehx file (licensed users only)

The command for creating an ehx file is

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

The <target> parameter corresponds to the concatenation of the target name and option, separated by an underscore (see page 30).

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>]
```

mkehx options	
−−target=TARGET_ID	Selects the target – the target ID is composed of 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.
−−sector-mask=SECTOR_MASK	Configures which flash sectors are erased, where SECTOR_MASK is a hex number: −−sector-mask=1: sector A −−sector-mask=2: sector B −−sector-mask=3: sectors A & B −−sector-mask=A: sectors B & D 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)
−−passphrase=PASS_PHRASE	Passphrase for extended hex file
−−bitrate=BIT_RATE	Bit rate for some protocols (such as CAN)
−−ta=TARGET_ADDRESS	Target address for multidrop protocols (such as CAN)
−−sa=SOURCE_ADDRESS	Source address for multidrop protocols (such as CAN)

load options	
−−port	Specifies communication port
−−passphrase=PASS_PHRASE	Passphrase for extended hex file
−−print-progress	Enables the display of progress information

extract options	
−−passphrase=PASS_PHRASE	Passphrase for extended hex file

mkehx-kp options	
--target=TARGET_ID	Selects the target – the target ID is composed of the target name, followed by an underscore "_" and the target option, for example "29H850TU-CPU1_UART".
--otp-kw=TI_OTP_KW_FILE	Specifies the TI Keywriter binary (HSM runtime image).
--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.

mkehx-cp options	
--target=TARGET_ID	Selects the target – the target ID is composed of the target name, followed by an underscore "_" and the target option, for example "29H850TU-CPU1_UART".
--passphrase=PASS_PHRASE	Passphrase for extended hex file
--sign-privkey=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 44 for PKCS#11 setup.
--pkcs11-pin=PIN	PIN for PKCS#11 device access
--cst-session=SESSION_FILE	Path to a TI Cybershield Toolkit (CST) password-encrypted session file for signing
--cst-session-password=PASSWORD	Password protecting the CST session
--cst-session-keyrev=REV	CST key revision to use for signing: 1 = SMPK (default), 2 = BMPK

3.10.1 F29x Notes

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

3.11 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
```

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

3.11.1 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.

3.11.2 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.

3.11.3 Shutdown command

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

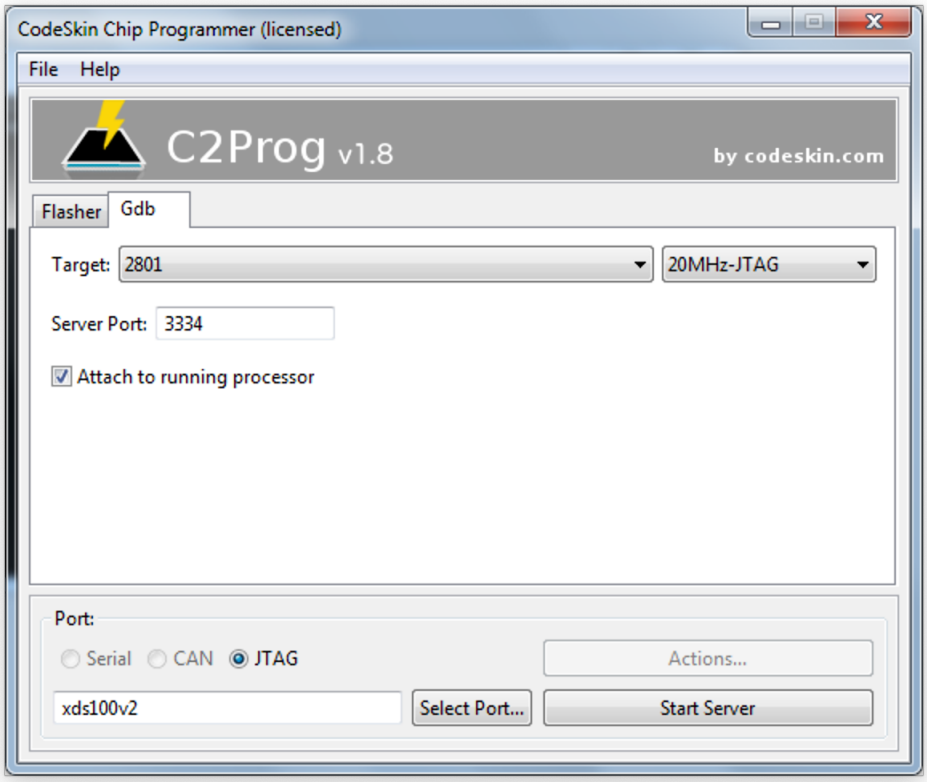
```
{
  "method": "shutdown",
  "jsonrpc": "2.0",
  "id": 0
}
```

3.12 GNU Debug Server

For *primarily experimental* purposes, C2Prog includes a rudimentary GNU Debug Server stub.

A few notes about the server implementation:

- C28 cores have a 16-bit wide architecture. Memory access must therefore be made with an even number of bytes, where each pair of bytes is interpreted as a little-endian 16-bit value.
- 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.



The functionality of the GDB server is still subject to change without notice. Please contact us if you wish to use this feature for important work.

3.13 GDB Client

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

c2p-gdb accepts the following primary commands and options:

c2p-gdb --help	Display help
c2p-gdb script [options] <filename>	Run Lua script
c2p-gdb load [options] <filename>	Load binary file

c2p-gdb options	
--endian=[big]/[little]	Target endianness
--lausize=<lau size>	Size of least addressable unit
--url=<server url>	Server URL
--port=<server port>	Server port
--nvic=<nvic base>	NVIC base address (ARM only)
--server-cmd="<cmd>"	GDB server launch command
--server-start-delay=<delay>	Server startup delay [ms]

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.

```
-- reset target
gdb.restart()
-- configure boot from flash
gdb.write_memory(0xD00, {0x55aa, 0x0b}, 2)
-- start application
gdb.cont()
```

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

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

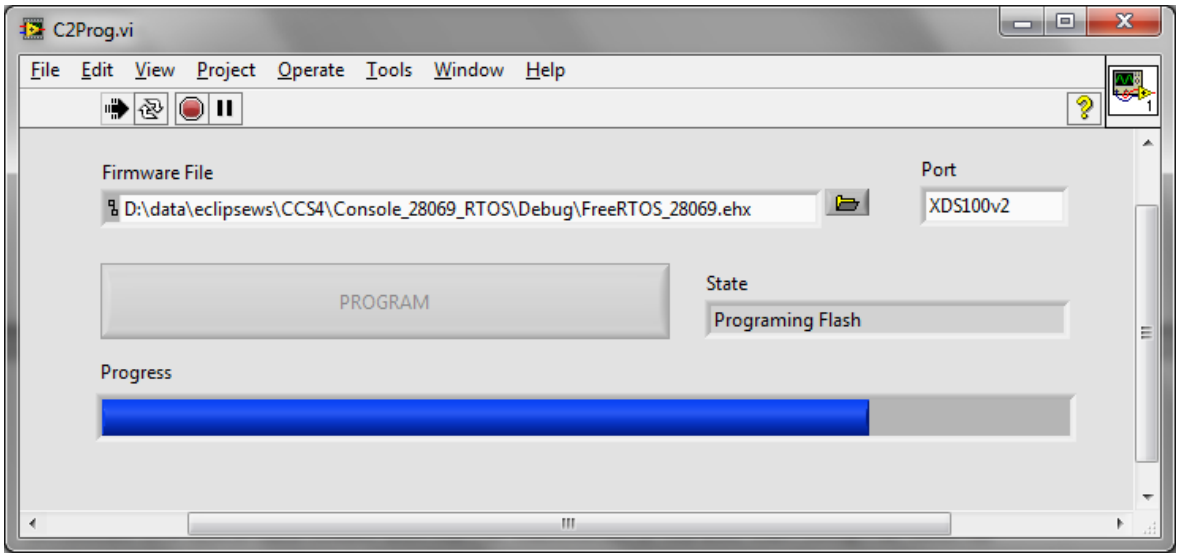
With the `--server-cmd` and `--server-start-delay` options, it is possible to automatically launch the GDB server for the duration of the interaction with the GDB client.

```
./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
```

3.14 DLL Interface

The C2Prog programming functionality is being 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 capability can easily be added to custom applications generated by 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 40 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

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.

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

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** 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 for how to use the API from different programming languages can be found in `{C2Prog}\api\examples`.

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.

C2Prog uses several open source software packages. For a comprehensive list of the libraries used and their respective license conditions, please refer to **Help→About C2Prog**.

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 description of error

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

Initiates flash programming. Note that 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 description of 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 state—memory allocated by caller
- infoStringMaxLen: Number of bytes allocated by caller to stateInfo string

Returns:

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

c2pStatus **c2pCloseProgrammingSession()**

Stops active programming session.

Returns:

- zero (0), if call successful,
- error code, otherwise—use **c2pGetErrorDescription** for description of 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 error—memory allocated by 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 state—memory allocated by caller
- infoStringMaxLen: Number of bytes allocated by caller to stateInfo string

Returns:

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

D PKCS#11 Infrastructure Setup

C2Prog supports enterprise-grade key management through PKCS#11 (Public Key Cryptography Standards #11) infrastructure. This standard provides a secure interface for cryptographic hardware devices 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, PKCS#11 infrastructure offers several advantages: keys never leave the secure hardware device, enhanced protection against key theft or unauthorized access, centralized key management for enterprise deployments, hardware-backed cryptographic operations, and compliance with security standards and regulations.

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 comprehensive cryptographic toolkit that provides 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
```

Note that macOS ships with LibreSSL, which may have slight differences 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` parameter). 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=29H850TU-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 different devices without requiring 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 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` parameter controls when PIN verification is required. For best security set it to `ALWAYS`.

The `--touch-policy` parameter controls when a physical touch of the key is required to unlock it. For use with C2Prog, a touch requirement is not practical, therefore 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` parameter 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. For maximum security, securely delete / backup the original private key file after confirming successful import and operation. The key can not 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 from e.g. 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` parameter 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. While the Nitrokey HSM 2 technically supports importing existing keys, this process is complex and requires additional tools and setup. For most use cases, generating keys directly on the device is the recommended approach.

Use **pkcs11-tool** to generate an, e.g. 4096-bit, RSA key pair:

```
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` parameter provides a human-readable name for the key, while `--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` parameter to reference the private key stored in the Nitrokey HSM 2.