Phyton™

CPI2-B1 In-System Device Programmer

User's Guide

Member of ChipProg-ISP2 family

CPI2-B1 In-System Device Programmer

© 2017 Phyton, Inc. Microsystems and Development Tools

Printed: August 2017 in (whereever you are located)

Table of Contents

© 2017 Phyton, Inc. Microsystems and Development Tools

Index 376

1   Introduction

Phyton™

CPI2-B1
In-System Device Programmers

User's Guide

Copyright © 2017, Phyton, Inc. Microsystems and Development Tools, All rights reserved

1.1   Terminology

Terms used in the document

ISP or **in-system programming**	Operations on device mounted on a board in user equipment. ICP is performed via a cable connecting programmer to the target either directly or via needles or pogo contacts.
ICP or **in-circuit programming**	Same as ISP above.
	Mode of the in-system programming that is usually defined by the programming signals voltage or the ISP interface (JTAG, SWD, UART, SPI, etc.). Distinct ISP modes are enabled for different target devices and more than one mode may exist for one device.
Target device or **Target**	A serial flash memory device, microcontroller or programmable logical device having memory inside which can be programmed by an in-system device programmer. In CPI2-B1 GUI device names comprised of part numbers (full or reduced) following types of ISP programming modes in [ ] brackets (for example: PIC10F200 [ISP HV Mode], M25PX80 [ISP Mode]).
DUT	Device Under Test - same as target device above.
Start and **End Addresses** (of the Target device)	Physical memory range of target device to perform programming operations (read, write, verify, etc.) on.
Programming Interface	On-device port that enables access to the internal memory that includes but not limited to: SPI, I2C, JTAG, SWD, UART.
ISP Mode	Mode of the in-system programming. Distinct ISP modes are enabled for

	different target devices and more than one mode may exist for one device.
ISP JTAG Mode	In-system programming using JTAG interface.
ISP SWD Mode	In-system programming using SWD (single wire debug) interface.
ISP EzPort Mode	In-system programming using Freescale proprietary EzPort interface.
ISP HV Mode	In-system programming that requires application of relatively high voltage to the target device (12V for example).
File	In the CPI2-B1 context the term **file** may represent: a) an image of information on a PC hard drive or other media that is supposed to be written into the target device's physical memory, or b) an image fetched from the target device and stored on the disk or other media. Files in ChipProg can be read from and writted to a PC hard drive or CD.
Buffer or **Memory buffer**	Buffers are intermediate data holders between data in files and data in the target device. A buffer is a portion of computer memory (RAM) used to temporarily store, edit and display data to be written to the target device or read from the device. User can open any number of buffers of any size only limited by available computer memory.
Buffer layer or **sub-layer**	A buffer may hold several layers (also known as sub-layers) that according to architecture and memory model of a particular target device. For example, for some microcontrollers one buffer can include the code and data memory layers (see more details below).
Buffer size	Buffers size may vary from 128KB to 32GB.
Buffer start address	The address to display the buffer contents from.
Checksum	An arithmetic sum of all bytes of data in a specified part of buffer calculated by programmer to ensure data integrity. The program has a variety of algorithms for checksum calculation and allows writing the checksum into a specified location of the target device.
Command Line mode	Method of controlling a CPI2-B1 in which the user issues commands to the computer program in the form of successive lines of text (command lines).
Standalone Operation Mode	CPI2-B1 device programmer contains internal memory card that can hold all information that the device programmer needs to run without further interaction with a PC.
Project	An integrated set of information that completely describes the target device, properties of data buffers, programming options and settings, list of source and destination files with their properties, etc. Each project with a unique name can be stored and promptly reloaded for immediate execution. Usually user creates a project to work with one type of device. Using projects saves a lot of time during initial configuration of programmer every time you start working with a new device.

1.2 CPI2-B1 device programmer

ChipProg-ISP2 is a family of in-system device programmers produced by Phyton, Inc. Microsystems and Development Tools. This family currently represented by two models: a single-channel CPI2-B1 and CPI2-Gx gang device programmer.

CPI2-B1 device programmers are primarily intended for use in test fixtures for programming single boards and multi-board panels. For this purpose multiple CPI2-B1 units can be driven from one computer in the gang mode. This device programmer can be also used for engineering and field service. The programmer works under control of the ChipProg-02 software package. See a single CPI2-B1 and four CPI2-B1 units mounted on a rail on the pictures below.

1.2.1 Features Overview

Features Overview

- Programs devices with Vcc from 1.2V to 5.5V.
- Supports JTAG, SWD, SPI, SCI, I²C, UART, and other interfaces.
- Extremely fast.
- Can program some devices at a long distance of up to 5m (~15ft).
- Up to 72x CPI2-B1 units can be controlled by a single computer.
- Each of ganged programmers works independently.
- USB 2.0 High Speed and LAN 100 Mbit/s communication interfaces.
- Opto-isolated RS-232 interface (optional).
- ATE interface for stand-alone operations.
- Each module has memory card that enables stand-alone operations.
- Friendly intuitive graphical user interface (GUI).

- Simplified graphical user interface for use by unskilled personnel.
- Application Control Interface (ACI) provided by a DLL.
- ACI enables control from programs in Visual Basic, C, C++, C#, etc.
- ACI enables control from National Instrument® LabVIEW™.
- On-the-fly utility allows controlling already launched programmer.
- Software includes scripting language.
- Project files are protected against hackers and corruption.
- Programmer kit includes a bracket for mounting on a standard DIN rail.
- Clip-on compartment for a battery, LEDs and a button for standalone operations (optional).

1.2.2   Hardware characteristics

NOTE. Some of the features and items below may be unavailable by the moment of sale of your CPI2-B1 device programmer

Housing Options and Applications
- Palm-size unit in a plastic enclosure.
- User-configurable gang programming system comprised of single CPI2-B1 units mounted on a standard DIN rail
- Hand-held battery-powered tool for in-field service.

Extra Options and Ordering Codes
- CPI2-B1 – single-channel programmer with no galvanic isolation of control lines.
- CPI2-ISO – single-channel programmer with galvanic isolation of control lines and RS-232 interface.
- CPI2-BB – add-on compartment with Li-Ion battery and controls for stand-alone operation.
- All above options include plastic brackets for mounting programmer units on a standard EN 50022 (TS35) 35 mm DIN rail.

Communication interfaces
- USB 2.0 High-speed.
- 100 Mbit/s Ethernet (LAN).
- RS-232C (with CPI2-ISO option only).

Powering the programmer
- From external power supply 5V/1A (not included).
- From PC USB port.
- Rechargeable Li-Ion battery (with CPI2-BB option only).

Powering Targets from the Programmer
- When powered from an external power supply (5V@1A), provides the target equipment with the voltages: Vcc (1.2 to 5.5V @ up to 350m) and Vpp (1.2 to 15V @ up to 80mA).

Signals to/from the Target
- Ten input/output lines with logical levels 1.2 to 5.5V that can be individually programmed as TTL/ CMOS logic I/O.
- The signal lines above alternate with GND lines for stable programming via long cables.
- Two input/output lines which can be individually programmed as TTL logic I/Os, GNDs, Vcc or Vpp.

Control Methods
- Start/Stop logic signal for external control.
- Output signals for external control: BUSY, GOOD and ERROR.
- Six logic inputs for choosing one of 64 preloaded projects.
- One low-current output for setting that can be used for project selection code.

- One output signal for charging an add-on battery (CPI2-BB).
- Three GND lines.

Dimensions
- CPI2-B1 unit: 114 x 73 x 32 mm (~4.5 x 2.9 x 1.25 inch).
- With CPI2-BB battery: 114 x 99 x 32 mm (~4.5 x 3.9 x 1.25 inch).

1.2.3   Software features

NOTE. Some of the features and items below may be unavailable by the moment of sale of your CPI2-B1 device programmer.

System Requirements
- Microsoft® Windows™ 7, 8 or 10.

Software Features
- Supports loading and saving files in all popular formats.
- Unlimited number of data buffers can be open and maintained.
- Enables arithmetic operations with data blocks in buffers.
- Enables writing serial numbers, MAC addresses and other device-specific parameters into user-selectable shadow areas of target devices.
- Allows writing of user-defined signatures and data blocks into target devices.
- Offers several algorithms for calculating checksums.
- Special DLL for user-defined checksum calculation.
- Writes programming session logs with real time stamps.
- The GUI has a special editor for easy setting of device and algorithm parameters, such as fuses, lock bits, boot loader vectors, etc.
- Comprehensive self-test procedure.

Managing Projects and Configurations
- The software supports unlimited number of projects.
- Project files are protected against hackers and corruption.
- The software ensures data integrity - every data transfer to/from a PC or ATE system or memory card is accompanied with CRC sum.
- The software allows storing and retrieving the state of user interface: configurations, colors, fonts, hot keys and other settable preferences.
- Battery powered option allows storing 4 projects on internal memory card; user-selectable by pressing the button on battery compartment.

Computer Control Methods
- From Automated Test Equipment (ATE), In-Circuit Test System (ICT), or programming fixtures.
- From command line or via Application Control Interface (DLL).
- Integration with National Instruments® LabVIEW™ software.
- On-the-fly management utility allows control of already launched and running device programmer.
- Built-in scripting language for writing user scripts. Auto programming can be started by closing fixture lid or by connecting a device.
- Friendly and intuitive graphical user interface (GUI) for creating and debugging projects.
- Optional simplified user interface for unskilled personnel.

Standalone Control
- The programmer can work in a standalone mode that does not require connection to a computer.
- Up to 255 standalone projects can be stored on a built-in memory card.

- Any project can be launched by ATE signals or from a computer.
- Special utility allows monitoring standalone activity on a computer.

1.2.4 Connector TARGET

TARGET connector

The **TARGET** connector positioned on the front panel enables connecting a CPI2-B1 device programmer to the target device by the 20-wire ribbon cable included in a CPI2-B1 kit. See here the connector pin assignment and description of the signals in the matrix below.

CPI2-B1 TARGET connector
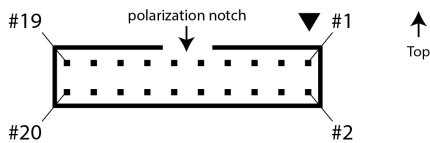
Pin#	Signal	Signal description – all signals are bidirectional
1	P1	Log 0/1, Vcc or GND
2	P11	Log 0/1, Vcc, Vpp or GND
3	P2	Log 0/1, Vcc or GND
4	GND	Ground
5	P3	Log 0/1, Vcc or GND
6	GND	Ground
7	P4	Log 0/1, Vcc or GND
8	GND	Ground
9	P5	Log 0/1, Vcc or GND
10	GND	Ground
11	P6	Log 0/1, Vcc or GND
12	GND	Ground
13	P7	Log 0/1, Vcc or GND
14	GND	Ground
15	P8	Log 0/1, Vcc or GND
16	GND	Ground
17	P9	Log 0/1, Vcc or GND
18	GND	Ground
19	P10	Log 0/1, Vcc or GND
20	P12	Log 0/1, Vcc, Vpp or GND

- P1 to P10 - logical signals formed by high-speed buffers that can output target-specific logic 0 or 1, Vcc or GND levels, according to the chosen target device type. These lines can output Vcc with levels from 1.2 to 5.5V @ up to 350mA. The buffers are bidirectional, also serving as inputs when the CPI2-B1 programmer reads data.

- P11, P12 – signals formed by high speed mixed-signal circuits that can also output target-specific logic 0 or 1, Vcc or GND levels according to the type of the chosen target device. These lines can output Vcc with levels from 1.2 to 5.5V @ up to 350mA. The mixed-signal buffers are bidirectional, also serving as inputs when the CPI2-B1 programmer reads data. In addition, these two signals can output Vpp voltage with levels from 1.5V to 15V @ up to 100mA.

The P1…P12 signals are target-specific. A CPI2-B1 user must ensure that the target device (DUT) is properly connected, according to the target-specific wiring diagram published on the **http://phyton.com/ products/isp/chipprog-isp2-family/cpi2-b1-connecting** web page. When programmer is controlled by the GUI, the same diagram can be viewed in a browser by clicking the **Connection to the target device** link in the **Device Information** window.

To "cut off" the target in the stand-by mode or after completion of any programming operation, CPI2-B1 programmer leaves the P1…P12 signals in high impedance state.

1.2.5   Connector CONTROL

CONTROL connector

The **CONTROL** connector positioned on the right side of the CPI2-B1 unit enables connecting the programmer to Automated Test Equipment (ATE) or the fixture by the 20-wire ribbon cable. See here the connector pin assignment and description of the signals in the matrix below. Since the programmer can be optionally equipped with a CPI2-ISO IO galvanic isolation board with RS232 interface, there are two different diagrams shown below.

CPI2-B1 CONTROL connector
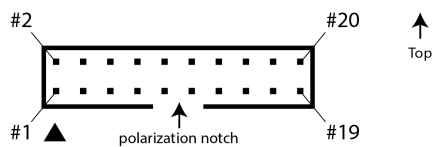
Variant WITHOUT optical isolation (CPI2-ISO is NOT installed inside of CPI2-B1)			
Pin#	Signal	Type of signal	Signal description – all signals are bidirectional
1	GND	Ground	Ground
2	GND	Ground	Ground
3	PROJ_SEL0	< Input	Project select 0; active log 1
4	START	< Input	Control signal that launches/stops programming; active: log 0
5	PROJ_SEL1	< Input	Project select 1; active: log 1
6	5V_CHARGE	Output >	5V @ 500 mA sending to battery compartment for charging the

			battery
7	PROJ_SEL2	< Input	Project select 2; active: log 1
8	5V_IN	< Input	5V input - either from external power supply or the CPI2-B1 battery
9	PROJ_SEL3	< Input	Project select 3; active: log 1
10	5V_IN	< Input	5V input - either from external power supply or the CPI2-B1 battery
11	PROJ_SEL4	< Input	Project select 4; active: log 1
12	GND	Ground	Ground
13	SAMODE	< Input	Standalone mode control; active: log 1
14	GND	Ground	Ground
15	ST_GOOD	Output >	Signal GOOD sent to ATE; active: log 0
16	GND	Ground	Ground
17	ST_BUSY	Output >	Signal BUSY sent to ATE; active: log 0
18	NC	Not connected	Not connected
19	ST_ERROR	Output >	Signal ERROR sent to ATE; active: log 0
20	NC	Not connected	Not connected

- **PROJ_SEL[4..0]** – 5-bit selector for choosing one of 32 preloaded projects - the #0 project select code is 000000, the #4 project - 000100;
- **ST_GOOD | ST_ERROR | ST_BUSY** - programmer status lines; active status: log 0;
- **START** - External signal launching and stopping the programmer; active status: log 0. If this signal remains applied to this connector pin for longer than 2 sec it switches the programmer to the Standalone Mode;
- **5V_CHARGE -** +5V @ 500mA max signal that charges CPI2-BB battery. It can be used for powering on the project selector;
- **5V_IN** – 5V supplied either from an external power adapter plugged to the programmer or from a stacked CPI2-BB compartment with a built-in battery or floating 0V if both external power adapter and CPI2-BB compartment are not connected to the CPI2-B1 unit.
- **SAMODE** - Standalone mode control; log 1 applied to this input at a moment of powering the programmer on switches the programmer to the standalone mode.

Variant WITH optical isolation (CPI2-ISO is installed inside of CPI2-B1)			
Pin#	**Signal**	**Type of signal**	**Signal description – all signals are bidirectional**
1	NC	Not connected	Not connected
2	NC	Not connected	Not connected
3	PROJ_SEL0	< Input	Optically isolated project select 0; active log 1
4	START	< Input	Optically isolated control signal that launches/stops programming; active: log 0

5	PROJ_SEL1	< Input	Optically isolated project select 1; active: log 1
6	V_ISO	Output >	Optically isolated 5V @ 10 mA max
7	PROJ_SEL2	< Input	Optically isolated project select 2; active: log 1
8	NC	Not connected	Not connected
9	PROJ_SEL3	< Input	Optically isolated project select 3; active: log 1
10	NC	Not connected	Not connected
11	PROJ_SEL4	< Input	Optically isolated project select 4; active: log 1
12	GND	Ground	Optically isolated GND line
13	SAMODE	< Input	Standalone mode control; active: log 1
14	GND_ISO	Ground	Optically isolated GND line
15	ST_GOOD	Output >	Optically isolated signal GOOD sent to ATE; active: log 0
16	GND_ISO	Ground	Optically isolated GND line
17	ST_BUSY	Output >	Optically isolated signal BUSY sent to ATE; active: log 0
18	RS232_TX	Output >	Data transmitted to computer
19	ST_ERROR	Output >	Optically isolated signal ERROR sent to ATE; active: log 0
20	RS232_RX	< Input	Not connected

- **PROJ_SEL[4..0]** – 5-bit selector for choosing one of 32 preloaded projects - the #0 project select code is 000000, the #4 project - 000100;
- **ST_GOOD | ST_ERROR | ST_BUSY** - Optically isolated programmer status lines; active status: log 0;
- **START** - Optically isolated external signal launching and stopping the programmer; active status: log 0; If this signal remains applied to this connector pin for longer than 2 sec it switches the programmer to the Standalone Mode;
- **5V_CHARGE** +5V @ 500mA max signal that charges CPI2-BB battery. It can be used as a power source for the project selector;
- **5V_IN** – 5V supplied either from an external power adapter plugged to the programmer or from a stacked CPI2-BB compartment with a built-in battery or floating 0V if both external power adapter and CPI2-BB compartment are not connected to the CPI2-B1 unit.
- **SAMODE** - Standalone mode control; log 1 applied to this input at a moment of powering the programmer on switches the programmer to the standalone mode.

1.2.6   Single- and Gang-programming control modes

ChipProg-02 software allows to drive CPI2-B1 device programmers in two different modes:

- **Single-programming** mode for programming one target device at a time by means of one CPI2-B1 programmer.

- **Gang-programming** mode for simultaneous programming of multiple devices by means of multiple CPI2-B1 programmers driven from one PC. This mode is intended for mass production in test fixtures or other ATE.

The programming mode is set in the **Startup** dialog by checking and unchecking the **Gang Mode** checkbox.

The software enables control of one, specified by its serial number, CPI2-B1 device programmer within a cluster of multiple programmers driven from one PC. This allows to switch between **Single-programming and Gang-programming** modes of control.

Gang-programming mode differs from **Single-programming** mode in the following ways:

1. in the **Gang-programming** mode only same device type may be selected for all programmers controlled by one copy of the ChipProg-02 program;
2. In the **Gang-programming** mode all programmers controlled by one copy of the ChipProg-02 program share the same data buffer;
3. Only the **Auto Programming** function can be performed by ChipProg-02 in the **Gang-programming** mode. In order to execute one command only (for example, Erase, Read, Write, etc.) it is necessary to modify a default set of **Auto Programming** commands by removing unwanted commands and leaving a single one you need.

By running several copies of the ChipProg-02 software it is possible to control some CPI2-B1 programmers controlled from one computer in the **Gang-programming** mode and others in **Single-programming** mode.

Read also about launching multiple CPI2-B1 programmers in the **Gang-programming** mode.

2   Installation and Launching

This chapter covers the following topics.

How to install the CPI2-B1 **hardware**

How to install the ChipProg-02 **software**

How to launch the CPI2-B1 device programmer.

It is **highly recommended** that before you start using the tool you read all basic topics in the chapters **Graphical User Interface** and **Operating ChipProg programmers** of this manual.

Experience using MS Windows and familiarity with basic Windows operation are required.

2.1   Getting Assistance

Context-Sensitive CPI2-B1 Online Help

The ChipProg-02 software comes with a comprehensive context-sensitive on-line **Help**. To access it press **F1** key or use **Help menu**. Almost every ChipProg-02 dialog, message box, and menu has a help item associated with it; for the active dialog or menu it can be viewed by pressing **F1**.

In most cases you can find the necessary topic by searching for a keyword. For example, if you type "Verify" in the first box of the **Find** tab, the third box will list topics related to the programming verification. Choose appropriate topic from this list and press **Display**.

A CPI2-B1 PDF manual is also available.

Technical Support

For the length of a product's warranty period Phyton provides technical support free of charge. Although we do our best to clean up and improve our software, ChipProg-02 software may contain minor bugs and some programming algorithms may not be stable on some of recently supported devices. We kindly ask you to report bugs when you get an error message or have a problem with programming a particular device or devices. We are committed to promptly checking your information and fixing discovered bugs.

To minimize difficulties using ChipProg-02 it is highly recommended to get familiar with the manual **before** using the programmer. The ChipProg-02 - user interface is quite friendly and intuitive; however, it includes some specific functions and controls that a user should learn about.

Before Contacting Phyton

- Make sure you use the latest ChipProg-02 version which is always available as free download from the **http://phyton.com/support/updates**.
- Make sure the detected error is reproducible under the same conditions and is not a casual glitch.

When Contacting Us

Please provide the following information to our technical support specialists.

- Your name, the name of your company, your contact phone, and your e-mail address.
- The CPI2-B1 serial number that can be found in the About information box or on a sticker on the CPI2-B1 bottom shell.
- Software version number taken from the About information box.
- The target device or DUT's part number.

- Basic parameters of your computer and operating system.
- Descriptions of detected errors, relevant bug reports and error screen shots.

Please send your requests or questions to **support@phyton.com**. This is the easiest way to get professional help quickly.

Contact Information

Phyton Inc., Microsystems and Development Tools

6701 Bay Parkway, Ste 3M-2
Brooklyn, New York 11204
USA

Web address: www.phyton.com

E-mail contacts:

General inquiry: info@phyton.com

Sales: sales@phyton.com

Technical Support: support@phyton.com

Tel: 1-718-259-3191

Fax: 1-718-259-1539

2.2   Hardware installation

Three connectors are situated on a rear panel of the CPI2-B1 unit: USB and Ethernet communication ports and 5V coaxial power socket. See the picture below:

Powering the programmer

A power adapter is not included into the CPI2-B1 kit. To power a CPI2-B1 device programmer use any regulated 5V/500mA+ adapter (2.1 mm, center positive). If the CPI2-B1 is controlled via a USB 2.0 port then it may get enough power via a USB port and therefore powering the programmer from an external supply is not a mandatory. However, powering the programmer from an external 5V power adapter insures more stable programming operations. Driving a CPI2-B1 device programmer via Ethernet (LAN) port always require use of an external 5V power supply.

Connecting to a computer

In case of driving a CPI2-B1 device programmer via USB it is recommended to connect the

programmer directly to a USB port on a computer main unit and do not use USB hubs. In case of controlling multiple CPI2-B1 device programmers via USB ports and use of one or more USB hubs these hubs should be powered. It is not allowed to use passive USB hubs.

Connecting to the target

See a picture of the CPI2-B1 TARGET connector below:

A ribbon cable with both-end mounted 20-pin headers are included in the CPI2-B1 kit. This cable is intended for connecting
a CPI2-B1 device programmer to the target device (board) or a test fixture in accordance to the device-specific connection diagram published on the **http://phyton.com/products/isp/chipprog-isp2-family/cpi2-b1-connecting** web page. Refer to the Connector TARGET pinout.

Grounding

Each IO signal wire in the ribbon cable alternates with ground wires. There are as many as 8 wires connected to the ground point inside of the CPI2-B1 device programmer unit. To insure stable programming operations it is **extremely important** to bring all 8 ground wires (GND) from the programmer's TARGET connector to the GND points on the target board. Do not join these GND wires in a single wire - this may cause programmer crashes!

Connecting to ATE controls

See a picture of the CPI2-B1 CONTROL connector below:

To control a CPI2-B1 device programmer from your test fixture or other ATE use the **CONTROL** port. The CPI2-B1 kit does not includes a cable with a 20-pin header to connect this port. Refer to the **CONTROL** connector pinout to make a custom connection to the ATE.

Mechanical mounting

CPI2-B1 device programmers kits include plastic brackets for mounting programmer units on a standard 35 mm DIN rail. Use these brackets for mounting multiple device programmers. See the picture below:

2.3   System Requirements

To run ChipProg-02 and control a CPI2-B1 device programmer, you need a personal computer (PC) with the following components:

- Pentium-V or higher CPU.
- Microsoft Windows 7, 8 or 10 operating system.
- A hard drive with at least 200MB of free space.
- In case of use the USB communication: at least one USB 2.0 port.
- In case of use the Ethernet communication: at least one LAN port or an Ethernet router with the Dynamic Host Configuration Protocol (DHCP).

2.4   Software Installation

Insert the disc that comes with ChipProg-02 into CD drive on your PC. When installer launces, click the **Install ChipProg-02** button, accept the license agreement, and follow the series of prompts that will guide you through the installation process.

Phyton ChipProg-02 Programmer v. 6.00.23 Installation

Transfer Working Enviromnent from Previously Installed Version

Setup has found that the following Phyton ChipProg-02 Programmer versions has been already installed on this computer. You may wish to transfer the working environment from one of the installed versions. Click the 'Details' button for more information.

☐ Transfer working environment from version:

- ◉ 6.00.23
- ○ 6.00.09

Details...

✖ Exit   ← Back   Next →

Phyton ChipProg-02 Programmer v. 6.00.23 Installation

Uninstall Previously Installed Version(s)

Setup has found that the following Phyton ChipProg-02 Programmer versions has been already installed on this computer. You may wish to uninstall some of these versions. Choose Phyton ChipProg-02 Programmer versions to uninstall before installing version 6.00.23:

- ☑ 6.00.23
- ☑ 6.00.09

Note: Version 6.00.23 has been already installed on this computer and will be uninstalled.

✖ Exit   ← Back   Next →

Phyton ChipProg-02 folder

At the end of the installation process the installer creates a folder with ChipProg-02 shortcuts.

Name ▲	Date modified	Type	Size
Phyton ChipProg-02 6.07.00	6/24/2017 5:52 PM	Shortcut	1 KB
Phyton USB Device Driver Installer	6/24/2017 5:52 PM	Shortcut	1 KB
Uninstall Phyton ChipProg-02 Programm...	6/24/2017 5:52 PM	Shortcut	1 KB

The first shortcut - **Phyton ChipProg-02,** following with the version number, - opens the startup dialog. In this dialog you can create multiple shortcuts for launching the device programmer(s) with different startup settings. All of them are accessible from the Phyton ChipProg-02 folder.

2.5   Startup Dialog

This dialog allows to setup and launch CPI2-B1 and CPI2-Gx device programmers. The dialog window is divided in a few zones: **Program Startup Options, Documentation, Contact Technical Support, For Developers.** A very bottom filed displays prompts for the dialog widget appointed by a mouse cursor - on the picture below the cursor is placed over the **Create a shortcut with this options** link in the top right corner. The picture below displays an example with some specified startup options.
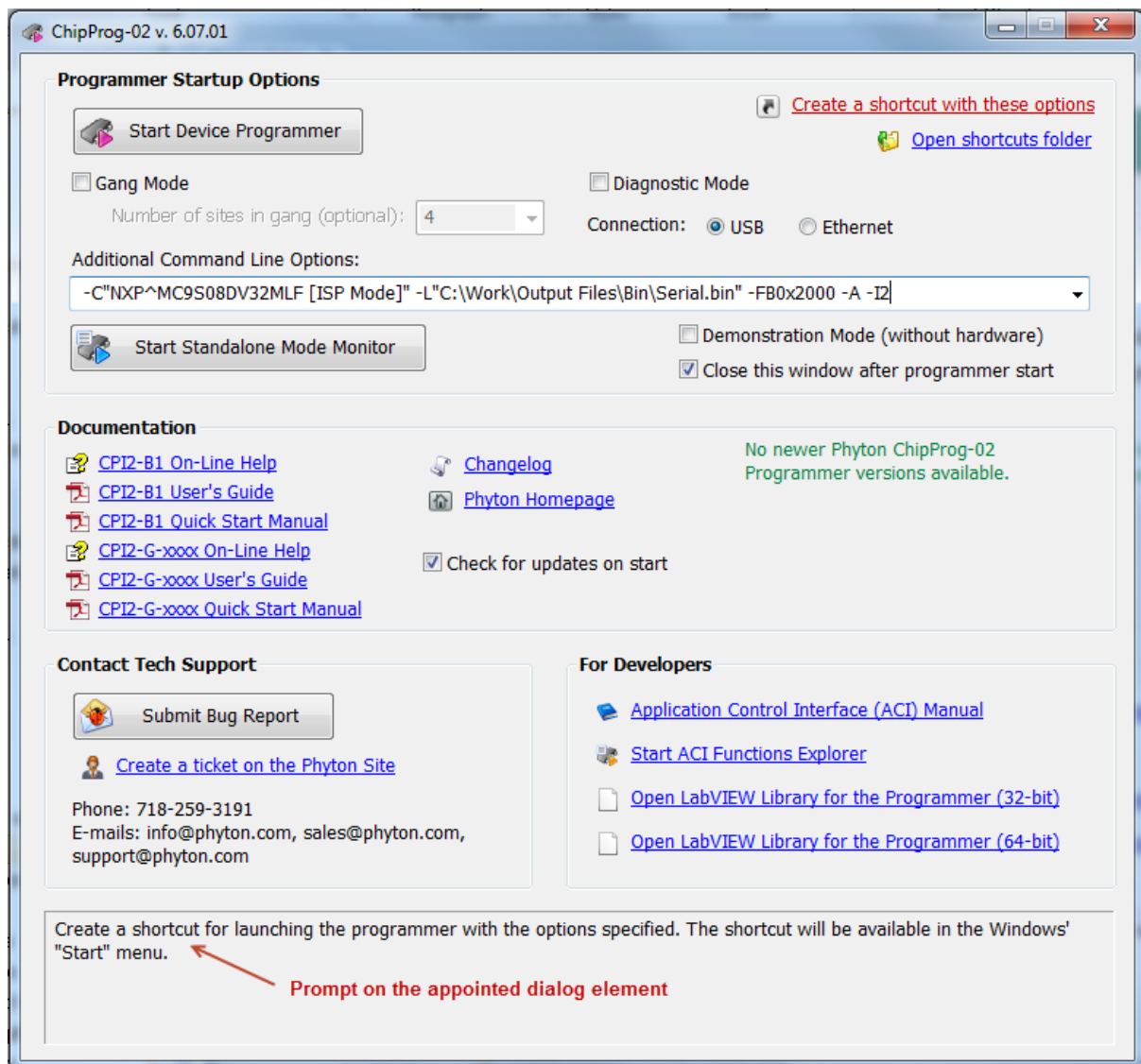
The **Program Startup Options** zone concentrates major settings, including:

Connection: Select one of communication interfaces: either **USB** (default) or **Ethernet** or Local Area Interface (LAN). Control of CPI2-B1 device programmer(s) via USB interface does not require any special settings. Connecting via Ethernet requires appropriate setting in the Command Line Command Line Options . See a description of the **-ETH** key and associated parameters (IP addresses, etc.)

Gang Mode: Leave it unchecked to control either a single CPI2-B1 device programmer or a certain one from a cluster of multiple CPI2-B1 programmers or a certain module number of a CPI2-Gx gang device programmer. Check this box to control either multiple CPI2-B1 device programmers or a CPI2-Gx gang device programmer connected to the computer.

Number of sites in gang: In this field you may optionally specify a number of CPI2-B1 device programmers or a number of programming modules in the CPI2-Gx gang device programmer to be controlled from the PC.

Diagnostic Mode: Check it if you wish the programmer would traced the programmer operations and

logged some diagnostic information in the log file. Then you may send this log file to Phyton for troubleshooting. Launching device programmers in the Diagnostic mode slows the programming down, so it is recommended only if you wish to get technical assistance from Phyton technical support.

Additional Command Line Parameters: Here you can type in command line keys which will be added to the options specified in this zone above, i.e. the **Gang Mode, Number of sites in gang, Diagnostic Mode** options. By default this field is blank.

Create a shortcut with this options: This link allows to store a shortcut for launching the device programmer with the options specified in the **Program Startup Options** zone. You may create multiple shortcuts for launching the programmers.

Open shortcut folder - Opens a folder that displays all the shortcuts launching the device programmer with different options.

Demonstration Mode: Check this box if you want to evaluate the product's user interface without in the absence of programmer hardware driven from a computer.

Start Device Programmer: click on this button launches the device programmer(s) connected to a computer with the options set in the **Program Startup Options** zone of the dialog.

Start Standalone Mode Monitor: if the programmer works in the standalone mode, click on this button launches the monitor.

The **Documentation** zone concentrates: links that invoke different types of user's guides for two device programmer models: **CPI2-B1** and **CPI2-Gx**.

Changelog link opens the Phyton ChipProg-02 Revision History file that lists most recent feature changes, newly added devices and bug fixes

Phyton Homepage links opens the www.phyton.com website in your default web browser.

The **Contact Tech Support** zone includes Phyton contact information and enable to open a new support case by clicking the **Create a ticket on the Phyton Site** link.

If the programmer was launched in the Diagnostic mode (see above) then you can send a bug report to the Phyton technical support by clicking the **Submit Bug Report** button.

The **For Developers** zone includes links to a few tools for those who develop applications for CPI2 device programmer control.

2.6   Launching device programmers

Launching a single CPI2-B1 device programmer.

A single CPI2-B1 programmers starts in the Single Programming mode. Unless a serial number of the CPI2-B1 device programmer was not specified in the **Additional Command Line Options** box of the Startup dialog, after clicking the **Start Device Programmer** button ChipProg-02 program attempts to establish communication to a CPI2-B1 programmer via a USB or Ethernet port, whatever is selected. On a very first attempt the programmer issues the **Choose a Programmer** dialog:

If a CPI2-B1 device programmer is connected via USB:
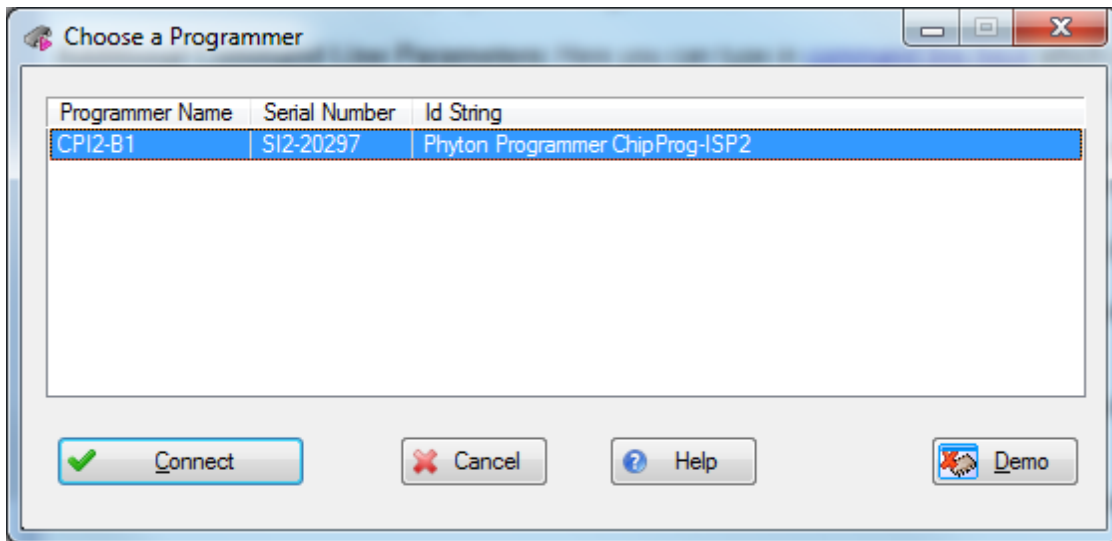
If it is connected via Ethernet:
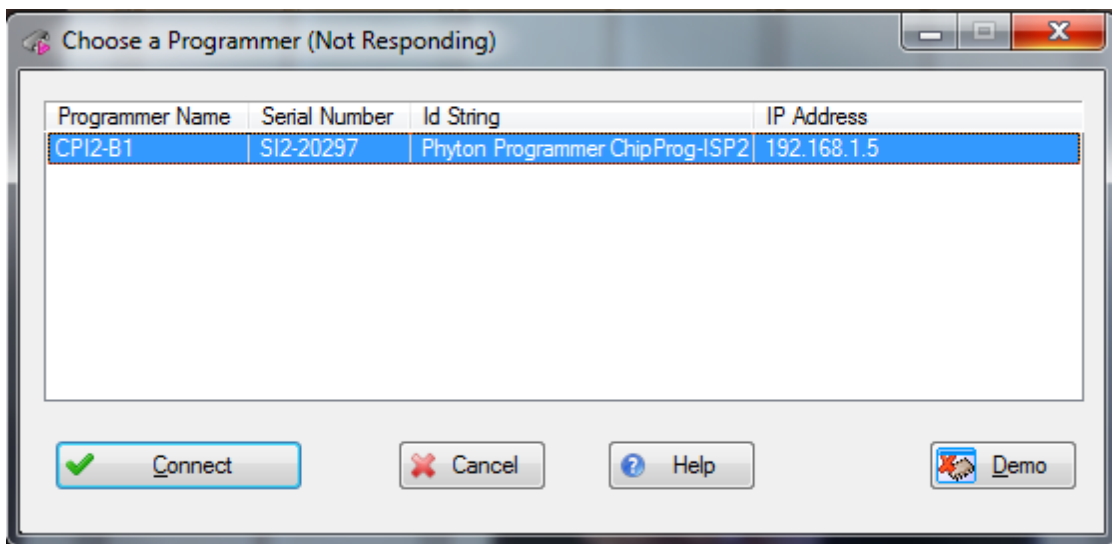
Click the **Connect** button to establish communications to the CPI2-B1 programmer and to open the ChipProg-02 main window. Later, when you launch the same CPI2-B1 device programmer the program will skip displaying the **Choose a Programmer** dialog.

Launching a cluster of multiple CPI2-B1 device programmers.

To launch ChipProg-02 program in the **gang-programming** mode, either check in the use the **Gang Mode** checkbox below the **Start** button in the Startup dialog or add the -**GANG** key to the command line.

A number of CPI2-B1 device programmers driven from one computer in gang-programming mode is limited to 72 pcs. Each single CPI2-B1 unit has its own unique serial number. Before operating with multiple CPI2-B1 programmers as a gang cluster you have to assign **Site Numbers** from 1 to N to serial numbers of the programmers' serial numbers. There are two ways to do this: a) direct specifying

a chain of CPI2-B1 serial numbers in the command line or b) manually.

Specifying site numbers in command line.

If the -**GANG** key is followed by '**#**' sign with a list of serial numbers separated by semicolons, the application waits until the number of connected single-site programmers matches the number of serial numbers in the list, then automatically assigns sequence numbers according to the serial numbers in the list. For example, if the -**GANG#SI2-10014;SI2-10022** is specified, the application waits for two programmers with serial numbers SI2-10014 and SI2-10022 to be connected. The programmer with serial number SI2-10014 will be assigned the sequence number 1 and programmer with serial number SI2-10022 will be assigned the sequence number 2.

Manual assigning site numbers.

If the -**GANG** key is not followed with CPI2-B1 serial numbers you can assign site numbers manually. Once Windows detected multiple CPI2-B1 programmers connected to a PC the ChipProg-02 opens the **Specify Site Numbers** dialog. It prompts for assignment of numbers to individual programmers (as shown in the figure for the case of three-programmer cluster). Press the **Start** button on the programmer to which you would like to assign the site #1. Then the ChipProg-02 will prompt to assign the site #2 to another programmer (in case there are more than two programmers in the programming cluster), etc.
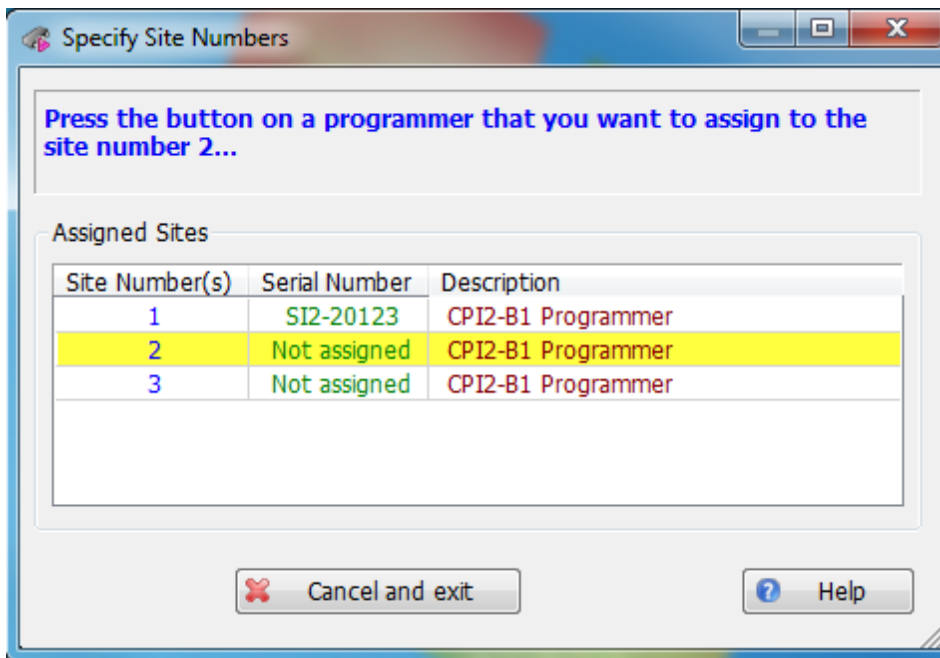
3   Control Interfaces

CPI2-B1 device programmers can be controlled by a personal computer via a Graphic User Interface (GUI), remotely from Automatic Test Equipment (ATE), In-Circuit Testers (ICT), automated handlers and other equipment connected to a PC or in Standalone mode. This chapter describes all the controls excluding the Standalone control mode.

Driving a CPI2-B1 programmer (or multiple programmers) connected to a PC is accomplished by using the friendly and intuitive **Graphic User Interface**, from a **Command Line**, or by launching **Script Files**.

Driving a CPI2-B1 programmer (or multiple programmers) remotely can be accomplished via the DLL supplied with ChipProg-02 software.

3.1   Using Projects

Using device programmer involves many operations such as choosing target device, loading a file to be programmed into the device, customizing programming algorithm, constructing a batch of commands for Auto Programming procedure, configuring the CPI2-B1 user interface, etc. These actions require working with tens of dialogs, menus and sub-menus in different ChipProg-02 windows. The ChipProg-02 program allows you to store all such settings in a single file called **project**. You can create any number of projects for programming a variety of devices and store them in the project repository. When needed, these files can be loaded and used just by a mouse-click, or by including a project name on command line. Use of projects saves time and simplifies programming process.

Projects are especially beneficial for **production** programming where a typical scenario includes replication of a lot of chips programmed with the same data but different serial numbers. In such case it is very convenient to create and lock a project that completely defines the programming session and then assign programming operation to a worker who will simply replace the chips being programmed while watching programming progress and results.

The table below lists major project options.

Option group	Project options	Where to set up...
Major properties	Project name; Description; Permissions (password, selected locking options); Files to be programmed into the device, File format, Start and end address for file loading, Destination buffers; Scripts to be preloaded; Desktop.	Menu **Project** - Options - Dialog **Project Options**
Device	Device type; Auto Detect; Insert test; Check device ID; What to do when the device insertion is detected; Device parameters (fuses, lock bits, special function registers, etc.); Programming algorithm (applicable chip sectors, voltages, oscillator frequency, etc.)	Menu **Configure** - Dialog **Select Device**; Window **Program Manager** - tab **Options** Windows **Device and Algorithm Parameters Editor**
Buffers	Buffer name; Buffer size; Default fill value; Swap file settings.	Menu **Configure** – sub menu **Buffers**; Window **Buffer** – toolbar; Dialog **Buffer Configuration**; Window **Buffer** – toolbar; Dialog **Memory Dump Windows Setup**
Serialization, Check sum, Log files	Algorithm for programming serial numbers; Custom signature patterns; Algorithm of the check sum calculation; Check sum formats; Parameters and locations of log files to be saved.	Menu **Configure** – tabs of the sub menu **Serialization, Check sum, Log files**
Actions on events	Actions triggered by certain events, issuing	Menu **Configure** – sub menu **Preferences**

Option group	Project options	Where to set up...
	error messages and sounds, logging results.	
Graphical User Interface	Screen configuration, fonts and colors of windows, key mappings, messages and miscellaneous settings.	Menu **Configure** – sub menu **Environment**
Statistics	Number of chips to be programmed and related settings.	Window **Program Manager** - tab **Statistics**

You can create, edit and save projects within the CPI2-B1 **Graphical User Interface -** read about the **Project Menu** and related dialogs. The project files have the name extension **.upp**.

Note. ChipProg-02 software does not automatically save changes to project options on exit. You must execute the **Save** or **Save as** command from the **Project** menu to save project changes made in all GUI settings dialogs since this project was opened.

3.2   Graphical User Interface

The ChipProg-02 graphical user interface (GUI) contains the following elements:

- Windows.
- Menus - global and local.
- Toolbars - global and local.
- Dialogs.
- Hot Keys.
- Context-sensitive help prompts.

The GUI features several useful additions designed specifically for the CPI2-B1 operations.

To make your using ChipProg-02 program easier we highly recommend you read the Menus and Windows chapters in full. You will be able to use the CPI2-B1 device programmers much more effectively.

3.2.1   User Interface Overview

ChipProg-02 features standard Windows interface with several useful additions.

1. Each window has its own local menu (the shortcut menu). To open this menu, click the right mouse button within the window area or press **Ctrl+Enter** or **Ctrl+F10**. Each command in the menu has a hot key shortcut assigned to a **Ctrl+<letter>** key. Pressing the hot key combination in the active window executes the corresponding command.

2. Each window has its own local toolbar. The toolbar buttons access most of the local menu commands of the window. A window toolbar buttons work only within that window. The main ChipProg-02 window has several toolbars which can be turned on or off (in the **Environment** dialog, the **Toolbar** tab).

3. Toolbar buttons feature mouse-over help: when you place the mouse cursor over a toolbar button for two seconds, a small yellow box appears nearby with a short description of the button's function.

4. To save screen space, you can hide any window title bar. To do this, use the **Properties** command in the local menu. You can identify the ChipProg-02 windows by their contents and position on the screen (and, if you wish, by color and font). When the title bar is hidden, you can move the window as if the toolbar were the title bar: place the cursor on a free space in the toolbar, press the left mouse button and drag the window to a new position.

5. You can open any number of windows of the same type. For example, you can open several **Buffer** windows.

6. Every input text field of any dialog box has a history list. ChipProg-02 saves them when you close programming session. Then a previously entered string can be picked from the history list.

7. All input text boxes in the dialogs feature automatic name completion.

8. All check boxes and radio buttons in the dialogs work in the following way: a double-click on the check box or radio button is equivalent to a single click on the box or button, followed by a click on the **OK** button. This is convenient when you need to change only one option in the dialog and then close it.

3.2.2   Toolbars

The ChipProg-02 program shows several toolbars at the top of the main window (see below).

The topmost toolbar (right under the CPI2-B1 main window title) includes the Main menu bar with drop-down submenus File, View, etc.. The second toolbar contains icons and buttons for the most frequently used commands on files and target devices (Open project, Load file, Save file... Check, Program, Verify, etc.). There is an indicator of the ChipProg-02 status (Ready, Wait, etc.). The third toolbar displays a target device selector. The fourth toolbar, which is not displayed by default, includes the built-in editor options and commands for scripts. The default toolbars can be customized. Refer also to the topics The Configure Menu, The Environment dialog, Toolbar.

NOTE. Hereafter some toolbar elements can be displayed grayed out - it means that these elements are unavailable for a particular target device or a mode of use. For example, since only one operation - Auto Programming - is available for gang programmers, the **Check, Program, Verify, Read, Erase** buttons are disabled and grayed out.

Besides the main window toolbars, windows of other types have their own local toolbars with buttons assigned to the most frequently used commands related to the window. See for example the Buffer window's toolbar below.
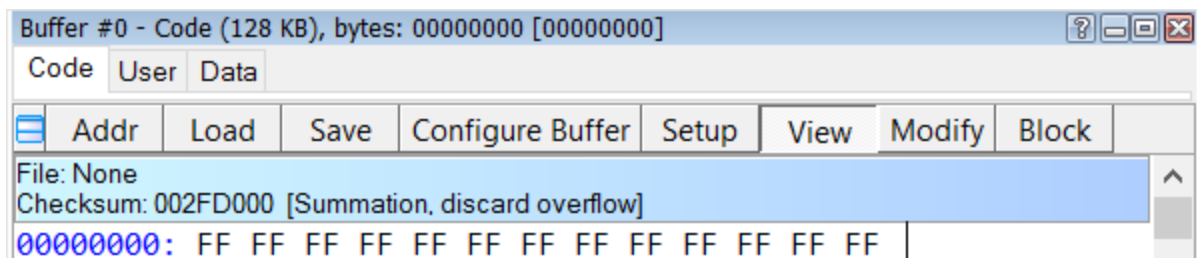
3.2.3 Menus

The ChipProg-02 Main menu bar contains the following pull-down sub-menus:

- **File menu**

- **View menu**

- **Project menu**

- **Configure menu**

- **Commands menu**

- **Scripts menu**

- **Window menu**

- **Help menu**

To access these menus, use the mouse or press **Alt+letter**, where "letter" is the underlined character in the name of the menu item.

- **Context Menus**

Each window has a context menu associated with it. To open context menu, either click the right mouse button within the window or press **Ctrl+Enter** or **Ctrl+F10**.

Most, but not all, context menu commands are also available as toolbar buttons at the top of the window.

3.2.3.1 The File Menu

File menu commands invoke file operations. For those commands that have a corresponding toolbar button, the button is shown in the first column of the table below. In case there is a shortcut key for a command, the shortcut key will be displayed to the right of the command in the menu.

Button	**Command**	**Description**
	Load ...	Opens the **Load file** dialog that specifies all the parameters of the file to be loaded and the file destination.
	Reload	Reloads the most recently loaded file.
	Save...	Saves the file from the currently active window to a disk. Opens the **Save file from buffer** dialog.
	Configuration Files	Gives access to operations with configuration files.
	Exit	Closes ChipProg-02. Alternatively, use the standard ways to close a Windows application (the **Alt+F4** or **Alt+X** keys combination).

3.2.3.1.1 Configuration Files

On exit ChipProg-02 automatically saves its configuration data in several configuration files named **UPROG.***. On start-up, configuration is restored from the most recently saved configuration files. In addition, you can save and load any of these files at any time using the **Configuration Files** command of the **File** menu. You can have several sets of configuration files for different purposes.

- The **Desktop** file stores display options and screen configuration as well as positions, dimensions, colors, and fonts of all open windows. The extension of this file is **.dsk.** The default file name is **UPROG.dsk.**

- The **Options** file stores target device type, file options, etc. The extension of this file is **.opt**. The default file name is **UPROG.opt.**

- The **Session** file stores session data and specifies the desktop and options; it can also be saved and loaded by means of the **Save session** or **Load session** subcommand of the **Configuration Files** command. The extension of this file is **.ses**. The default file name is **UPROG.ses.**

- The **History** file contains all settings entered in the text boxes of all the ChipProg-02 dialogs. This file is hidden but the settings stored earlier are available for quick selection from the History lists. The extension of this file is **.hst**. The default file name is **UPROG.hst.**

3.2.3.2   The View Menu

This menu provides a way to show various to ChipProg-02 windows.

Button	**Command**	**Description**
	Program Manager	Opens the **Program Manager** dialog.
	Device and Algorithm Parameters	Opens the **Device and Algorithm Parameters** dialog.
	Buffer Dump	Opens the **Buffer** dialog.
	Memory Card Window	Opens the **Memory Card** window
	Device Information	Opens the **Device Information** dialog.
	Console	Opens the **Console** dialog.

3.2.3.3   The Project Menu

This menu contains commands for working with projects.

Button	Command	Description
	New	Opens the **Project Options** dialog.
	Open	Opens the **Open Project** dialog for loading an existing project file.
	Close	Closes and saves current project.
	Save	Saves all settings of current project.
	Save As	Opens the **Save project** dialog. Duplicating projects under different names and/or in different folders is helpful for cloning similar projects.
	Export	Opens the Exporting Project dialog
	Import	Opens the Importing Project dialog
	Repository	Opens the **Project Repository** dialog for storing current project in project data base for convenient handling.
	Options	Opens the **Project Options** dialog for editing project options.

> **Note.** ChipProg-02 software does not automatically save changes to project options on exit. You must execute the **Save** or **Save as** command from the **Project** menu to save project changes made in all UI settings dialogs since this project was opened.

3.2.3.3.1 The Project Options Dialog

This dialog is used for setting initially and editing project options.

Control	Description
Project File Name	Specifies the project file name and path. If extension is omitted. when you close the dialog by clicking the OK button, the program saves the project file with extension **.upp**.
Permissions...	Opens the **Editing Permission Settings** dialog. Here you can protect the project file against unauthorized editing. By checking appropriate boxes in this dialog you can lock major groups of project options.
Project Description (optional)	Here you can enter your custom comments for the project.
Desktop	Two radio buttons which allow you to choose if current project will have its own desktop, or all ChipProg-02 projects will use the same desktop settings.
Files to Load to Buffers	One or more files to be loaded into the buffers upon opening the project.
Add file	Opens the **Load File** dialog for adding this file to the **Files to Load to**

Buffers.

Remove file	Remove selected file from field **Files to Load to Buffers.**
Edit file options	Opens the **Load File** dialog for editing a file highlighted in the **Files to Load to Buffers** list.
Script to execute before loading files:	Here you can enter the name of a script to be executed before loading the files to the project.
Script to execute after loading files:	Here you can enter the name of a script to be executed after loading the files to the project.

The dialog should be completed by clicking the **OK** button. Then a specified project file with the extension **.upp** will appear in a specified folder.

3.2.3.3.2 The Open Project Dialog

This dialog is used to open a previously created project.

Control	Description
Project File Name	Here you can enter full path of a project file name or browse project files. The ChipProg-02 project files have file name extension **.upp**.
Project Open History	Shows a list of previously opened projects. Double-clicking on a line in the list opens corresponding project.
Remove from list	Deletes selected project from the **Project Open History** list.

3.2.3.3.3 Export and Import Project Dialogs

The ChipProg-02 allows **exporting** and **importing** projects created for the CPI2-B1 control.

The **Export Project** dialog allows moving an entire project along with the user's data to another computer.
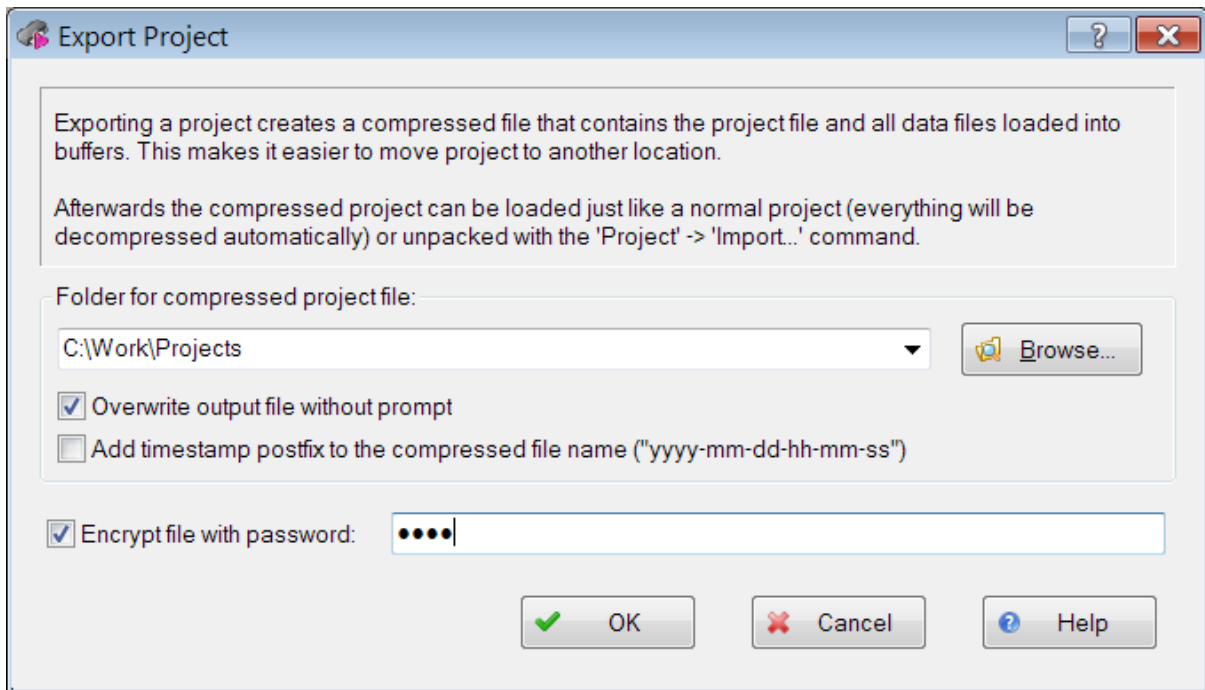
The program zips a specified the project file (for example, ABC.upp) with the data file(s) to be loaded by opening the ABC.upp project to the CPI2-B1 programmer's buffer and stores the exported compressed project into a specified folder (here C:\Work\Projects). Exported project files have the **.upc** extension - in this case the ABC.upc file. The .upc files have a standard zip format.

Checking the Overwrite output file without prompt box prevents casual spoiling of a previously stored compressed project.

Checking the Add timestamp postfix to the compressed file name enables to create a series of .upc files with the same name but made at a different time.

For security you may encrypt the .upc file. Check the **Encrypt file with password** box and type in your password in a field at right. Later, when you attempt opening or importing the project, you will be prompted to enter this password.

These exported files can be moved or copied to another PC and then can be open by the **Project > Import** command.

The **Import Project** dialog enables extracting a project exported from one computer to another.
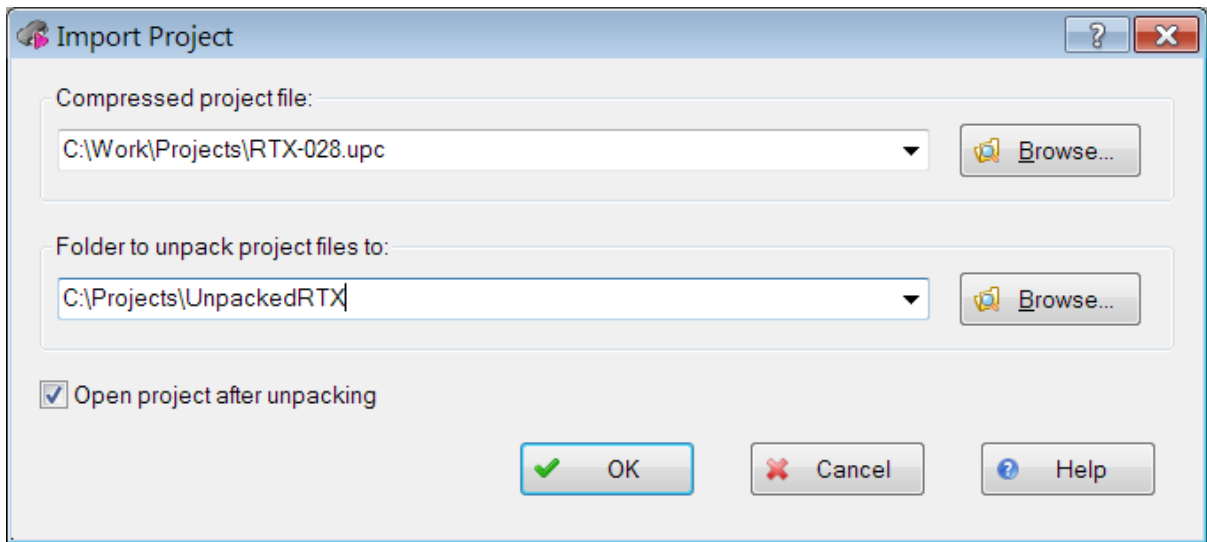
Specify an exported .upc file, a destination folder to unpack it and click OK. If the source .upc file was encrypted with a password enter it into a popped up box.

For the example above, all parts of the RTX-028.upc compressed project will be extracted into the folder UnpackedRTX, including the RTX-028.upp project file and all the data files associated with this project.

Compressed .upc files can be loaded to ChipProg-02 by the Open Project command as well as "simple" .upp project files. When you use the Open Project command from the Project menu ChipProg-02 program extracts a .upc file to a temporary folder, loads the extracted project and then deletes this temporary created folder. If the .upc file includes large data, opening the project may take quite a long time. Use of the **Import Project** function vs **Open Project** saves time because an imported project extracts to a specified folder and all extracted files remain in this folder.

Since opening a compressed .upc project completes with deleting a folder that temporary stores extracted files they cannot be stored and modified.

, .	,

,	.upc.	.

3.2.3.3.4 Project Repository

The **Project Repository** command of the Project menu opens the **Project Repository** tree.

Project Repository is a small database that stores records with links to project files. Here you can see the CPI2-B1 projects in a tree form similar to the Windows File Explorer, to logically organize projects for convenient access. Operations with the repository do not change the projects themselves - the repository works only with records about the projects (links to the project files). A tree branch may show projects and other branches. Any branch may contain different projects with the same names. Different branches may contain links to the same project.

Tree branches show each project file as a name (without a path) and a description in square brackets. The ChipProg-02 remembers state of tree branch (expanded/collapsed) and restores it next time you open the dialog.

When you install a new version of the ChipProg-02 software and copy the working environment from the previously installed version, the new version will inherit the existing project repository (the **repos.ini** file).

Dialog Control	Description
Add New Branch	Opens the **Add New Branch** dialog in which you can specify the name of a new branch.
Add a Project to Branch	Show the **Open Project** dialog to select a project to be added. Clicking the **Open** button adds the selected project to the selected branch.
Add Current Project to Branch	Adds the currently opened project to the selected branch.
Remove Project/Branch	Deletes the selected project or branch from the repository. All child branches are also deleted.
	When deleting a project from the repository, the ChipProg-02 deletes only the repository record about the project, and **does not delete the project file from disk**.
Edit Branch Name	Opens the **Edit Branch Name** dialog for the selected branch.
Move Up	Moves a selected project or branch up within the same level of hierarchy. The branch moves together with all its child branches .
Move Down	Moves the selected project or branch down within the same level of hierarchy. The branch moves together with all its child branches .
Save Repository	Writes or updates the repository to the disc file **repos.ini** in the CPI2-B1 working folder.
Browse Project Folder	Opens MS Windows Explorer with the opened folder of the selected project.
Open Project	Writes the repository to the disk file and opens a selected project.
Close	Closes the dialog. If the repository is changed, ChipProg-02 will

prompt to save it.

3.2.3.4 The Configure Menu

This menu gives access to all ChipProg-02 configuration dialogs.

Button	Command	Description
Select Device...	**Select device ...**	Opens the **Select Device** dialog.
	Device selection history	Lists previously selected devices.
	Buffers	Opens the **Buffers** dialog.
	Serialization, Checksum, Log file	Opens the **Serialization, Checksum, Log File**
	Preferences	Opens the **Preferences** dialog.
	Simplified User Interface editor	Opens the editor for setting **Simplified User Interface**
	Environment	Opens the **Environment** dialog with tabs: the **Fonts** tab, the **Colors** tab, the **Key Mappings** tab, the **Toolbar** tab and the **Misc** tab.

3.2.3.4.1 The Select Device dialog

The dialog allows specification of the device to work with; it has several groups of controls.
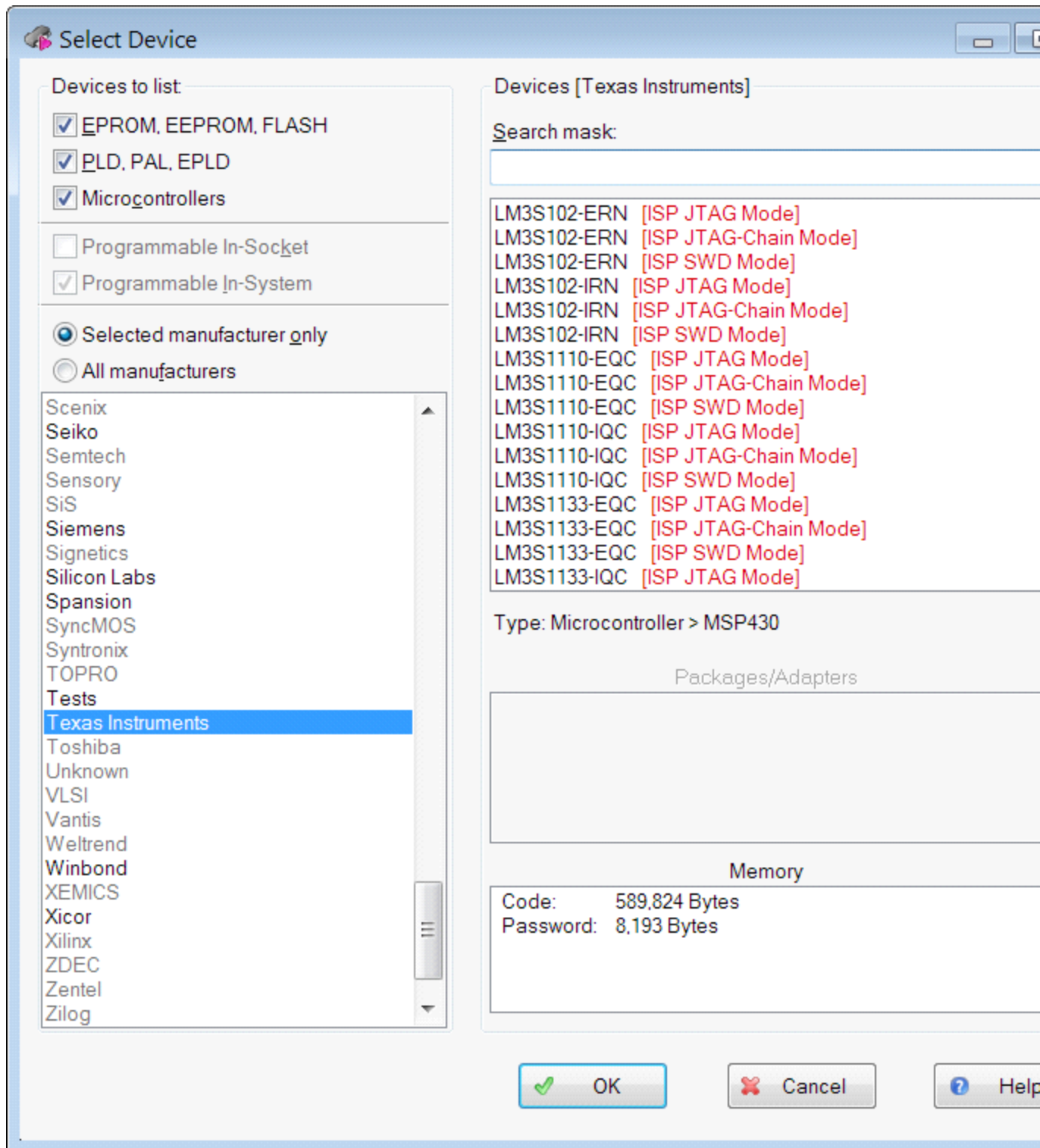
Control	Description

Devices to list:	In this field you can check one or more boxes to specify the target device type. Devices are combined into three functional groups: a) Serial memory devices; b) Programmable Logical Devices; c) Microcontrollers. Speed up the search by specifying the device properties if possible.
Manufacturer	The box lists the device manufacturers in alphabetic order.
Search mask:	Here you can enter a mask to speed up device search. The '*' character (star) represents any number of any characters in device part number. For example, the mask 'PIC18*64' will list all PIC18 devices ending in '64'.
Devices	Displays all devices by the chosen manufacturer that satisfy search criteria specified in **Devices to list, Search mask,** and **Packages/ Adapters** fields.

Sometimes you may see some devices listed in the Devices pane "greyout":
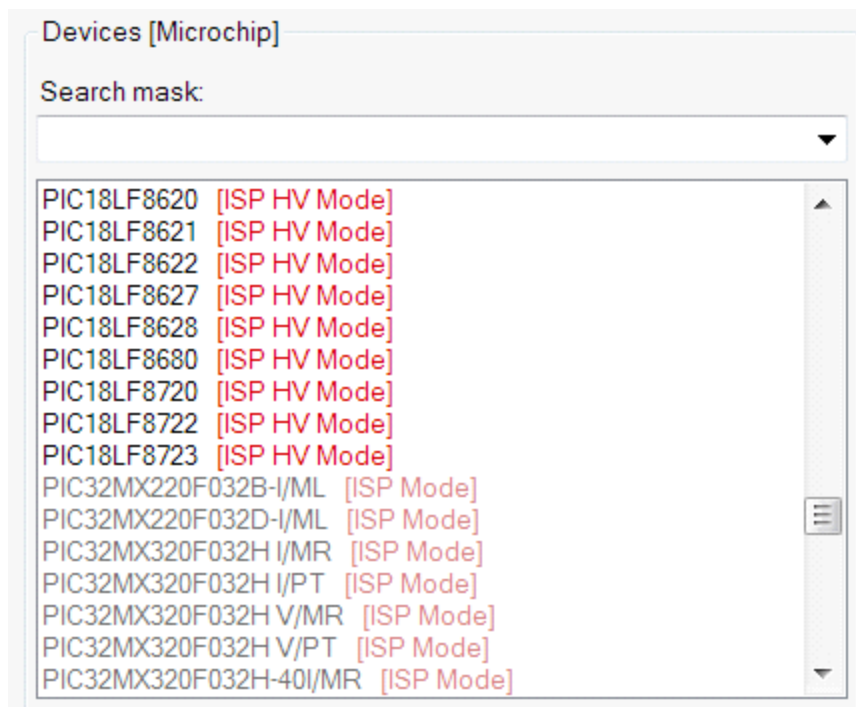
Support of "greyout" part numbers require having CPI2-D-xxxx device library licenses. After locking a certain CPI2-D-xxxx device library license all the devices covered by this license become visible and a device belonging to this library can be selected.

3.2.3.4.2 The Buffers dialog

Control	Description

Buffer list:	Displays names, sizes and sub-layers of all open buffers
Add...	Opens **Buffer Configuration** dialog to create a new buffer
Delete	Deletes the buffer highlighted in the 'Buffer list' box.
Edit...	Opens **Buffer Configuration** dialog for editing.
View	Switches focus to the window displaying the buffer highlighted in the 'Buffer list' box. If this window is hidden behind others it will be brought to the foreground.
Memory Allocation	This drop-down menu allows limiting the amount of computer RAM allocated for each buffer. The amount of free memory available for allocation is shown here in this screen area.
Swap Files	If computer's RAM is limited, the ChipProg-02 can temporarily store buffer images on PC hard drive to free some RAM. You can select the hard drive or allow the program to swap files automatically.
Use network drives	Checking this box enables swapping memory to the network drives connected to your computer.
Amount of space to leave free on each drive (GB):	Here you can reserve space on the hard drive that will never be used for file swapping.

3.2.3.4.2.1 The Buffer Configuration dialog

The **Buffer Configuration** dialog allows to setup sub-layers in buffers and to make their presentation easier to work with. To open this dialog click the **Buffer Configuration** button in the toolbar of the Buffer window.

The dialog has one tab for each sub-layer of a particular device. Every buffer has at least one main layer, so the tab 'Code' is always displayed in the dialog foreground. If selected device has other address spaces ('Data', 'User', 'ID location', etc.) the buffer will have additional sub-layers. For example: Microchip PIC16LF18875-I/PT device has two sub-layers: ID location and Data (see the picture below). Here the **Buffer Configuration** dialog has three tabs: one main for Code settings and two for ID location and Data sub-layers.

The "Buffer name, Code settings" tab contains a dialog for configuring the main buffer layer - the 'Code' layer.
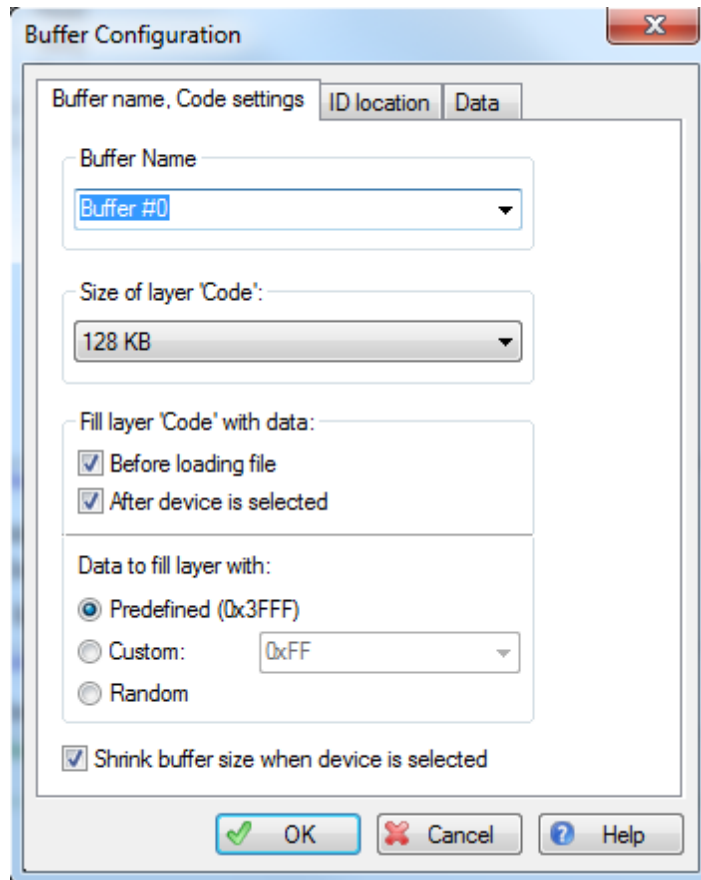
Dialog Control	Description
Buffer Name	Here you can type a name for the buffer or pick it from the history list. By default the first opened buffer gets the name "Buffer #0", the next one "Buffer #1", etc. Using this field you can give the buffer any name you wish.
Size of sub-layer 'Code'	Here you can select the size of the 'Code' layer using drop-down menu, from 128KB to 32MB.
Fill sub-layer 'Code' with data:	The program fills the buffer sub-layers with default data pattern, usually 'FF's or zeros. By checking these boxes you specify when the 'Code' layer fills with default information - before loading the file or right after device type has been chosen or both.

Leaving the "Before loading file" box unchecked enables merging multiple files in a single buffer with following programming a merged file into a target device. This, for example, can be convenient for merging code with configuration data for programming microcontrollers if the configuration file exist separately from the main code file. |

Data to fill sub-layer with:	These two radio buttons define whether the 'Code' sub-layer will be filled with default information specific for the selected device, or by a custom bit pattern or randomly.
Shrink buffer size when device is selected	Initially, buffer size usually exceeds target device 'Code' size. By checking this box you decrease buffer size to match target device layer size and to free unused PC memory.

Other tabs open appropriate dialogs which control filling the sub-layer with data similarly to filling the main (Code) layer.

3.2.3.4.3 The Serialization, Checksum, and Log Dialog

The dialog allows writing serial numbers, unique signatures, checksums and user-specified information into target device memory. It also allows to configure writing log of the process of mass production device programming.

<div style="border:1px solid green; padding:10px;">

Important!
All functions available with these dialogs: Serialization, writing in Checksums, Signatures, etc.
work ONLY when you use the Auto Programming mode for mass production.

</div>

The tabs of the dialog shown below allow manual setting of the parameters and methods of their calculation:

Serialization, Checksum, Log File

| General | Serial Number | Checksum | Signature String | Custom Shadow Areas | Log File |

Attention! All operations with Serial Number, Checksum, Signature String, Shadow Areas and Log File are performed with Auto Programming only.

Using Serialization

(●) Discard serial numbers of defective devices. In this mode serial numbers of the device yield may include gaps in the sequence of numbers written into successfully programmed devices.

( ) If a programming operation fails, discard the device but keep incrementing serial numbers (in accordance with the 'Serial Number' dialog's settings). In this mode, serial numbers of the device yield are always represented by continuously increasing sequence, i.e. without gaps.

General

Serial Number

Checksum

Signature String

Custom Shadow Areas

Log File

ChipProg-02 merges: a) the data loaded to buffers and b) special data set in the shadows areas and then writes the merged data array into the target memory device. If some addresses of the merged data overlap each other then the data taken from the shadow areas overwrite ones taken from the memory buffer and the merged data physically move to the target device memory.

3.2.3.4.3.1 Shadow Areas

The Concept of Shadow Areas

Shadow areas are special memory locations that **do not** belong to the buffer; they are located in a separate area of computer RAM. The contents of shadow areas may include individual chip serial numbers, buffer checksums, special signatures, constants, etc. Contents of a shadow area are not part of the source file loaded into a buffer; instead it can be set either manually via CPI2-B1 user interface or remotely via Application Control Interface. There are several shadow areas for each buffer layer that can be specified in this dialog - three for dedicated parameters: Serial Number, Checksum, and Signature String; plus multiple Custom Shadow Areas.

Overlapping of Shadow Areas and Buffer Data

Whenever **Program** command is executed, the ChipProg-02 merges a) data loaded into buffers, and b) special data in the shadows areas. It then writes the merged data array into target memory device. If any addresses in the merged data overlap, the data read from shadow areas overwrite the data read from memory buffer, as shown below.

—————————————————

Custom shadow area N   ?
Custom shadow area N-1 ?
Custom shadow area N-2 ?
....
....
Custom shadow area 2 ?
Custom shadow area 1 ?

—————————————————

Signature string ?

—————————————————

Checksum ?

_____

Serial Number ?

_____

Data in memory buffer

_____

Note. It is important to carefully check correctness of the addresses set in the the Serialization, Checksum and Log File dialog to prevent data corruption as a result of areas overlapping by mistake!

3.2.3.4.3.2 General settings

The tab contains a dialog to handle serialization of the devices in case a device programming fails. The two options are shown in the figure below.
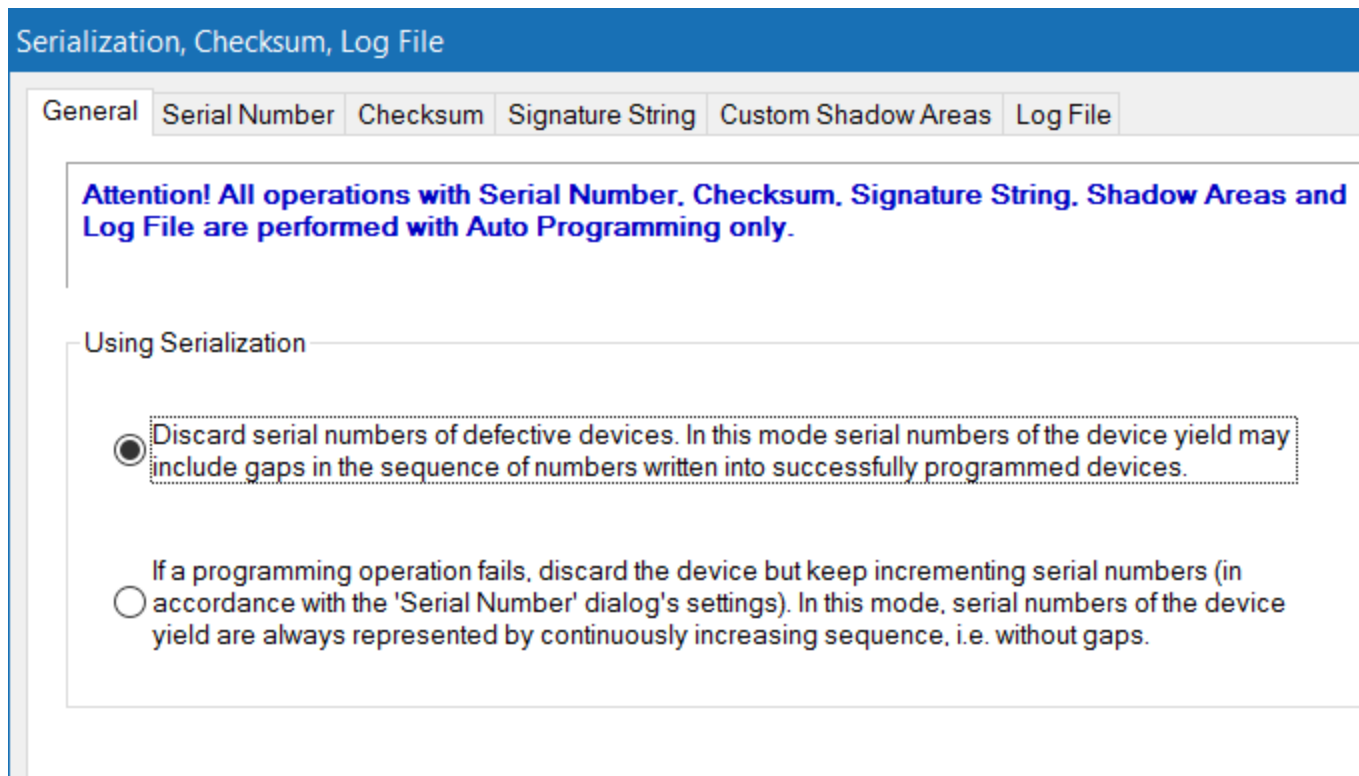
3.2.3.4.3.3 Device Serialization

The **Serial Number** tab defines a procedure of assigning a unique number to each single device from a series of devices to be programmed. By default serial number starts at 0, is incremented by 1, and occupies one byte.

Element of dialog	Description

Write S/N to address:	If this box is checked, the programmer will write a serial number into the specified address of the specified memory layer of the target device, as defined by the settings below.
Current serial number:	Use this field to specify the starting serial number. Default value is 0.
S/N size, in byte:	Specify the size of serial number in bytes; for example: 1, 2, 4, etc. Default is one byte.
Byte Order	These radio buttons define the order of bytes in the serial number (if it occupies more than one byte): either the least significant byte (LSB) follows the most significant byte (MSB) or vise versa.
Display S/N as:	These radio buttons choose the serial number display format - decimal or hexadecimal.
Increment serial number by:	By selecting this radio button you set the serial number increment as the fixed value specified here: 1, 2, 10, etc.
Use script to increment serial number:	By checking this radio button you set the increment value to the result of executing the specified script file.

3.2.3.4.3.4 Checksum

The **Checksum** tab controls automatic calculation of checksums of data in buffers and writing the checksums into the target device memory. Checksums can be calculated using a commonly used "standard" algorithm, or using a complex custom algorithm implemented in a script.

Element of dialog	Description
Write checksum to address:	If this box is checked, the programmer will write a checksum into the specified address of the specified memory layer of the target device, in accordance to the parameters below.
Address range for checksum calculation:	There are two options for setting the address range: **Auto** and **User-defined**.
Auto:	The address is defined as a full range of the selected device memory layer. This is the default.
User-defined:	Here you can specify the start and end addresses of the selected device memory layer for which the program calculates the checksum.
Use algorithm to calculate checksum:	This drop-down menu allows to select one of several available algorithms. The default is "Summation, discard overflow".

Use script to calculate checksum:	By checking this radio button you specify a script that implements custom checksum calculation.
Size of calculation result:	These radio buttons choose the size calculated checksum: one, two or four bytes.
Size of data being summed:	These radio buttons choose the size of data being summed up: one, two or four bytes.
Operation on summation result:	These radio buttons allow to perform no operation on the calculated checksum, or to negate or complement it.
Byte Order:	These two radio buttons define an order of bytes that represent the checksum - either the least significant byte (LSB) follows the most significant byte (MSB) or vice versa.
Exclude the following areas from checksum calculation:	Checking off this box allows to specify one ore more memory ranges that will be skipped by the checksum calculation algorithm. To specify a range enter its start and end addresses and click the 'Add' button.

3.2.3.4.3.5 Signature string

The tab contains settings for writing user-defined signature string into the target device. The signature may include generic data (such as the date when the device was programmed) and unique data (such as project name, operator name, etc.).

Dialog Control	Description
Write Signature String to address: **in sub-layer:**	When this box is checked, the programmer will write the specified signature into the specified address of the specified memory layer of the target device, according to parameters below.
Max. size signature string:	This field defines the maximum length of the signature string as a number of characters.
Use Signature String template:	One of two radio buttons. If checked, the string of parameters from **Template String Specifiers** drop-down menu will be programmed into the target device.
Use script to create Signature String:	This radio button selects an alternative method of composing the signature string by means of a custom script.
Template String Specifiers:	This field lists available parameters (specifiers) for inserting into the **Use Signature String template** field. Each parameter starts with the '$' symbol.

3.2.3.4.3.6 Custom Shadow Areas

The tab provides means to specify user-defined data to program into target device. User can define any number of custom shadow areas. The data can be either entered manually or created by a script.

3.2.3.4.3.7 Log file

The tab allows set up of a log or logs of the device programming.

Dialog Control	Description
Enable log file	Check this box to enable logging device programming sessions and to set log parameters below.
Separate log file for each device	Radio buttons to select whether separate logs will be written for each manufacturer or target device type, or single log will be written for all devices programmed.
File Name (Generated Automatically)	Radio buttons to select what kind of specifier will be included in the log file name: both manufacturer and device type (for example: Atmel ATSAM3S1BB-AU, Microchip PIC18F2525, etc.) or device type only (for example: ATSAM3S1BB-AU, PIC18F2525, etc.).
Folder for log file:	The field for entering the full path to the folder where log files will be created. There is also a button for path browsing.
Single log file for all device types	Check this radio button to write single log for all types of devices programmed.
File Name	The field for entering the full path to the folder where the common log file will be created. There is also a button for path browsing.
Log File Contents	Log file settings.
Gang mode: Socket #	If device is programmed in Gang (multiprogramming) mode when this box is checked, the socket number will be logged.
Date/Time	Check this box to log date and time of device programming.
Events (device type change, file names, etc.)	Check this box to log all events associated with device programming, such as target device replacement, loaded file names, etc.
Device operation	Check this box to log all events associated with device manipulations.
Detailed Device operation	Check this box to enable more detailed logging of all events associated with device manipulations.
Operation Result	Check this box to log results of programming operations.

Device #/Good devices/Bad devices	Check this box to log the total number of the devices programmed, the number of successfully programmed devices and the number of failed ones.
Serial Number	Check this box to log serial number read from the device.
Signature string	Check this box to log signature string read from the device.
Checksum	Check this box to log checksum value read from the device.
Buffer name	Check this box to log buffer name.
Programming address	Check this box to log ranges of device locations that have been programmed.
Programming options	Check this box to log all programming options.
Log File Format	A Pair of adio buttons: one selects plain text format of the log file, the other selects comma-separated text that can be imported into Microsoft Excel.
Log File Overwrite Mode	A pair of radio buttons. Checking the top one selects the mode of appending new records to a specified log file. Checking the other selects overwriting the old log each time CPI2-B1 re-starts.
Warn if size exceeds	If this box is checked, ChipProg-02 will issue a warning every time log size exceeds a user-specified value.
Immediately write log file to disk, no buffering	If this box is checked, ChipProg-02 writes log directly to hard drive without buffering it in computer RAM.

3.2.3.4.4 The Preferences Dialog

This dialog contains settings for miscellaneous options.
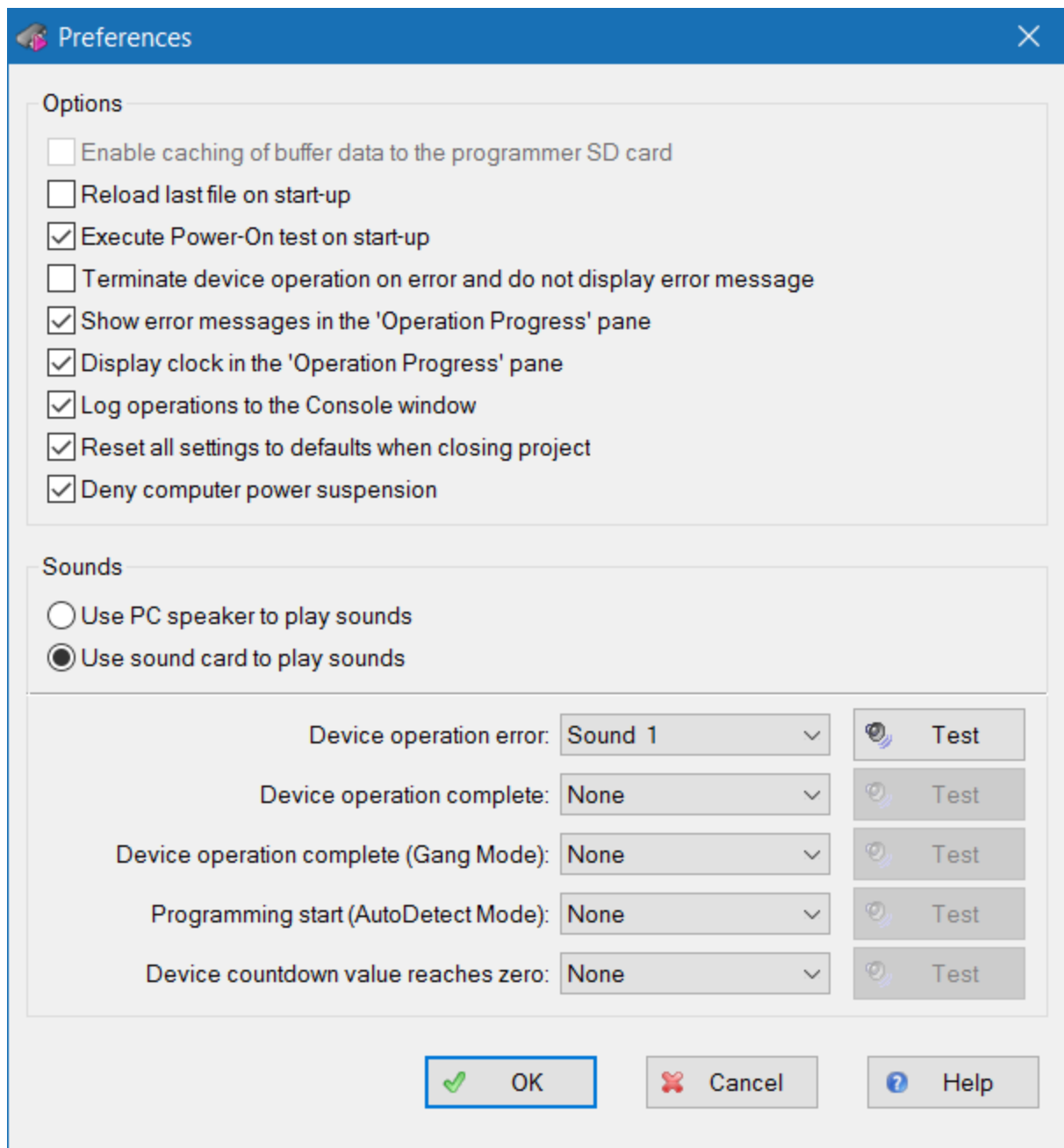
Dialog Control	Description
Options	Some (but not all) dialog options are described below.
Reload last file on start-up	Check this box to reload the last loaded file into the open buffer(s) every time you start CPI2-B1.
Execute Power-On test on start-up	This box is checked by default. Uncheck it to skip running self-test at

CPI2-B1 start-up.

Terminate device operation...	Check this box to stop programmer operations operations on any error and suppress error messages in the user interface.
Log operations in the Console window	Check this box to enable dump of programming session trace to the Console window.
Deny computer power suspension	While the programmer is not communicating with the target device, the computer may switch to the sleep mode. Check this box to prevent Windows from entering the sleep mode. This does not prevent entering sleep mode when an operator closes notebook lid or shuts down the computer by selecting Start > Shut down. This option will not disable screen saver nor prevent powering off the monitor.
	In the process of CPI2-B1 executing any command on the target device, entering sleep mode is disabled regardless of this check box status because powering off USB port may cause damage to the target device.
	If this box is unchecked, PC wake-up will cause ChipProg-02 software crashes. If a crash happens, it is necessary to cycle CPI2-B1 power and re-launch the ChipProg-02 application.
Sounds	All programmable sounds can be picked from the preset ChipProg-02 sounds
Device operation error:	Select the sound for error operations.
Device operation complete:	Select the sound for successful completion of the programming operations in a single programming mode (i.e. when one CPI2-B1 is in use).
Device operation complete (Gang Mode):	Select the sound for successful completion of the programming operations in a gang programming mode (either a few single-site programmers are connected to one PC for multi-device programming or the CPI2-B1 gang programmer is in use).
Programming start (AutoDetect Mode):	Select the sound for indicating the start of the device programming when the CPI2-B1 automatically detects insertion of a device into programming socket.

3.2.3.4.5 The Environment Dialog

The Environment dialog includes the following tabs:

Fonts tab,

Colors tab,

Mapping Hot Keys tab,

Toolbar tab**,**

Miscellaneous Settings tab**.**

3.2.3.4.5.1 Fonts

The **Fonts** tab of the **Environment** dialog provides settings for fonts and some UI elements in ChipProg-02 windows. Only monospaced (non-proportional) fonts are used to display information in windows (default is Fixedsys). To change window appearance you can select a font to be used in all windows, or in any particular window.

The **Windows** area lists the types of windows. Select a type to change its settings. The settings apply to all windows of selected type, including the windows that are already open.

Control	Description
Window Title Bar	Toggles display of title bar for windows of the selected type. If the box is checked it adds a toolbar at the position specified by the **Windows Toolbar Location** option**.** To save screen space uncheck the box. Also, see notes below.
Window Toolbar Location	Sets the toolbar location for selected window.
Grid	Toggles display of the vertical and horizontal grids in windows of certain types, and enables adjustment of column width if the vertical grid is allowed.
Additional Line Spacing	Provides additional line spacing to be added to the standard line spacing. Specify a new value or choose from the list of most recently used values.
Define Font	Opens the **Font** dialog. The selected font applies to all windows of the selected type.
Use This Font for All Windows	Applies the font of the selected window type to all ChipProg-02 windows.

Notes

1. To move a window that does not have a title bar, place the cursor on its toolbar, where there are no buttons, and then act as if the toolbar were the window title bar. Also, you can access the window control functions via its system menu by pressing the **Alt+<grey minus>** keys.

2. Each window has **Properties** item in its context menu, which can be accessed by a right click. The **Title** and **Toolbar** items of the **Properties** sub-menu toggle the title bar and toolbar on/off for the active window.

3.2.3.4.5.2 Colors

The **Colors** tab of the **Environment** dialog contains color settings for window elements such as background, font, etc. By default most colors are inherited from MS Windows; here you can set your preferred colors.

Control	Description
Color Scheme	Name of the color scheme. Your can type a name or choose a recently used one from the list.
	Save button saves the current scheme to disk; later you can restore color settings by just a mouse click. **Remove** button removes the current scheme.

Colors	Lists the names of color groups. Each group consists of several elements.
Inherit Windows Color	When this box is checked, the selected color is inherited from MS Windows color scheme. If later you change the MS Windows colors through the Windows Control Panel, this color will change accordingly. This option is available only for background and text colors.
Use Inverted Text/ Background Color	When this box is checked, the program inverts the selected window colors (for text and background). For example, if the **Watches** window background is white and the text is black, then the line with the selected variable will be highlighted with black background and white text.
Edit	Opens the **Color** dialog if the **Inherit Windows Color** and **Use Inverted Text/ Background Color** boxes are unchecked for this type of window.
	The **Color** dialog also opens with a double-click on a color in the **Colors** list.
Spread	Sets the selected color for all windows. This option is useful for text and background colors. For example, if you set yellow text on blue background for the **Source** window, and then click the **Spread** button, these colors will be set as the text and background colors for all windows.
Font	To highlight syntax in the **Source** window you can specify additional font attributes - Bold and Italic.
	In some cases when synthesizing bold fonts, MS Windows increases character size so that the font becomes unusable, because the bold and regular characters should be of the same size. In these cases, the Bold attribute is ignored.
	Sometimes this effect takes place with Fixedsys font. If you need to use Bold fonts, choose the Courier New font.

3.2.3.4.5.3 Mapping Hot Keys

The **Key Mapping** tab of the **Environment** dialog is used to assign hot keys to all ChipProg-02 commands. The **Menu Commands Tree** column displays a tree-like expandable diagram of all commands. The **Key 1** (**Key 2**) columns contain hot key combinations corresponding to commands. The actions apply to the currently selected command.

Control	Description
Define Key 1 **Define Key 2**	Opens the **Define Key** dialog. In the dialog, press the key combination you want to assign to the selected command, or press **Cancel**. Alternatively, double-click the "cell" in the row of this command and the **Key 1** (**Key 2**) column.
Erase Key 1 **Erase Key 2**	Deletes the assigned key combination for the selected command. Alternatively, right click the "cell" in the row of this command and the **Key 1** (**Key 2**) column.

3.2.3.4.5.4 Toolbar

The **Toolbar** tab of the **Environment** dialog controls display and contents of window toolbars.

Control	Description
Toolbar Bands	Lists the ChipProg-02 toolbars. To enable/disable a toolbar check/uncheck its box.
Buttons/Commands	Lists the buttons available for the toolbar selected in the **Toolbar Bands** list. To enable/disable a button on the toolbar check its box.
"Flat" Local Window Toolbars	Toggles between "flat" and 3D appearance of toolbar buttons in specifyed windows.
Toolbar Settings are the Same for Each Project/Desktop File	Applies current settings of this dialog to other projects or future opened files.

3.2.3.4.5.5 Messages

Check messages that program should display, uncheck messages that you do not want to be displayed.

3.2.3.4.5.6 Miscellaneous Settings

The **Miscellaneous** tab of the **Environment** dialog contains settings for miscellaneous properties of ChipProg-02 windows and messages.

Control	Description
Main Window Status Line	Sets visibility and location of the <%CM%> window status line.
Quick Watch Enabled	Turns Quick Watch function on or off.
Highlight Active Tabs	Toggles highlighting for the currently active tab (MS Windows-style) in windows that have tabs.
Double Click on Check Box or Radio Button in Dialogs	Makes mouse double click equivalent to single click plus pressing **OK** button in dialogs.
Show Hotkeys in Pop-up Descriptions	Toggles display of hot keys in mouse-over help for toolbar buttons.
Do not Display Box if Console Window Opened	If **Console** window is open it will show messages. Otherwise messages will be shown in message box.

Always Display Message Box	All messages will be displayed in the message box. In addition, the **Console** window will also display same messages.
Automatically Place Cursor at OK Button	When this box is checked, the cursor will always be on the **OK** button whenever message box opens.
	You can also press **Enter** key instead of using the mouse to click **OK.**
Audible Notification for Error Messages	If this is selected, error message will be accompanied with a beep. Information (as opposed to error) messages never come with a beep.
Log Messages to File	Specifies message log file name. All messages will be written to this file. Writing method depends on the radio button with two options:
Overwrite Log File After Each Start	For every session, erase previous log file if exists, and create a new one.
Append Messages to Log File	Append messages to the existing log file. In this case log file can grow without limit.

3.2.3.4.6 The Editor Otions Dialog

The ChipProg-02 software includes a **built-in Scripts Files editor**. The **Editor Options dialog** provides access to editor settings and includes the following tabs.

General Editor Settings tab,

Key Mapping tab.

3.2.3.4.6.1 The General Tab

The **General** tab of the **Editor Options** dialog has settings for common options that apply to every **Source** window.

Dialog Control	**Description**
Backspace Unindents	Toggles Backspace Unindent mode (see below).
Keep Trailing Spaces	When this box is checked, the editor does not remove trailing spaces in lines when copying text to a buffer or saving it to a disk. When the box is unchecked such spaces are removed.
Vertical Blocks	If checked, the Vertical Blocks mode is enabled for block operations.
Persistent Blocks	If checked, the Persistent Blocks mode is enabled for block operations.
Create Backup File	If checked then each time a file in the **Source window** is saved ChipProg-02 creates a back-up file (with file name extension *.BAK).
Horizontal Cursor	If checked, the cursor will have the shape of a horizontal line, similar to DOS command prompt.

CR/LF at End-of-file	If checked, a carriage return/line feed sequence will be added to the end of the file (if it does not have it already) when saving file to disk.
Syntax Highlighting	If checked, forces syntax highlighting for language elements.
Highlight Multi-line Comments	If checked, enables highlighting of multi-line comments. By default, only single-line comments are highlighted.
Auto Word/AutoWatch Pane	If checked, new **Source** windows will have **Auto Word/AutoWatch** pane at their right, and *the automatic word completion function* will be enabled.
Full Path in Window Title	If checked, the **Source** window caption bars display full path to the open file.
Empty Clipboard Before Copying	If not checked, previously kept data remains retrievable after copying to the clipboard.
Convert Keyboard Input to OEM	If checked, the **Source** window converts input characters from MS Windows character set to OEM (local) character set that corresponds to your localized version of Windows operating system. Also, see note below.
AutoSave Files Each … min	If checked, ChipProg-02 will save the file being edited every 'X' minutes. The value of 'X' can be selected from a list.
Tab Size	Sets the tabulation size for text display. Possible values are from 1 to 32. If the file being edited contains ASCII tabulation characters, they will be replaced with the number of spaces equal to this tabulation size.
Undo Count	Sets the maximum number of available undo steps (512 by default). Maximum allowed value is 10000 steps; however, larger values increase the editor's memory usage.
Automatic Word Completion	If the **Enable** box is checked, it enables the automatic word completion function. The **Scan Range** drop-down list sets the number of text lines to be scanned by the automatic word completion system.
Indenting	Toggles automatic indentation of new lines created on pressing. **Enter**.

NOTE 1. Convert Keyboard Input to OEM box only needs to be checked when adding characters to a file with OEM character encoding in the **Source** window. To only display such file correctly without modifying it, select the Terminal font for use in **Source** windows. This can be done in the **Fonts** tab of the **Environment** dialog: select **Editor** in **Windows** list and press the **Define Font** button.

NOTE 2. The **Backspace Unindents** mode establishes the editing result from pressing the **Backspace** key in the following four cases, when the cursor is positioned at the first non-space character in the line (there are several spaces between the first column of the window and the first non-space character):

	Backspace Unindent enabled	**Backspace Unindent disabled**
Insert mode	Any preceding blank spaces in the line are deleted. The rest of the line shifts left until its first character is in the first column of the window.	One space to the left of the cursor is deleted. The cursor and the rest of the line to the right of the cursor shift one position left.

Overwrite mode	The cursor moves to the first column of the window. The text in the line remains in place.	Only the cursor moves one position left. The text in the line remains in place.

3.2.3.4.6.2 The Key Mappings Tab

You can manage the list of available editor commands in the **Key Mappings** tab of the **Editor Options** dialog. You can add and delete editor commands, assign or reassign hot keys for new and built-in commands.

In the list, the left column shows command descriptions, corresponding command types are in the right column. The term *Command* refers to a built-in ChipProg-02 command; *Script NNN refers to* an added user-defined command. Two columns on the right specify hot key combinations that invoke the command, if they are defined.

Dialog Control	Description
Add	Opens the **Edit Command** dialog for adding a new command to the list and setting up the command parameters.
Delete	Removes a selected user-defined command from the list. Any attempt to remove a built-in command is ignored.
Edit	Opens the **Edit Command** dialog to change the command parameters. For built-in commands, you can only reassign the hot keys (the **Command Description** and **Script Name** boxes are not available).
Edit Script File	Opens the script source file of this command in the **Script Source** window.

Creating new commands

To create a new command, you should develop a script for it. In fact, you add this script to the editor, not the command. This means that your command is able to perform much more complex, multi-step actions than a usual editor command. Moreover, you can tailor this action for your convenience, or for a specific work task or other need. Your scripts may employ the capabilities of the script language with its entire set of built-in functions and variables, text editor functions and existing script examples.

A script source file is an ASCII file. To execute your command, the editor compiles the script source file. Note that before you can switch to using the script which you have been editing, you must first save it to the disk so that ChipProg-02 can compile it.

Script source files for new commands will reside only in the KEYCMD subdirectory of the ChipProg-02 system folder. Several script example files are available in KEYCMD. For more information about developing scripts, see **Script Files**.

This **Edit Command** dialog defines parameters for a new or existing command.

Control	Description
Command Description	Enter the command description here (optional). Text entered in this box will be displayed in the list of commands, to ease identification of the command.

Script Name	Name of the script file that implements this command.
Define Key 1 **Define Key 2**	Opens a special dialog box where you can assign two hot key combinations.

Script source files for commands will reside only in the KEYCMD subdirectory of the ChipProg-02 system folder. Enter the file name only, without the path or extension.

Notes

1. You should not specify any key combinations reserved for Windows (e.g. **Alt+–** or **Alt+Tab**).

2. We do not recommend assigning any combinations already used for commands in the **Source** window or ChipProg-02, as you'll have fewer ways to access those commands. Some examples are **Alt+F**, **Shift+F1**, **Ctrl+F7** which open application menus; pthers are local menu hot keys of the editor window.

3. You can use more than one modifier key in the keystroke combinations. For example, you can use **Ctrl +Shift+F** or **Ctrl+Alt+Shift+F** as well as **Ctrl+F** combination.

4. Hot keys for some built-in commands cannot be reassigned (e.g. cursor movement keys).

3.2.3.5 The Commands Menu

This menu items invoke main commands (a.k.a. functions) that control programming process - from Blank Check to Auto Programming, mode switches as well as some utility commands. Most

Command	Hot key	Description
Blank Check	F8	Launches the procedure of checking the target device before programming to make sure it is blank. Programming of certain memory devices does not require erasing them before re-programming. For such devices the **Blank Check** command is disabled and shown grayed out on the screen.

Program	F9	Launches the procedure of programming the target device, i.e. writes the contents of the buffer into the target device's cells.
Verify	F10	Launches the procedure of comparing the information taken from the target device with the corresponding information in the buffer.
Read	F11	Launches the procedure of reading the content of target device cells into the active buffer.
Erase	F7	Launches the procedure of erasing the target device. Some memory devices cannot be electrically erased. In this case the Erase command is disabled and shown grayed out on the screen
Auto Programming	F12	Launches the Auto Programming.
Self-Tests		Launches testing the CPI2-B1 hardware. In case of failure the diagnostic results screen will open.
Switch to Stand-Alone mode		Switches the CPI2-B1 from the computer-controlled mode to standalone operation mode.
Switch to Simplified User Interface		Hides a standard GUI and replaces it with a preset Simplified User Interface.
Local menu		Opens local menu of the active window.
Calculator		Opens Calculator dialog which performs calculator functions.

3.2.3.5.1 Calculator

The primary purpose of the embedded calculator is to evaluate expressions and to convert values from one radix to another. You can copy the calculated value to the clipboard.

Control	Description
Expression	The text field for entering an expression or a number.
Copy As	Specifies format of the result to copy to clipboard.
Signed Values	If checked the result of calculation will be interpreted and displayed as a signed value (for decimal format only).
Display Leading Zeroes	If checked, binary and hexadecimal values retain leading zeroes.
Copy	Copies result to clipboard using format set by **Copy As** radio button.
Clr	Clears the **Expression** text box.
Bs	Deletes one character (digit) to the left of the insertion point (Backspace).
0x	Inserts "0x".
>>	Shifts expression to the right by specified number of bits.
<<	Shifts expression to the left by specified number of bits.
Mod	Calculates the remainder of division by specified number.

While you are typing the expression in the **Expression** field, a drop-down list box ChipProg-02 tries to evaluate the expression and immediately display the result in different formats in the **Result** area. States of **Copy As** radio button and two check boxes in this area define format of the result.

You can assign values to program variables and SFRs by typing an expression that contains the assignment. For example, you may type SP = 66h and the value of 66h will be assigned to SP.

Examples of expressions:

```
0x1234

-126

main + 33h

(float)(*ptr + R0)

101100b & 0xF
```

3.2.3.6   The Script Menu

The Script menu contains several commands related to script files.

The ChipProg-02 contains a script language interpreter. Its purpose is automation of programming operations by mastering complex procedures involving both the device programmer and the programmer operator's actions. The ChipProg-02 supports composing and executing script files (SF). Working with scripts is describe in the **Script files** topics.

Commands in this menu are user-configurable, and the list can be expanded by adding new items (commands). To add a new item to the menu, place a script file into current folder or into the ChipProg-02 installation folder. The first non-empty line of any script file must contain three forward slashes followed by a title that will appear in the **Scripts** menu:

```
///<Menu item title>
```

When ChipProg-02 builds the **Scripts** menu, it searches the current folder and its installation folder for *.CMD files whose first line starts with '///' (please remember that '//' denotes beginning of a single-line comment) and inserts the text following '///' into the **Scripts** menu.

When you select an item from the **Scripts** menu, click the **Start** button, ChipProg-02 launches the selected script.

Button	Command	Description
	Start...	Opens the **Script Files** dialog from which you can
	New Script Source	Create a new **Script File** text.
	Open Watches window	Opens the Watches window.
	Add watch...	Adds watch to the **Watches window** .
	Editor window	Opens a list of the commands to **Compose a new**, **Open**, **Save**, **Save as, Print** a script file. of the **Editor** window.
	Text Edit	Edit a list of the commands for editing a selected **Script File**
	Example Scripts	Invokes the
	Help on this menu	

Working with scripts is describe in the **Script files** topics.

3.2.3.7   The Window Menu

This menu lets you control how the windows are arranged within the computer screen. The list of open windows is shown in the lower part of the menu. By choosing a window in from list you activate it and bring it to the foreground.

Command	Description
Tile	Arranges all windows without overlap. Makes the window sizes approximately equal.
Tile Horizontally	Arranges all windows horizontally without overlap. Makes the window sizes as close to each other as possible.
Cascade	Cascades windows.
Arrange Icons	Arranges icons of minimized windows.
Close All	Closes all windows.

3.2.3.8   The Help Menu

This menu gives access to the help system. See also, How to Get On-line Help.

Command	Description
Contents	Opens the contents of the help file.
ChipProg-02 User's Guide (PDF)	Opens complete User's Guide PDF file
ChipProg-02 Quick Start Manual	Opens Quick Start Manual PDF file
Search for Help on	Opens a dialog for searching the tool's help system for the content, index and keywords.
License Management...	Opens the dialog that displays a list of current licensed features and device libraries available for this CPI2-B1 and enabling to upgrade them.
Visit Phyton WEB site	Opens the **www.phyton.com** site in your default Internet browser.
Create problem report	If the CPI2-B1 crashs you can create a problem report and send a it to Phyton technical support. ChipProg-02 generates problem reports only when it was launched in the Diagnostic mode. In case the programmer is running in a working mode click on this menu line causes restarting it in the Diagnostic mode and then leads to sending a report to Phyton technical support.
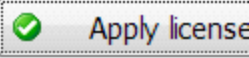
Check for updates	Opens the **Update Checking** dialog that checks whether you are running the most recent software version of ChipProg-02 and enables automatic checking with different period of time.
Phyton HelpDesk	Opens the HelpDesk web page where you can open a new ticket for Phyton technical support, track your old tickets or send a question to Phyton.
About CPI2-B1	Displays the ChipProg-02 and CPI2-B1 software versions, paths selected target device type, and device type and manufacturer, the CPI2-B1 serial number, memory card capacity and some other parameters.

3.2.3.8.1 License Management Dialog

This dialog displays a list of current licensed features and device libraries available for this CPI2-B1. It also enables adding new features and licenses.

License Management

License options on Phyton WEB site
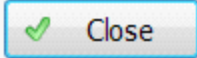
✓ Apply license

Extended Features

License	Feature	Status
CPI2-ACI	Using the Application Control Interface (ACI)	Enabled

Device Libraries

License	Device Library	Status
[00] Basic	Basic library	Enabled
[01] CPI2-D-ATCM	Microchip & Atmel Cortex device library	Enabled
[02] CPI2-D-CYCM	Cypress Cortex device library	Enabled
[03] CPI2-D-FRCM	NXP & Freescale Cortex device library	Enabled
[06] CPI2-D-ELMOS	Elmos controller library	Not licensed
[08] CPI2-D-SLCM	Silicon Labs Cortex device library	Enabled
[09] CPI2-D-STCM	ST Microelectronics Cortex device library	Enabled
[10] CPI2-D-TICM	Texas Instruments Cortex device library	Enabled
[11] CPI2-D-TOCM	Toshiba, Maxim Cortex device library	Enabled
[12] CPI2-D-ALPLD	Altera PLDs device library	Enabled
[13] CPI2-D-FR0812	NXP & Freescale HC08/S08/S12 device library	Enabled
[14] CPI2-D-TI430	Texas Instruments MSP430 device library	Enabled
[15] CPI2-D-STM8	ST Microelectronics ST7/STM8 device library	Enabled
[16] CPI2-D-RE26	Renesas RL78/RX200/RX6xxx device library	Enabled
[17] CPI2-D-UPD78	Renesas uPD78xx device library	Enabled
[18] CPI2-D-SL51	Silicon Labs EFM8/8051 device library	Enabled
[19] CPI2-D-PIC32	Microchip PIC24/32, dsPIC30/33 device library	Enabled
[20] CPI2-D-RE8C	Renesas R8C device library	Enabled

✓ Close   ❓

Click on the **License options on Phyton WEB site** link opens a page in the CPI2-B1 item catalog where you can check a list of all currently available licenses - both **Extended Features** and **Device Libraries** licenses.

The **Extended Features** pane lists the licenses that expands a set of CPI2-B1 default features. For example, the **CPI2-ACI** license enables use of the Phyton ChipProg-02 Software Development Kit (SDK), On-the-Fly Control utility and integration with NI LabVIEW software.

The **Device Libraries** pane lists Device Library licenses available at the moment of building the ChipProg-02 distributive. The **Status** column indicates the licenses physically tided up to the CPI2-B1 with a certain serial number as "Enabled" in green color, the licenses which can be added - as "Not licensed" in grey color.

If you have purchased a new license or licenses Phyton sends you some binary file. To update the license list for a CPI2-B1 with a certain serial number click the **Apply license file...** button, browse for the file on your PC, pick it and click **Open** to update the license list.

3.2.4   Windows

The following types of ChipProg-02 windows can be open from the View menu:

- **Program manager**
- **Device and Algorithm Parameters' Editor**
- **Buffer**
- **Device Information**
- **Console**

In addition there are two types of windows associated with ChipProg-02 script files:

- Editor
- Watches

3.2.4.1   The Device Information Window

This window displays the type of selected target device and a link opening a connection diagram between the TARGET connector of CPI2-B1 and a selected target device (DUT).
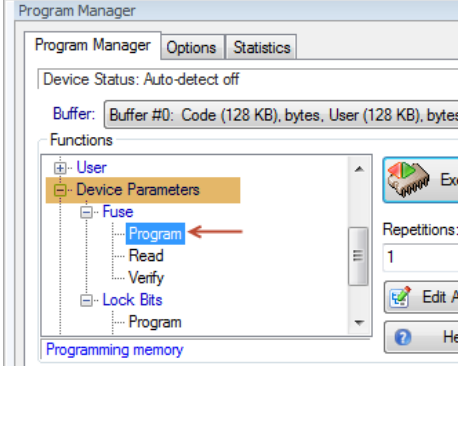
It is highly recommended to verify correctness of the CPI2-B1 - to - DUT connection before beginning your programming session either by clicking the **Connection to the target** device link in this window or on the **http://phyton.com/products/isp/chipprog-isp2-family/cpi2-b1-connecting** web page.

3.2.4.2 The Device and Algorithm Parameters Window

The **Device and Algorithm Parameters Editor** window displays and allows editing (where appropriate) target device internal parameters and settings. The edited settings must be programmed into target device by executing the **Program** command in the **Program Manager** window.

Parameters are displayed as two groups: **Device Parameters** and **Algorithm Parameters**. The groups are separated by a light blue stripe.

Device Parameters	This group includes parameters specific to each selected device, such as **sectors for flash memory devices, lock and fuse bits, configuration bits, boot blocks, start addresses** and other settings for microcontrollers. Usually these parameters represent certain bits in a microcontroller Special Function Registers (SFRs). Some of the SFRs can be set in the CPI2-B1 buffers in accordance with device manufacturer data sheets. However, setting the parameters in the **Device and Algorithms Parameters** window is more intuitive. It is impossible to specify all features that may become available in future devices; therefore not all possible parameters for new devices are described here.
	Important! Changing device parameters in the **Device and Algorithm Parameters Editor** window does not immediately result in corresponding changes inside the target device. By editing the changes you just prepare a new configuration that is different from the default for the device to be programmed. The parameters will be changed inside target device only when you execute the **Program** function in **Device Parameters** group in the **Function** pane of the **Program Manager** window as shown in the illustration.
Algorithm Parameters	This group includes parameters of the programming algorithm for the selected device – including the algorithm type and editable programming voltages. .

The window has three columns: 1) parameter name, 2) parameter value or setting, 3) a short description. **Names** of the editable parameters are shown in blue; other names are shown in black. Default values in the **Value** column are shown in black; after changing a parameter the new value will be shown in red. If the value is too long to display, it is shown as three dots ('…'). The red color of these dotst means that the parameter has been edited.

To edit a parameter double click its name. Some editable parameters are represented by a group of check boxes, others have to be typed into text fields.

Local toolbar located at the top of **Device and Algorithm Parameters Editor** window contains the following buttons:

Toolbar Button	Description
Edit	Opens a dialog to modify highlighted parameter in the format most convenient for the parameter. Double click on a highlighted parameter also opens such dialog.
Min.Value	If the parameter being modified is restricted to values from a certain

	range, clicking on the **Min.Value** button sets the highlighted parameter to the minimum allowed value.
Max.Value	If the parameter being modified is restricted to values from a certain range, clicking on the **Max.Value** button sets the highlighted parameter to the maximum allowed value.
Default	Clickin on this button returns the highlighted parameter to the default value.
All Default	Clicking on this button sets default values for all parameters in the window.

Depending on the type of a parameter ChipProg-02 offers the most convenient format for editing the parameter:

Method of Editing	Description
Drop-down menu	When a parameter value may be picked from a few preset values, the dialog shows a drop-down list of such values. Highlight a new value in the list and click OK to complete editing. For example, some microcontrollers can be programmed to work with different types of clock generators, so the menu prompts to select one of them.
Check Box dialog	When some options can be set or reset, the dialog appears in a form of several boxes showing the default or recently set option states. To toggle this behavior, check or uncheck the box. For example, some microcontrollers allow locking of particular part of memory by setting several lock bits, so the menu prompts to select lock bits represented as a set of check boxes.
Customizing the parameter	When a parameter value may be set to any value within allowed range, the dialog offers a box for entering a new value and a history list displaying a few recently set values. The dialog prompts with the min and max values and restricts entry to values in the allowed range. This type of editing is used for custom values of Vcc and Vpp voltages.

3.2.4.3   The Buffer Dump Window

The **Buffer Dump** window is used to display contents of memory buffer.

CPI2-B1 provides flexible buffer management:

- You can create an unlimited number of buffers. The number of buffers that can be created is limited only by the available computer RAM.

- Every buffer has a certain number of sub-levels depending on the type of target device. Each sub-level is associated with a specific section of the target device address space. For

example, for the Microchip PIC16F84 microcontroller, every buffer has three sub-levels: 1) code memory; 2) EEPROM data memory; 3) user identification.

This flexible structure facilitates manipulating with several data arrays mapped to different buffers. To open a **Buffer Dump** window, select **Main Menu > View > Buffer Dump**..
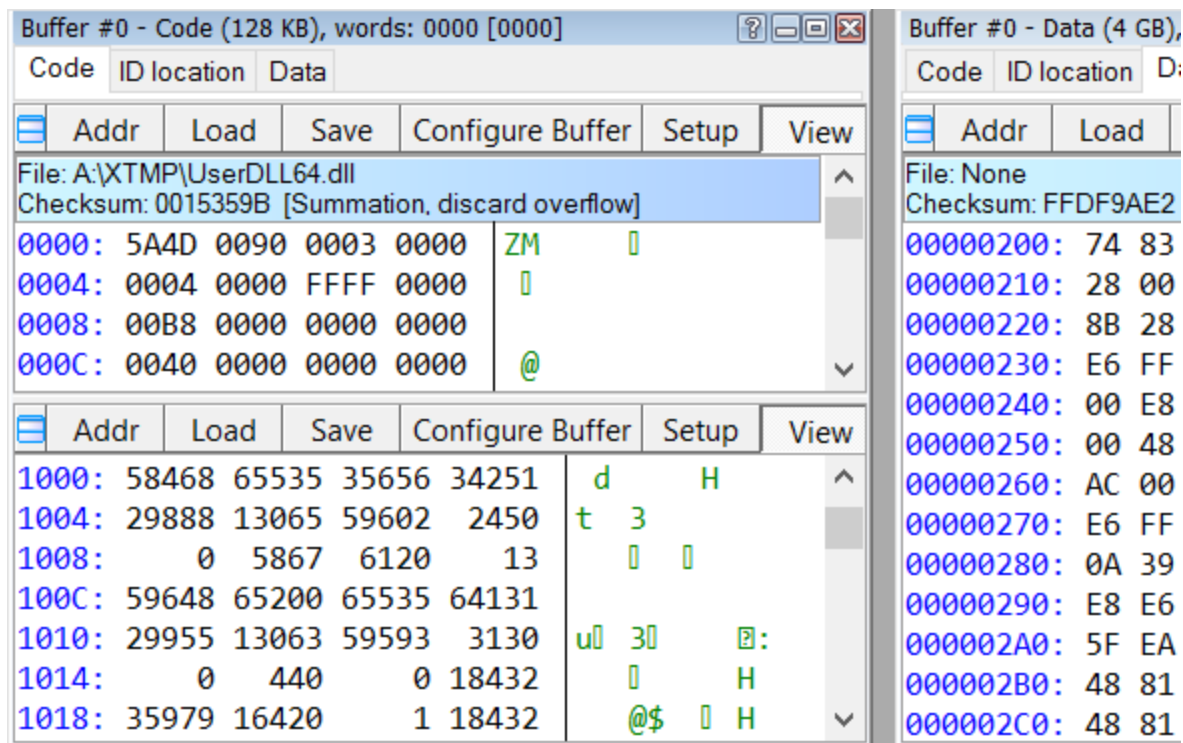
The figure above shows three **Buffer Dump** windows representing three parts of the same buffer:

- Window #1 (the largest) shows buffer contents starting at address 0h.
- Window #2 shows the same buffer contents starting at the same address, displaying data in decimal format.
- Window #3 shows the data starting at address 200h.

The leftmost column of the above windows shows absolute address of the first cell in each row. The addresses always increment by one byte: 0, 1, 2…. Each address is followed by a colon (:). When you resize a window, the addresses shown in the address column automatically change in accordance with the number of data items in each line. Some windows may be split into two panes – the left pane showing data in a selected format, and the right pane showing the same data in ASCII format.

The window has a toolbar for invoking settings dialogs and commands. Full path to the loaded file and checksum of the dump are displayed beneath the toolbar.

Local Menu and Toolbar

The context-sensitive menu brought up by a right mouse click is used to invoke context commands and dialogs of the **Buffer Dump** window. Most, but not all, local menu entries are duplicated by local toolbar buttons at the top of the window. Following are local menu and toolbar items:

Menu Item	Toolbar button	Description
New address...	Addr	Opens the **Display from Address** dialog.
Load file to buffer...	Load	Opens the **Load Window Dump** dialog.
Save data to file...	Save	Opens the **Save Window Dump** dialog.
Configure buffer...	Configure buffer	Opens the **Configuration Window Dump** dialog.
Window setup...	Setup	Opens the **Window Dump Setup** dialog.
View only, edit disabled	View	Editing in the buffer dump windows is disabled by default, so you can only view the data. If this box is unchecked editing will be enabled and you will be able to modify value under the cursor.
Modify data	Modify	Opens the **Modify Data** dialog. This is only enabled when the **View only, edit disabled** is unchecked.
Operations with memory blocks	Block	Opens the **Operations with Memory Blocks** dialog.
Swap fields	No button	Moves the cursor between right and left window panes.

3.2.4.3.1 The 'Configuring a Buffer' dialog

The dialog allows to configure buffer dumps using the most convenient way, and name or rename open buffers. By default, the first opened buffer is named 'Buffer #0', the next buffer is named 'Buffer #1', and so on. You can, however, rename buffers to your liking.

Buffer Configuration ✕

Buffer name, Code settings | ID location | Data

Buffer Name

Buffer #0 ▾

Size of layer 'Code':

128 KB ▾

Fill layer 'Code' with data:

☑ Before loading file
☑ After device is selected

Data to fill layer with:

⦿ Predefined (0x3FFF)
◯ Custom: 0xFF ▾
◯ Random

☑ Shrink buffer size when device is selected

✔ OK   ✖ Cancel   ❓ Help
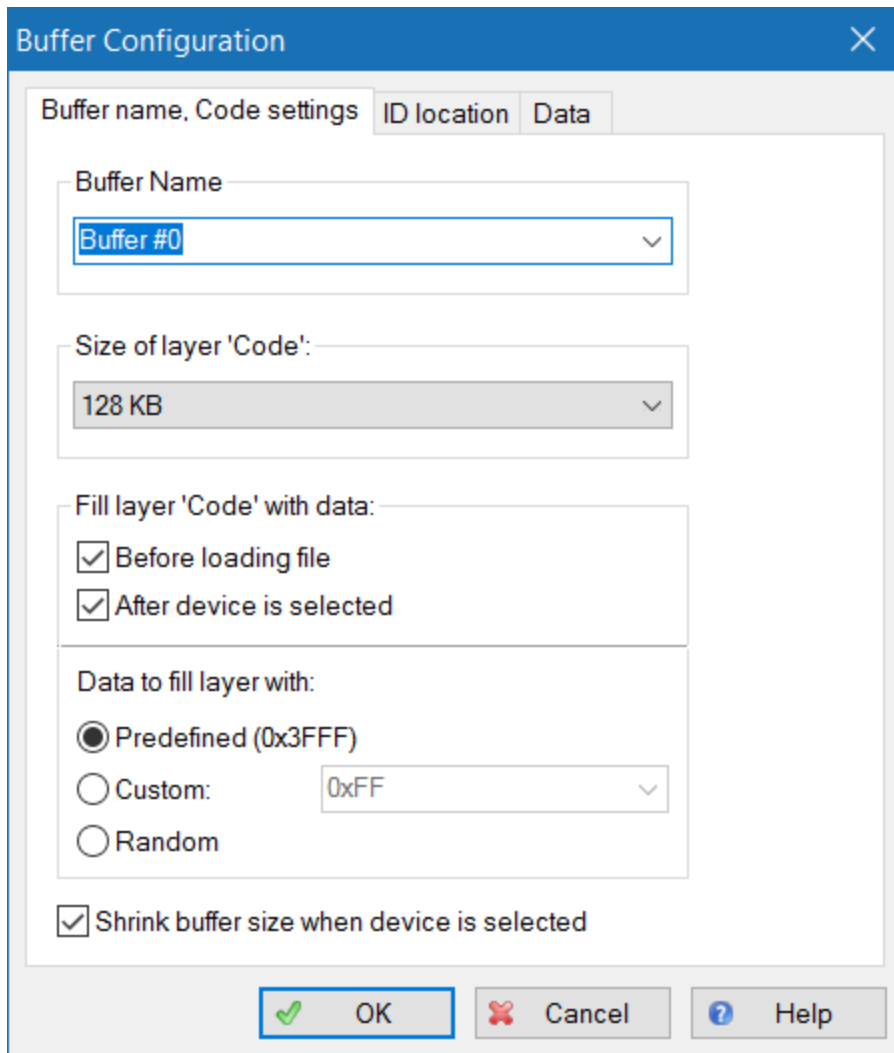
Initially each buffer is allocated a minimum of 128K of PC RAM and the ChipProg-02 program fills the buffer with a predefined pattern (usually 0FFh). You can customize these buffer settings - check the Custom radio button and type in the pattern to be used to fill the buffer..

By default ChipProg-02 program fills the buffer sub-layers with default data pattern, usually 'FF's or zeros. By checking these boxes you specify when the 'Code' layer fills with default information - before loading the file or right after device type has been chosen or both.

Leaving the "Before loading file" box unchecked enables merging multiple files in a single buffer with following programming a merged file into a target device. This, for example, can be convenient for merging code with configuration data for programming microcontrollers if the configuration file exist separately from the main code file.

3.2.4.3.2 The 'Buffer Setup' dialog

The dialog allows controlling the data presentation in the **Buffer Dump** window. You can open the dialog using the **Windows Setup** command of the local menu or by clicking the **Setup** button on the local

toolbar.

Control	Description
Buffer:	Displays a list of all open buffers. Programming functions will be applied to the active one.
Display Format	Three radio buttons select the format for the data displaye: binary, decimal or hexadecimal.
Display Data As:	Four radio buttons select the format of data presentation in the buffer: 1, 2, 3 or 4 bytes.
Options	Options to customize display format.
ASCII pane	If checked, the right pane will display ASCII characters corresponding to the data in the buffer dump.
Display checksum	If checked, calculated checksum will be displayed in the blue strip over the data dump, beneath the local toolbar.
Limit dump to sub-layer size	If checked, dump window will display part of the memory whose size is equal to the size of the active sub-layer.
Signed decimal and hex values	If checked, the most significant bit (MSB) of the data shown in binary or hexadecimal formats will be treated as a sign. If MSB=1 the data is negative, if MSB=0 they are positive.
Always display '+' or '-'	This is a sub-setting for the **Signed decimal and hex values** option. If both boxes are checked then the signs '+' and '-' will be displayed.
Leading zeroes for decimal numbers	If checked, data in decimal format will be shown with leading zeros; for example, 256 will be shown as 00000256.
Reverse bytes in words (LSB first)	If checked, the order of bytes in words will be reversed so that the MSB follows the LSB.
Reverse words in dwords	If checked, the order of 16-bit words in 32-bit words will be reversed.
Reverse dwords in qwords	If checked, the order of 32-bit words in 64-bit words will be reversed.
Non-printable ASCII characters	Characters in the range **0 00...0 20** and **0 80...0 FF** are non-printable. Following options customize display of non-printable ASCII characters in the ASCII pane of the buffer dump window.
Replace characters 0 00...0 20	If checked, all characters in the range **0 00...0 20** will be replaced with the dot ('.') or space (' '). Pair of radio buttons **Replace with** selects the replacement character: dot ('.') or space (' ').

Replace characters 0 80...0 FF	If checked, all characters in the range **0 80...0 FF** will be replaced with dot ('.') or space (' '). A pair of radio buttons **Replace with** selects the replacement character: dot ('.') or space (' ').

3.2.4.3.3 The 'Display from address' dialog

The dialog allows to set a new starting address for the visible part of the **Buffer Dump** window.

Element of dialog	Description
Type new address to display from:	Here you may enter any address within valid range.
History	Displays a list of previously entered addresses. You can pick one to set as starting address for the buffer dump.

3.2.4.3.4 The 'Modify Data' dialog

The dialog allows to edit data in the **Buffer Dump** window. The dialog can be invoked only when the **View** toolbar button if off, otherwise editing is disabled. To modify a data item in the buffer move cursor to its location and click the **Modify** toolbar button. You will be able to enter a new data value in the pop-up box or pick one from the history list. Alternatively, select a location by moving cursor to it and enter new value using the PC keyboard.

3.2.4.3.5 The 'Memory Blocks' dialog

The ChipProg-02 program supports complex operations with memory blocks. This dialog controls operations with blocks of data within a selected buffer or between different buffers.
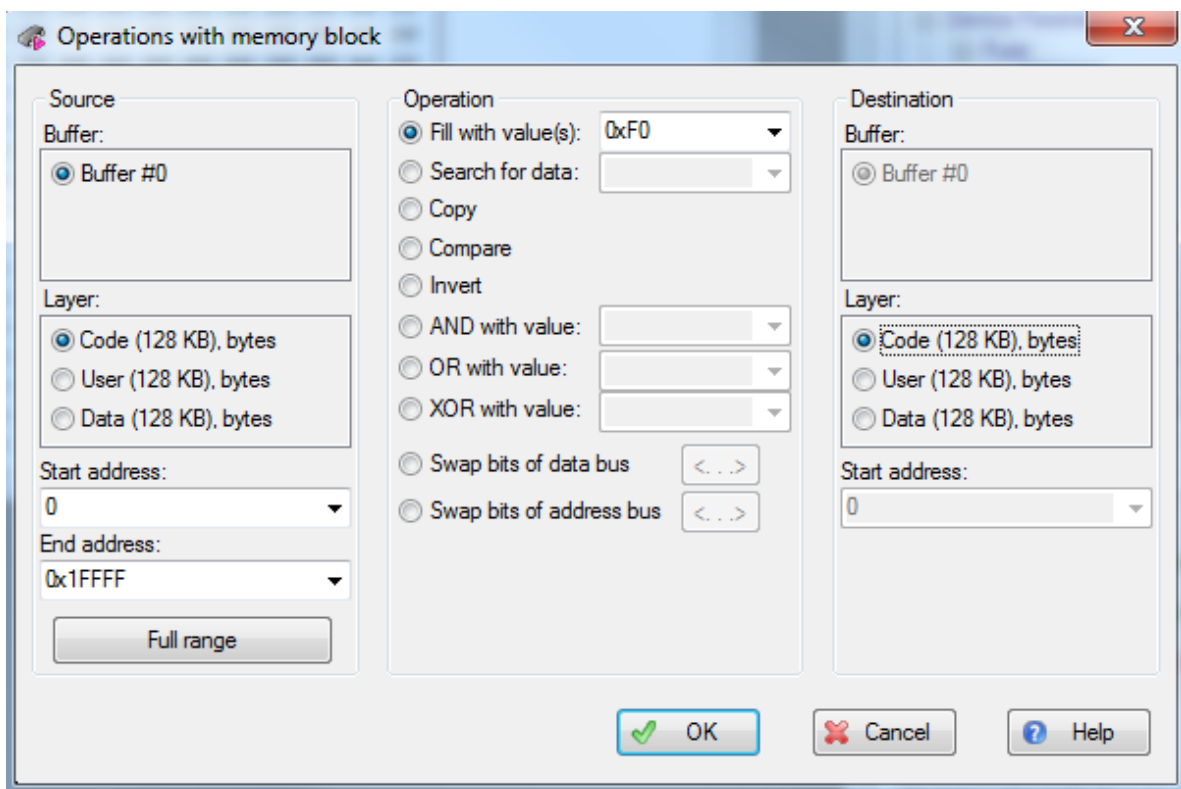
The dialog has three columns. **Source**, the left column, describes the source memory area used in operations described in the middle column. Operation result will be placed in the area described by **Destination**, the right column. By default, destination is same as source. Two operations – Fill and Search – do not require destination; if any of these two operations is chosen, **Destination** radio button will be disabled.

Control	Description
Start Address (of the Source)	Starting address of the memory area in the selected **Source buffer** to which the operation will be applied.
End Address (of the Source)	Ending address of the memory area. Ending address can be entered for the **Source** area only. Once the source address range is defined, program automatically calculates destination area ending address.
Full Range (of the Source)	Sets the starting and ending addresses to span entire address space of selected target device.
Start Address (of the Destination)	Starting address of the memory area in the **Destination buffer** where the result of the selected **Operation** will be stored.

The following operations are available via this dialog. Operation starts when you click **OK** in the dialog box (see notes below).

Operation	Description
Fill with Value	Fills the source buffer with a value (or a sequence of values) specified in the text box at the right.

Search for Data	Searches the source memory area for a particular value (or a sequence of values) specified in the text box at the right.
Copy	Copies contents of the source area to the destination address. A block can be copied within the same address space or to another one.
Compare	Compares contents of source and destination memory areas. The sizes of source and destination areas are equal. If there is a mismatch, a mismatch message box will request permission to continue the comparison process.
Invert	Inverts contents of the source area bit-wise and stores the result in the destination area.
Calculate Checksum	Calculates a 32-bit checksum for the source area. The calculation is done by simple addition. See note below.
Negate Result	If checked, the 32-bit checksum will be subtracted from zero (this is a widely used method of checksum calculation).
Write Result to Destination	If checked, the 32-bit checksum will be written to the destination sub-level at destination **Start Address**. If this box is cleared, the checksum wil onlyl be displayed in a message.
AND with Value	Performs bit-wise AND operation on the contents of the source memory area using operand specified in the text box on the right. The result is stored in the destination area. See notes below.
OR with Value	Performs bit-wise OR operation on the contents of the source memory area using operand specified in the text box on the right. The result is stored in the destination area.
XOR with Value	Performs bit-wise XOR operation on the contents of the source memory area using operand specified in the text box on the right. The result is stored in the destination area.

Notes

1. Source and destination memory areas may overlap; since operations on memory blocks are carried out using a temporary intermediate buffer, the overlap does not cause corruption of results.

2. The **Copy** and **Compare** commands use blocks specified in the **Source** address space and the **Destination** address space.

3. The checksum is calculated as a 32-bit value by simple addition. If a memory space has byte organization, then 8-bit values will be added. If it has word organization then 16-bit values will be added.

4. Logical operations (AND, OR, XOR) are performed on the contents of the **Source** address space, while the operation result is written to the **Destination** address space. The program automatically converts the operands to the word size of the selected type of memory (16-bit for **Prog, Data16, Reg** and **Stack** memory, 8-bit for **Data8** memory).

3.2.4.3.6 The 'Load File' dialog

The dialog defines how a file is loaded into the buffer.

Control	Description
File Name:	Enter a full path to the file in this box, pick the file name from a drop-down menu list, or browse files on your computer or network.
File Format:	Select **format** of the file to be loaded by checking one of the radio buttons in the **File Format** field of the dialog.
Buffer to load file to:	Select buffer to load the file into, by checking one of the **Buffer#** radio buttons. There may be just one such button.
Layer to load file to:	The **Buffer to load file to** can have more than one memory layer. Select the layer into which the file will be loaded by checking one of the radio buttons. There may be just a single button available for selection.
Start address for binary image:	Files in **Binary** format do not carry any address information. When loading binary files you have to specify the starting address for loading. In case the file to be loaded is a binary image enter starting address in the box here.
Offset for loading address:	Files in formats other than **Binary** may carry information about the starting address for the loading. If the file to be loaded is not a binary image, enter the offset for the file addresses in the box here. The offset can be positive or negative.

3.2.4.3.6.1 File Formats

The ChipProg-02 program supports a variety of file formats that can be loaded to the CPI2-B1 buffers.

File Format	Description
Standard/Extended Intel HEX (*.hex)	The Intel HEX file is a text file, each string of which includes the starting address to load the data to the buffer, the data to load, line checksums, and some additional information. The ChipProg-02 loader supports both Standard and Extended Intel HEX format.
Binary image (*.bin)	Binary image contains only data. These data will be loaded to the buffer beginning with the specified starting address.
Motorola S-record (*.hex, *.s, *.mot)	The Motorola S-record is a text file, each line of which includes starting address to load the data into buffer, the data to load, line checksums, and some additional information. The ChipProg-02 loader supports all kinds of the Motorola S-records with filename extensions .hex, .s, .mot.
Altera POF (*.pof)	The Altera POF-file is a text file, each line of which includes starting

address to load the data into buffer, the data to load, line checksums, and some additional information. The format is mostly used for programming PALs and PLDs.

JEDEC (*.jed)

This format is used for programming PALs and PLDs. A JEDEC-file includes starting address to load the data into the buffer, the data to load, test-vectors, and some additional information.

Xilinx PRG (*.prg)

The Xilinx PRG-file is a text file, each line of which includes starting address to load the data into buffer, the data to load, line checksums, and some additional information. The format is used for programming the Xilinx PLDs.

Holtek OTR (*.otp)

This format is presented by Holtek company. An OTP-file includes the starting address to load the data into the buffer, the data to load, line checksums, and some additional information.

Angstrem SAV (*.sav)

This format is presented by Angstrem company. A SAV-file includes the starting address to load the data into the buffer, the data to load, line checksums, and some additional information.

ASCII Hex (*.txt)

The ASCII TXT-file includes the starting address to load the data into the buffer, the data to load, line checksums, and some additional information.

3.2.4.3.7 The 'Save File' dialog

The dialog defines how the buffer is to be saved to a file.

Control	Description
File Name:	Enter a full path to the file in this box, pick the file name from a drop-down menu list, or browse files on your computer or network.
Addresses	**Start and End Addresses** define buffer address range that will be written to the **File**. To save entire buffer click the **All** button.
File Format:	Selected **format** of the file to be written by checking one of the radio buttons in the **File Format** field of the dialog.
Buffer to save file from:	Select the source buffer to write into the file by checking one of the **Buffer#** radio buttons. There may be just one such button available.
Sub-level to save file from:	The **Buffer to save file from** can have more than one memory layer. Select the source layer by checking one of the radio buttons. There may be just one such button available.

3.2.4.4 The Console Window

The Console window displays messages generated by the ChipProg-02 program. These messages fall into two categories: the CPI2-B1 error messages and what-to-do prompts. The window accumulates messages even when it is closed. You can open it at any time to view the last 256 messages, and get help for any of them. Error messages are shown in red color, others in black.

The window should be large enough to see several messages. To save screen space you can close the **Console** window and redirect all messages to pop-up message boxes. To do this, go to the **Configure menu** > **Environment** > **Misc** tab and select the **Always Display Message Box** option. Alternatively, you can select the **Do not open box if Console window opened** option, redirecting all messages to **Console** window.

Click the **Help** button in the box to show the CPI2-B1 context-sensitive **Help** topic associated with the error, or click the **Close** button and continue after correcting a parameter error.

Local Menu and Toolbar

The local menu contains **Console** window context commands and dialog calls. This menu can be opened by a right mouse click in the window. Most, but not all, local menu commands are duplicated as local toolbar buttons at the top of the window. Following are the local menu and toolbar commands:

Menu Command	Toolbar Button	Description
Clear Window	**Clear**	Deletes all messages from the window
Help on message	**MHelp**	Opens context-sensitive **Help** topic associated with the error or information in the highlighted message
Help on window	No button	Opens the **Console** window **Help** topic
Help on word under cursor	No button	Opens the context-sensitive **Help** topic associated with the word under cursor

3.2.4.5 The Program Manager Window

The **Program Manager window** is the primary screen object used by an operator to control the CPI2-B1 in the GUI mode. While some windows can be closed during programming operation, the Program Manager is supposed to be always open and visible. The window includes three tabs:

The Program Manager tab - by default this tab is open (see below)

The Options tab

The Statistics tab

The contents of the **Project Manager** and **Options** tabs depend on the CPI2-B1 programmers working in single-programming and gang-programming modes. Below you can see the window appearance for a CPI2-B1 device programmer operating in the Single Programming mode

3.2.4.5.1 The Program Manager tab

This tab serves for setting major programming parameters, carrying out programming operations and displaying the CPI2-B1 status.

Control	Description
Buffer:	Displays the active buffer to which the programming operations (functions) will be applied. A full list of open buffers is available here via the drop-down menu.
Functions	Shows a tree of functions available for the selected target device. Some functions represent CPI2-B1 commands while others group several sub-functions and can be expanded or collapsed. Double-clicking on a function invokes the command and is equivalent to single-clicking the **Execute** button (see below).
Blank check	Checks if the target device is blank
Program	Programs the target device (physically writes the information from active buffer to the target device).
Read	Reads contents of the target device into active buffer.
Verify	Compares contents of the target device with contents of active buffer.
Auto Programming	Executes a preset sequence of operations (batch operations). The sequence can be defined using the Auto Programming dialog. The **Edit Auto** button opens this dialog.
Addresses	Here you can set the addresses for the buffer and the target device to which the programming functions will be applied.
Device start:	Starting address of the target device physical memory which will be programmed or read.
Device end:	Ending address of the target device physical memory which will be programmed or read.
Buffer start:	Starting address of the buffer memory from which the data will be written to the target device or to which the data will be read from the device.
Execute	There are three alternative ways to activate a highlighted function: a) to click the **Execute** button; b) to double click on the function line; c) to press **Enter** button on PC keyboard.
Repetitions:	Any function can be executed repeatedly. The number of repetitions can be set here.
Edit Auto	Clicking on this button opens the Auto Programming dialog.
Operation Progress	Displays progress bar and the status (OK, failed, etc.) of current operation.

Besides generic functions such as **Blank Check, Read, Verify, Program, Auto Programming**, the **Functions** window often includes collapsed submenu of functions specific to the selected target device. When expanded it shows a list of commands for the parameters that can be set in the **Device and Algorithm Parameters** editor window.

IMPORTANT NOTE!

Any changes made in the 'Device and Algorithm Parameters' window do not *immediately* cause corresponding changes in the target device. Parameter settings made within this window just prepare a configuration of the device to be programmed. Physically, the programmer makes all these changes only upon executing an appropriate command from the 'Program Manager' window.

3.2.4.5.1.1 Auto Programming

Each device has its own typical set of programming operations that usually includes: **Erasing, Blank Checking, Programming, Verifying** and often **Protecting** against unauthorized reading. The ChipProg-02 stores default batches of these programming operations for each supported device type. A batch can be executed by a simple mouse click or pressing the Start button on the programmer panel. A sequence of functions (operations) can be customized via the **Auto Programming** dialog. To open this dialog click on the **Edit Auto** button.
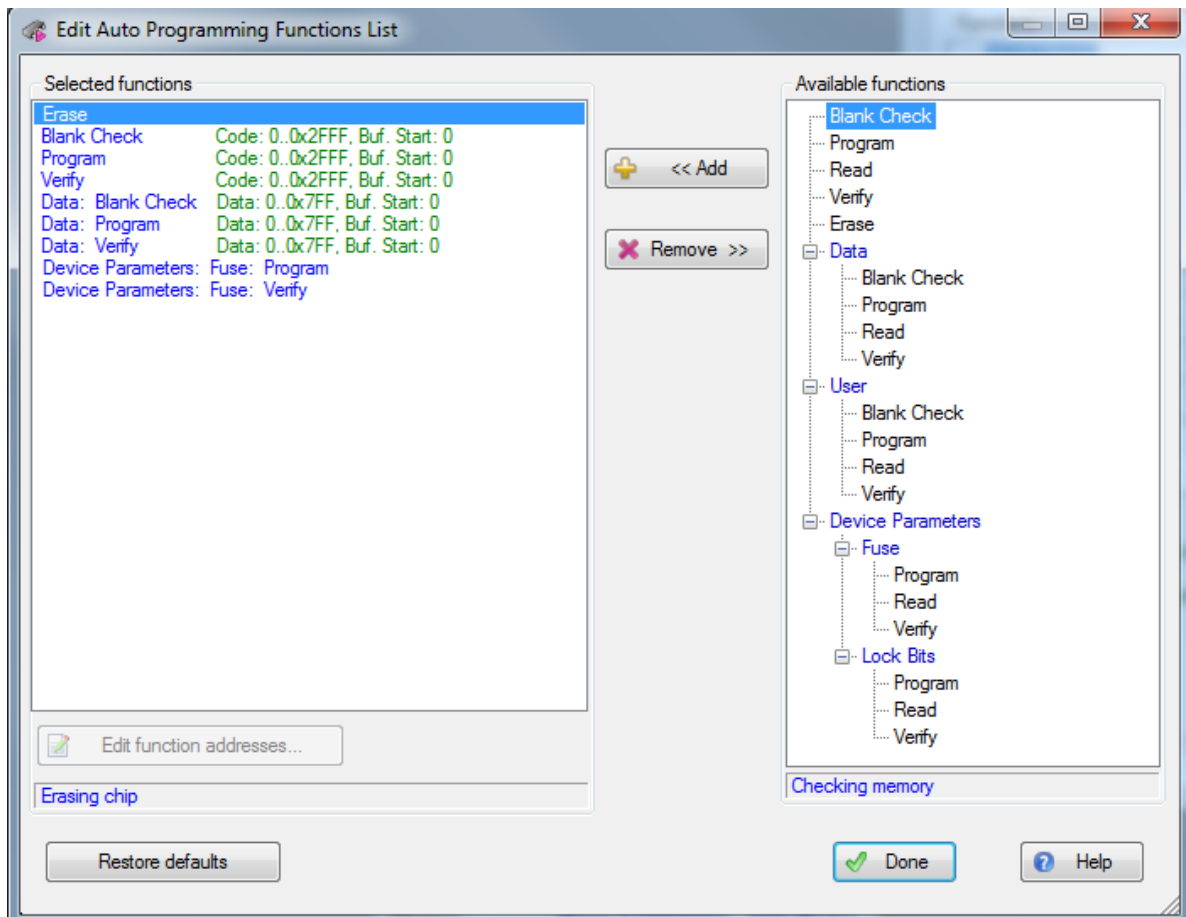
A tree of all functions available for the selected device is shown in the right pane, **Available functions**. To add a function to the batch highlight it in the right pane and click the **Add** button - the function will appear in the left pane, **Selected functions.** The functions will be executed in the order in which they are listed in the **Selected functions** pane, starting from the top. To edit a batch highlight the command to be removed and click the **Remove** button.

3.2.4.5.2 The Options tab

This tab contains controls for setting additional programming parameters and options:

Control	Description
Split data	Radio buttons in the **Split data** group control programming of 8-bit memory devices to be used in microprocessor systems with 16- and 32-bit address and data buses. In such cases buffer contents have to be properly prepared in order to split single memory file into several smaller files.
Options	
Check device ID	This option is on by default, and the CPI2-B1 always verifies target device identifier assigned by device manufacturer. If this box is unchecked the program will not check device ID.
Reverse bytes order	If checked, the ChipProg-02 will reverse byte order in 16-bit words while it executes **Read**, **Program,** and **Verify** operations. This option does not affect data in CPI2-B1 buffers.
Blank check before program	If checked, the ChipProg-02 will make sure the target device is blank before programming it.
Verify after program	If checked, the ChipProg-02 will verify the device content after it has been programmed.
Verify after read	If checked, the ChipProg-02 will verify device content once it has been read.
On Device Auto-Detect or 'Start' Button:	The checked radio button in this group defines what CPI2-B1 will do upon when either **'Start'** button has been pushed or when the programmer detected the START signal applied to the pin #4 of the CONTROL connector.

3.2.4.5.2.1 Split data

Radio buttons in the **Split data** group of the **Option** tab control programming of 8-bit memory devices to be used in microprocessor systems with 16- and 32-bit address and data buses. In such cases

buffer contents have to be properly prepared in order to split single memory file into several smaller files. Splitting the data allows to convert data read from 16- or 32-bit devices in a way required to create file images for writing them to memory devices with byte organization.

Radio Button	Description
No split	This is the default. The buffer is not split and is treated as an array of single-byte data.
Even byte	The data in the buffer is treated as an array of 16-bit words. The buffer-device operations are conducted using **even bytes only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 0. The byte read from device address 1 will be stored in the buffer location at address 2, etc.
Odd byte	The data in the buffer is treated as an array of 16-bit words. The buffer-device operations are conducted with **odd bytes only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 1. The byte read from device address 1 will be stored in the buffer location at address 3, etc.
Byte 0	The data in the buffer is considered to be an array of 32-bit words. The buffer-device operations are conducted with **the byte #0 only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 0. The byte read from device address 1 will be stored in the buffer location at address 4, etc.
Byte 1	The data in the buffer is considered to be an array of 32-bit words. The buffer-device operations are conducted with **the byte #1 only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 1. The byte read from device address 1 will be stored in the buffer location at address 5, etc.
Byte 2	The data in the buffer is considered to be an array of 32-bit words. The buffer-device operations are conducted with **the byte #2 only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 2. The byte read from device address 1 will be stored in the buffer location at address 6, etc.
Byte 3	The data in the buffer is considered to be an array of 32-bit words. The buffer-device operations are conducted with **the byte #3 only**. For example, programmer reads one byte of device data at address 0 and stores the byte in buffer location also at address 3. The byte read from device address 1 will be stored in the buffer location at address 7, etc.

3.2.4.5.3 The Statistics tab

This tab displays statistics of programming session - **Total** number of devices programmed during the session, what was the yield (**Good**) and how many devices have failed (**Bad**). These statistics are helpful when you need to program a series of same type devices. It is important to remember that statistic counters are affected by executing the Auto Programming only, execution of other functions has no effect on statistics.

Control	Description
Clear statistics	Resets the statistics.
Device Programming Countdown	Normally the **Total** counter increments after each Auto Programming; the **Good** and **Bad** counters also count up. The ChipProg-02 reverses the counters to decrement their content (to count down).
Enable countdown	If checked the ChipProg-02 will count the number of the programmed devices down.
Display message when countdown value reaches zero	If checked the ChipProg-02 will issue a warning when the **Total** counter reaches zero.
Reset counters when countdown value reaches zero	If checked the ChipProg-02 will reset all counters when the **Total** counter reaches zero.
Count only successfully programmed devices	If checked the ChipProg-02 will count only successfully programmed (**Good**) devices. All other statistics will be ignored.
Set initial countdown value	Clicking on this button opens a field for entering a new **Total** number that will then be decremented after each Auto Programming.

Below you can see an example of **Statistic** tab displays programming session statistics for each of four programming sites. **Total** number of devices that were programmed during the session, what was the yield (**Good**) and how many devices have failed (**Bad**).

Program Manager				
Program Manager	Options	Statistics		

Site #	Total	Remaining	Good	Bad
1	8		8	0
2	8		7	1
3	8		8	0
4	8		7	1
All	**32**	**970**	**30**	**2**

⤺ Clear statistics

Device Programming Countdown

☑ Enable countdown

☑ Display message when countdown value reaches zero

☑ Reset counters when countdown value reaches zero

☑ Count only successfully programmed devices

Set initial countdown value...

Current initial value: 1000
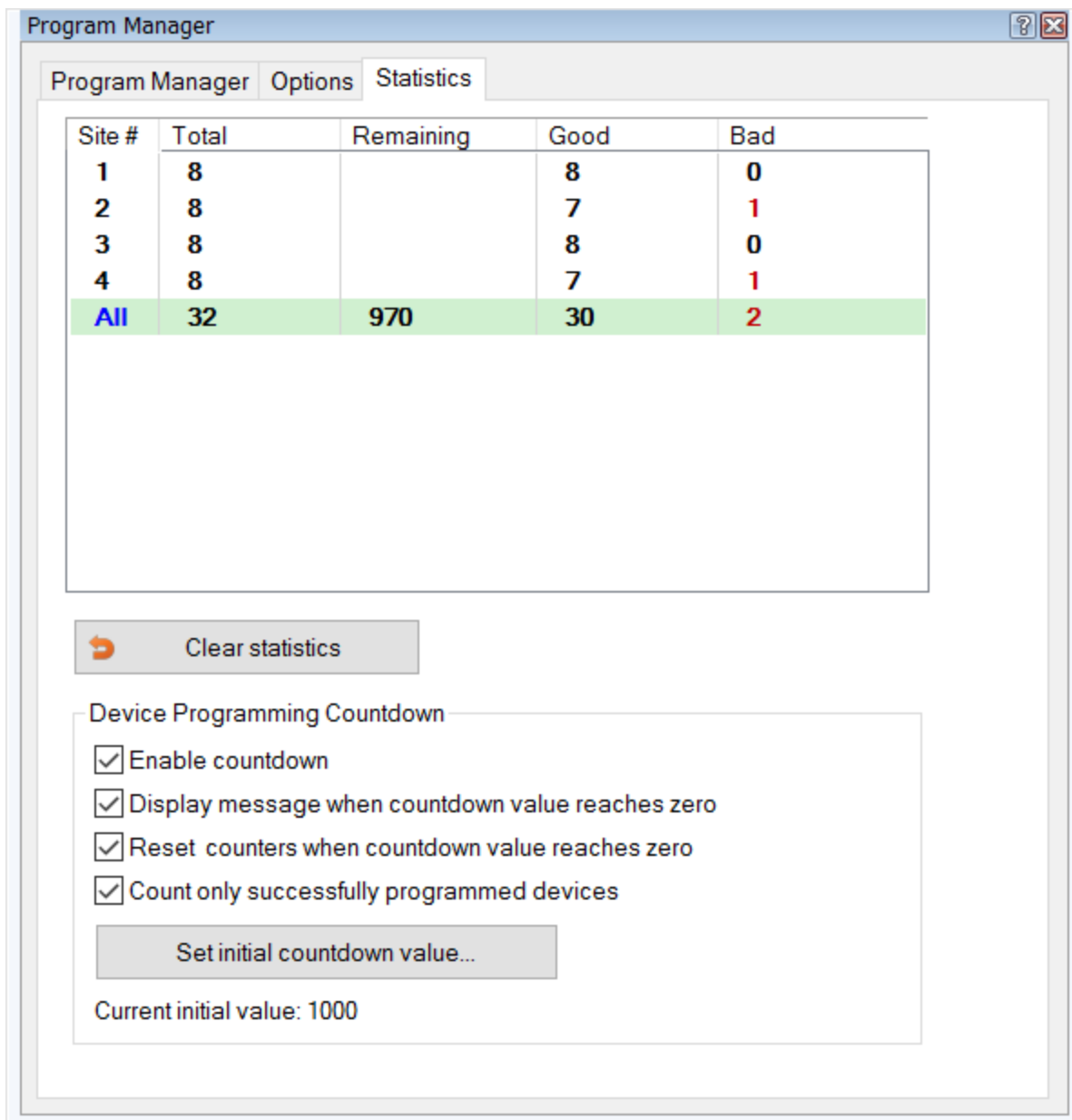
3.2.4.6 The Memory Card Window

The window displays information about projects stored on memory cards in programmers, about limit counter, and about serialization record counter. The window can be brought up using menu **"View"** -> **"Memory Card Window."**

```
Memory cards window                                                    ? _ □ ✕
Collapse All │ Expand │ Erase
□ S/N: SI2-10002, Card: 7.42 GB
   ⊞ Project   0: Job: -, "<Unnamed>", Chip: "Macronix MX25L12873FM2I    [ISP Mode]", Da
   □ Project   1: Job: -, "TXY-01-Atmel", Chip: "SST SST25VF040B    [ISP Mode]", Data: 1
      ⊞ Files loaded into buffers
      ─ Limit: None
      □ Shadow areas (0 of 1000 devices programmed)
         □ Shadow Area Descriptors
            ─ #0: SubLevel: 0, Addr:    0x2008, Size:     4, Type: Serial Number
            ─ #1: SubLevel: 0, Addr:    0x2020, Size:     4, Type: CRC
            ─ #2: SubLevel: 0, Addr:      0x32, Size:     8, Type: User
         □ Shadow Areas Data (1000 total)
            ─ S/N: 0000AB78, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB79, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7A, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7B, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7C, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7D, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7E, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ S/N: 0000AB7F, CRC: 000001FE, User: 00000032: 00 00 00 00 00 00 00 00
            ─ ...992 more records
```

Click the **Erase** button in the window toolbar deletes selected project from the card. This is useful when the card is filled up to capacity.

3.2.4.7 Windows for Scripts

ChipProg-02 provides windows for working with scripts.

- **(Script) Editor** window
- **Watches** window
- **User** window
- **I/O Stream** window

These windows cannot be opened from the **View menu**; they can only be opened when you work with scripts. Operations with these windows are described in the **Scripts Files** chapter.

3.3 Simplified User Interface

The CPI2-B1 default graphic user interface makes heavy use of menus, windows and controls that are redundant in case of mass production. Furthermore, an unskilled operator is usually employed for such production. Programming a lot of chips or boards of the same type with the same data is routine work that consists of two operations: replacing target boards in a test fixture and executing a predefined batch of programming operations (**Auto Programming** command). To prevent casual CPI2-B1 mismanagement and to simplify routine operations, the ChipProg-02 enables switching the CPI2-B1 graphical user interface from the default mode to the **Simplified User Interface** mode (**SUI**). In this mode, operator can see a very simple PC screen with very limited information: a single Start button and three virtual LEDs that indicate CPI2-B1 status: Good, Busy or Error. Or, for the Gang Programming control mode, each site has its own **Start** button (see the SUI screen examples below).
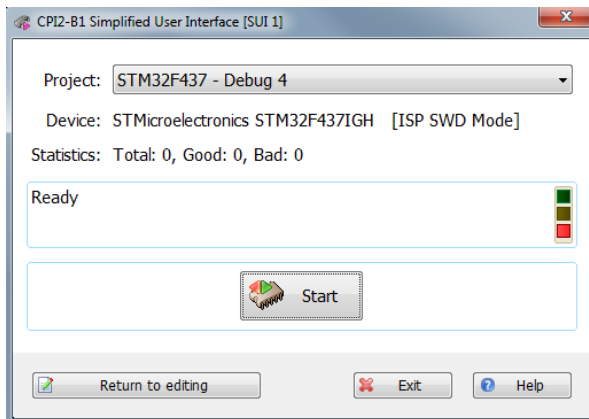
The screen shot above displays SUI set for launching a single CPI2-B1 device programmer.
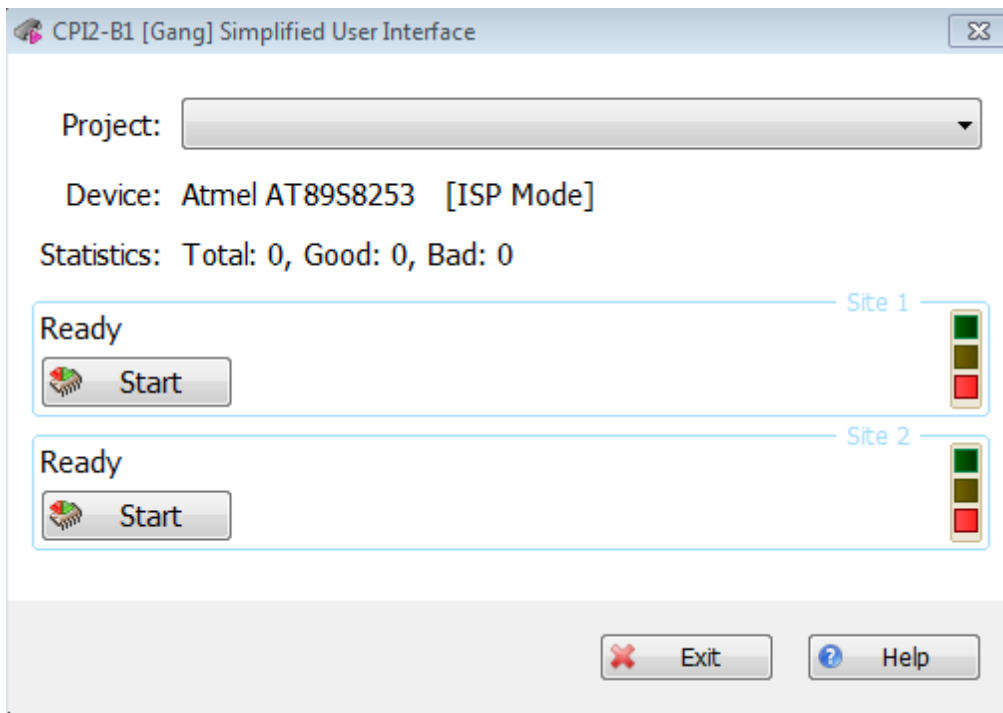
The screen shot above displays SUI set for launching two CPI2-B1 device programmers running in the Gang Programming mode when each of two device programmers can be launched asynchronously and independently. Each site has its own **Start** button. The screen shot below displays the same but when both device programmers starts synchronically by clicking one common **Start** button.
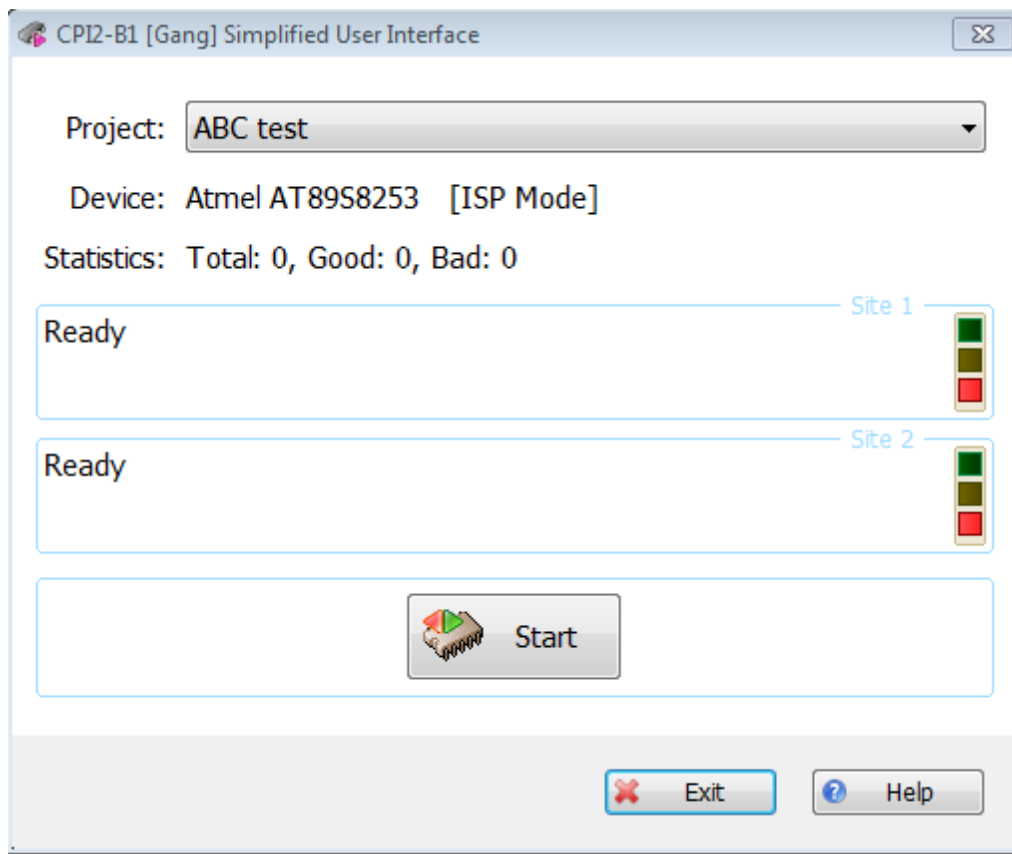
NOTE. Two conditions should be preserved for use of SUI mode. A programming session:
- should be configured by making a **project:**
- can be started by executing **Auto Programming** command, only.

A typical use scenario consists of two steps: Preparation and Use.

1. **Preparation.** An engineer or a technician (hereafter a supervisor) configures the programming session using the default CPI2-B1 graphical user interface and saves the session project. Project file can be stored at any location on PC hard drive. To launch the CPI2-B1 with the SUI, a supervisor can create a PC desktop icon and specify the project and configuration files. After that supervisor switches the user interface to SUI mode for use of the CPI2-B1 by a less skilled operator.

2. **Use.** There are two methods launching the programming when it is controlled via SUI: automatically by an ATE signal or manually by an operator. In case the ATE (test fixture) generates the START signal on the CPI2-B1 CONTROL connector (for example, upon closing the fixture lid and contacting test needles the target device) this launches preset programming session. An operator then keeps replacing target boards and close the fixture lid to continue programming boards. Alternatively launching the programming can be initiated by either clicking the **Start** button in the CPI2-B1 Simplified User Interface window or by pressing the **Start** button on a top of the CPI2-B1 unit.

Settings of Simplified User Interface

Operations with Simplified User Interface

3.3.1 Settings of Simplified User Interface

A session project contains information on device type, file name, serialization parameters, check sum, list of the functions included in the Auto Programming batch and other options, including the SUI windows and controls configurations. The SUI interface settings contain a list of pre-configured projects, so that operator can pick a project from the list in the **Use project** pane unless the **Allow operator to select project from the list** box is unchecked. This option can be set by a supervisor.

To control programming sessions using SUI you first need to create a project. Start with the following steps.

- Configuration menu - select target device.
- Configuration menu - set up a buffer.
- Configuration menu - set options for device serialization, writing check sum and signatures, and log file controls.
- Device and Algorithm Parameters Editor window - specify the options different from default for a chosen device.
- Program Manager window > Program Manager tab > Edit Auto dialog - configure Auto Programming batch of functions.
- Program Manager window > Program Manager tab - set programming options.
- Program Manager window > Statistics tab - enter the number of chips to be programmed and select other options. When using SUI, countdown of programmed chips is disabled, and the program only displays the numbers of successfully programmed and failed chips. Other options set in this tab remain in force.

Once the above settings are done, create the project. In the menu select **Project > New.** In the Project Options dialog enter project name, file name, format, and other information. Click **OK** button to save the project to disk.

NOTE. *It is absolutely crucial* that the project is stored on disk before use. The ChipProg-02 does not protect the SUI project files and window configurations against unauthorized modifications by an operator or any third party.

Once the project has been created and stored on the hard drive, set SUI options. In Configuration menu select the **Simplified Mode Editor** command. This will bring up **Simplified User Interface Setup** window docked to the **SUI** window at its left. The picture below displays the Setup pane only. Any changes made in the **Simplified User Interface Setup** window immediately become visible in the **SUI** window. Clicking the **OK** button in the **Simplified User Interface Setup** window completes the SUI setup; the setup window is closed and **Return to Editing** button appears in the **SUI** window. This allows quick switching back and forth between SUI session setup and actual device programming.

The **Simplified User Interface Setup** dialog has two tabs described below**.**

The General Settings Tab
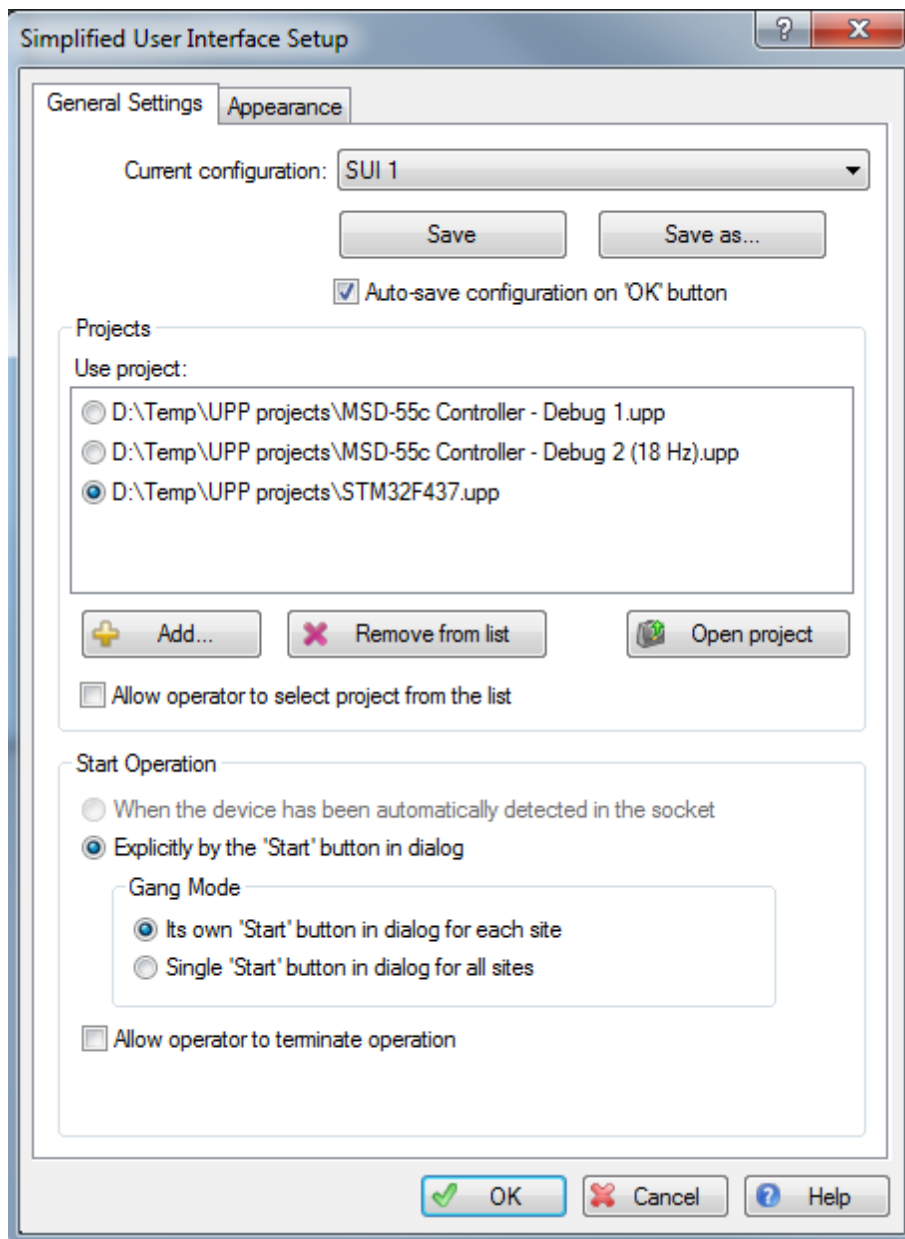
The **Current configuration** field displays the name of currently active SUI configuration. SUI configuration files with have name extension **.smc** and are stored in **SMConfig** sub-folder of ChipProg-02 working folder.

The **Save** button writes current configuration to a file under the name shown in the **Current configuration** field; the **Save as...** button allows saving configuration file under a different name. If the **Auto-save configuration on 'OK' button** box is checked, clicking on **OK** button at the bottom automatically saves current configuration before dismissing the dialog.

The **Projects** pane lists all projects associated with current configuration. When **Simplified User Interface Setup** window is opened for the first time, the **Projects** list will be empty. To add a project use the **+ Add** button. Single configuration may include more than one project; this allows operator to

change projects without restarting the programmer. If **Allow operator to select project from the list** box is checked, the **SUI** window will list all projects associated with current configuration. Otherwise, only one project selected from the **Use project** list will be displayed. To remove a project from the **Use project** list, highlight it and click the **x Remove from list** button. Removing project from the list does not remove it from disk. The **Open project** button loads selected project from disk; this will not close editor window.

The **Start Operation** pane specifies a method of **manual** launching programming operation.

The only batch command that can be launched in SUI mode is **Auto Programming**. This command is executed either by pressing the physical button on the CPI2-B1 unit or by clicking the '**Start'** button in the **SUI** window.

NOTE. These settings do not block or influence in any other way launching CPI2-B1 by an external START signal generated by ATE on the CONTROL connector.

If **Allow programming termination by operator** box is checked, the operator will be able to interrupt programming by clicking **Exit** button in the **SUI** window, otherwise the operator will only be able to initiate device programming.

The Appearance Tab

Here you can choose the type, size and color of the **Default Font** for each element in the **SUI** window: **Project name, Device part number, Statistics, Device operation status,** and **"Start" button**. Checking boxes in **Display elements** list makes corresponding elements visible in the **SUI** window. Clicking **Move up** and **Move down** adjusts position of selected element within the window.
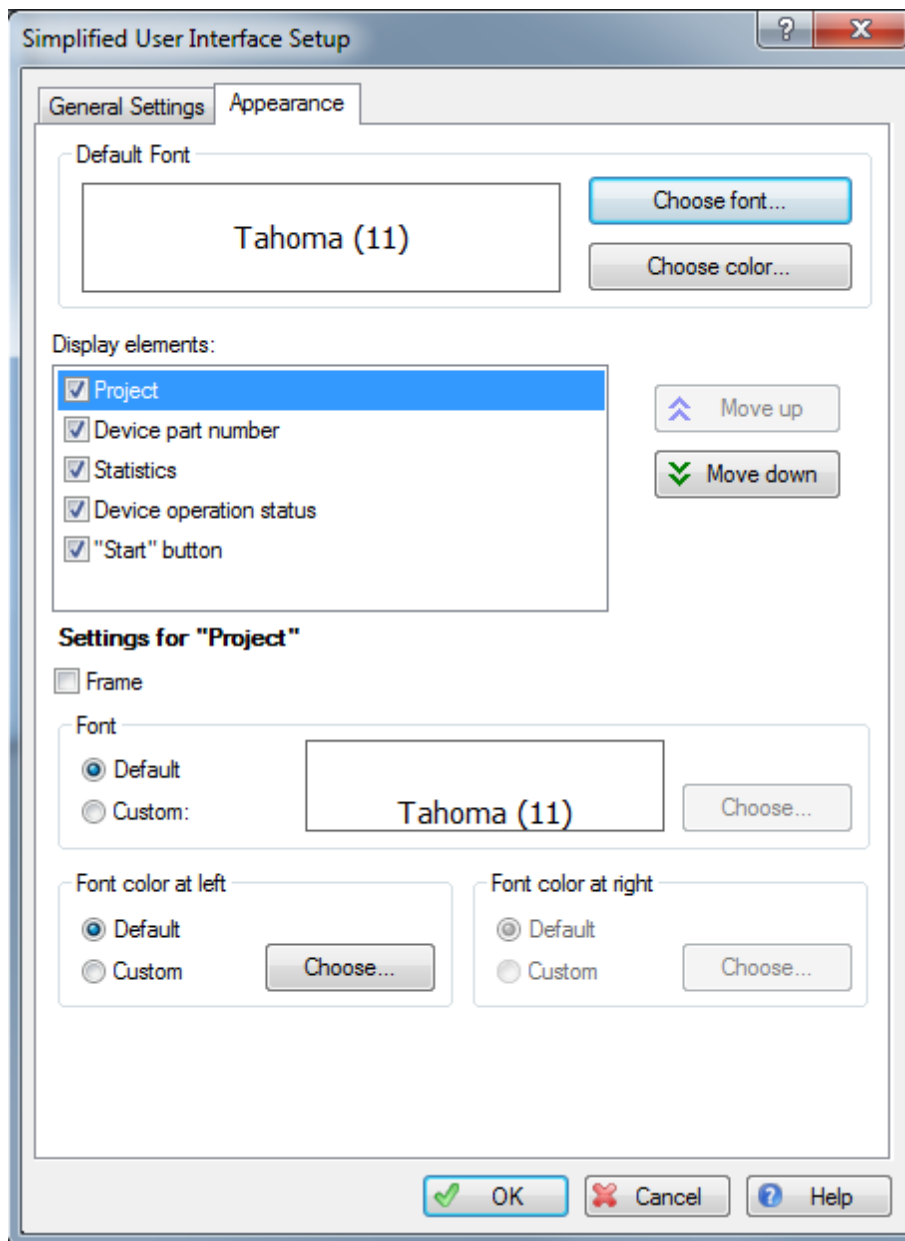
If an element is set to be visible in the **SUI** window, you can modify its appearance to differ from the default and from other elements. Checking the **Frame** box causes a thin blue frame to appear around the element. The **Font**, **Font color at left** and **Font color at right** radio buttons modify appearance of an element to make it distinct from other elements in the **SUI** window.

When the **Statistics** element is highlighted, **Allow operator to reset statistics** box will be displayed. Check this box to allow operator clear displayed programming statistics.

When the **Device operation status** element is highlighted, two additional checkboxes, **Serial number** and **Checksum** are displayed. Checking these boxes makes serial number and check sum written into the last programmed device be displayed below the status line.

3.3.2   Operations with Simplified User Interface

To launch programming operations controlled by a configured **Simplified User Interface** open the **Command** menu, and double click the **Switch to Simplified User Interface..** line.

To launch the ChipProg-02 with the **Simplified User Interface** (or in the **Simplified Mode**) use the **/Y<configuration name>** option key in Command line mode (there must be no spaces between **/Y** and **<configuration name>**). If **<configuration name>** includes spaces, it must be quoted. For example, if the configuration name is **STM32F429BGT [ISP SWD Mode] - Release,** the command line may look like this:

C:\Program Files\ChipProg-02\6_00_20\UprogNT2.exe /Y"STM32F429BGT [ISP SWD Mode] - Release" ,

When launched in the Simplified Mode, the ChipProg-02 only displays the SUI window. The main ChipProg-02 window remains invisible unless an error occurs. If a programming operation fails, the programmer performs actions according to error handling settings. These settings are available via Configure > Preferences menu. If the **Terminate device operation on error and do not display error message...** box in the **Preferences** dialog is unchecked (default setting), the ChipProg-02 issues an error message and prompts the user to either ignore the error and resume operation or terminate it. If this box is checked, any error will cause the programming session to come to a halt; in such case no error message will be issued.

3.4   Command Line Interface

The ChipProg-ISP2 device programmers (both CPI2-B1 and CPI2-Gx) can be controlled from **Command Line** using **UProgNT2.EXE** executable.

Command line has the following format:

UProgNT2.exe [option 1] [option 2] ... [Name of the project file] [option 3] [option 4]…

Elements in square brackets are optional and may follow in arbitrary order, separated by spaces. These elements are called *options,* square bracket characters themselves are not part of the option. Options specify certain CPI2-B1 functions and settings. Some options are called *keys*. Command line may also optionally contain the name of a project file that will be used to control programmer operation.

Each option begins with either '/' (slash) or '-' (hyphen) followed by an option name. The slash and hyphen characters can be used interchangeably; for example: '/L', is the same as '-L'. Valid names are listed in the Command line options table.

Otion names, project names, and the application executable name are case-insensitive, so there is no difference between the **'/A'** and **'/a'** options. Names containing spaces must be quoted, for example: -L"Data file 5.hex".

Some options listed in Command line options table require additional parameters; these are shown in the table enclosed in angle brackets (< >). Parameters specify file names, devices, text strings, serial numbers, etc. Parameters must follow options without space. For example: "/LData file 5.HEX" (load the Data file 5.HEX into the buffer after launching the programmer) or "/FH" (file format is hexadecimal).

Upon executing a command line the ChipProg-02 checks whether a project loaded before the program has been closed at the previous programming session. If it has, the program automatically reloads this old project unless a new project name is specified in the command line.

There is no difference between loading a project by executing a command line, or loading it manually by means of the ChipProg-02 user interface menus.

Some command line examples are:

1) UProgNT2.exe -C"Atmel^AT89C51ED2 [ISP BL Mode]" -L"C:\Work\Output Files\Bin\Serial.bin" -FB0x2000 -A -I2

Launch the ChipProg-02 application, then:

-C"Atmel^AT89C51ED2 [ISP BL Mode]" - select the Atmel AT89C51ED2 [ISP BL Mode] device;
-L"C:\Work\Output Files\Bin\Serial.bin" - then load the file C:\Work\Output Files\Bin\Serial.bin into the buffer #0;
-FB0x2000 - specify the binary format for the Serial.bin file with the start address 0x2000 in the buffer;
-A - then begin the Auto Programming session using the default set of commands programmed in the Auto Programming menu;
-I2 - make the ChipProg-02 main window invisible, when the Auto Programming session completes. If an error occurs, copy error message to clipboard and close the ChipProg-02 application.

2) UProgNT2.exe "C:\Work\Programmer Projects\Nexus.upp" /A1

Launch the ChipProg-02 application, then load project file "Nexus.upp" from folder "C:\Work\Programmer Projects" and launch the Auto Programming session from buffer #1. If programming was successful close the ChipProg-02 application. The CPI2-B1 main window remains visible.

3) UProgNT2.exe

Launch the ChipProg-02 with no options.

3.4.1   Command Line Options

Option name starts with either '**/**' (slash) or '**-**' (hyphen), followed by one of the reserved names listed below. The slash and hyphen characters have the same effect and can be used interchangeably, for example: '/C', '-C'.

Option	Description
-N<serial number>	If more than one CPI2-B1 programmers are connected to one computer the **-N** key enables control of a certain device programmer by specifying its serial number. This key *cannot be used* in combination with the key **-GANG** i.e. when multiple programmers were launched in gang-programming (gang) mode (see below).
	The serial number can be found on the bottom of programmer case or by using the Help > About... menu command. Serial numbers of all programmers connected to a PC are also available in the "**Choose programmer**" dialog. The ChipProg-02 program shows this dialog if the command line does not have the **-N** option.
	For example, the option **-NSI2-10012** specifies that all other command line options apply to the programmer with serial number **SI2-10012** only.

Option	Description
-GANG	This option launches multiple CPI2-B1 device programmers in gang-programming (gang) mode. In this mode the ChipProg-02 software controls multiple CPI2-B1 programmers connected to a single computer. The **-GANG** key *cannot be used* in a combination with the **-N** key. The **-GANG** option can be used either alone, without any specifiers, or with one of two following: **<number of sites>** or **#<list of serial numbers>.** Each specifier requires use of its own **-GANG** key. For example: **-GANG:4,** **-GANG#SI2-10014;SI2-10022.** You cannot set both of these specifiers by a single **-GANG** key. Below see detail descriptions of use the **-GANG** option with the **<number of sites>** and **#<list of serial numbers>** descriptors:
-GANG:<number of sites>	If the **:<number of sites>** parameter follows the **-GANG** key then after launching the ChipProg-02 application it is waiting until the program detects a specified number of CPI2-B1 device programmers connected to a PC or for 16 sec, whatever is longer. For example, the **-GANG:2** key stops attempts to establish communication after the first two CPI2-B1 device programmers have been detected. The **:<number of sites>** parameter can be omitted.
-GANG#<list of serial numbers>	If the **-GANG** key is followed by '**#**' sign with a list of serial numbers separated by semicolons, the application waits until the number of connected single-site programmers matches the number of serial numbers in the list, then automatically assigns sequence numbers according to the serial numbers in the list. For example, if the -**GANG#SI2-10014;SI2-10022** is specified, the application waits for establishing connections with two device programmers with serial numbers **SI2-10014** and **SI2-10022**; the programmer with serial number **SI2-10014** will be assigned the sequence site number **1** and programmer with serial number **SI2-10022** will be assigned the site number **2**.
-ETH	This option initiates control one or more CPI2-B1 device programmers connected to a local network (LAN) via Ethernet (USB is a default option that does not require use of any keys). The **-ETH** option can be used either without any specifiers or with one of two following: **<number of sites>** or **#<IP addresses list>.** Each specifier requires use of its own **-ETH** key. For example: **-ETH:4, -ETH#192.168.1.{2-128}.** You cannot set both of these specifiers by a single **-ETH** key. Below see detail descriptions of use the **-ETH** option with the **<number of sites>** and **#<IP address list>** descriptors:
-ETH:<number of sites>	If no parameters follow the **-ETH** key the program pings IP-addresses of LAN adapters in a range automatically detected by a computer. This process may take up to 16 seconds. To speed up connecting all the programmers it is recommended to specify a **<number of sites>** parameter. For example, for driving a single CPI2-B1 programmer via Ethernet include the **-ETH:1** option in the command line. In most cases this allows to establish communications in a few seconds.
-ETH#<IP addresses list>	This option specifies an individual IP address or a range of multiple IP addresses to be pinged by a computer while it tries connect CPI2-B1 device

Option	Description
	programmer(s). Normally, in a local network (LAN), IP addresses are assigned by a DHCP server automatically. The DHCP server dynamically distributes IP addresses used by CPI2-B1 programmers.
	However, it is possible to specify static IP address if it is assigned to a particular CPI2-B1 unit or a list of IP addresses or a range of IP addresses assigned to multiple units. See the examples below:
	-ETH#192.168.1.32 - connect a device programmer with the 192.168.1.32 static IP address.
	-ETH#192.168.1.32;192.168.1.38 - connect device programmers with either the 192.168.1.32 or the 192.168.1.38 IP address. After launching the program you will be prompted to select one of two IP addresses above.
	-ETH#192.168.1.{16-128} - scan IP addresses in a range of 192.168.1.16 to 192.168.1.128.
	-ETH#192.168.1.* - scan IP addresses in a full range of 192.168.1.1 to 192.168.1.254.
	-ETH#192.168.1.{12-33,127,164-254} - scan IP addresses in a range of 192.168.1.12 to 192.168.1.33, then ping a single address 192.168.1.127 and then scan a range of 192.168.1.164 to 192.168.1.254.
	-ETH#192.168.1.* -ETH:1 - scan IP addresses in a full range of 192.168.1.1 to 192.168.1.254 and stop scanning upon connecting to the first detected device programmer.
-C"<manufacturer ^device>"	This option tells the ChipProg-02 program to use the device specified as manufacturer name followed by a **^** character followed by device part number specified here exactly as it presents in the CPI2-B1 device list. The device specified in a previously loaded project will be replaced by a device specified by the -**C"<manufacturer^device>"** key.
	For example: **-C"NXP^MC9S08DV32MLF[ISP Mode]"**.
	Note. The use of the -**C** option is less beneficial than using projects. Projects provide much more flexible and effective control of device programming. It is highly recommended, especially for mass production, to create, configure, and save as many projects as needed and use them with command line.
-L<file name>	This option loads the **<file name>** file into the CPI2-B1 buffer upon launching the ChipProg-02 program. If other files were previously loaded using some project, then a new one will be loaded in accordance to the file format and start address. The loader determines file format from the file name extension. If actual file format differs from the one listed in the file format list use the **-F** option to explicitly specify file format (see below).

Option	Description
-F\<file format\>	This option sets format of the file specified by the -**L\<file name\>** option. The **\<file format\>** must be one of the following letters: **H** - standard or extended Intel HEX format **B** - binary format **M** - Motorola S record format **P** - POF (Portable Object Format) **J** - JEDEC format **G** - PRG format **O** - Holtek OTP format **V** - Angsrem SAV format For example, -FH option loads file in the HEX format, which contains starting address in CPI2-B1 buffer. If binary format **(B)** is specified by the -**F** option, it may be followed by a hexadecimal value for destination starting address. For example: the option -FBFF04 loads binary file and places data starting at the address FF04h in the buffer. In the absence of **-L\<file name\>** the -**F\<file format\>** option is ignored.
-A[buffer number]	This option initiates the Auto Programming session upon launching the ChipProg-02 application. Upon successful completion the application terminates. In case of error the ChipProg-02 application remains open until it is manually closed by operator. If the **[buffer number]** is omitted, the data for Auto Programming are taken from buffer #0; otherwise the data are taken from the buffer with the number that follows -**A**. For example: the option -A2 specifies that data for the Auto Programming session will be taken from the buffer number 2. The **-A** option is only meaningful if a project name or an **-L\<file name\>** option is also specified on the same command line.
-I	This key hides the ChipProg-02 main window. If an error occurs during programming process, the window is displayed on the PC screen along with the error message. This option is only meaningful if an **-A** (Auto Programming) options is specified on the same command line; otherwise the **-I** option is be ignored.
-I1	This key is similar to the **-I** key except the **-I1** keeps the ChipProg-02 main window hidden even if a programming error occurs. The first occurrence of a programming error terminates the ChipProg-02 program and returns the error code 1. (Successful Auto Programming session ends with return code 0.) Return codes can be used by external applications that control the CPI2-B1 remotely, such as LabVIEW, similar programs, or batch files.
-I2	This key is similar to the **-I** key; however, **-I2** keeps the ChipProg-02 main window hidden at all times, suppresses error messages display, but copies the error message to Windows clipboard.

Option	Description
-M	This key starts the ChipProg-02 software in the demo mode, without use of the CPI2-B1 hardware and without real data exchange between computer and programmer hardware. This mode is convenient for evaluating the product without use of CPI2-B1 hardware.
-S<file>	This key replaces the default session configuration file **UPROG.ses** with a new one named **<file>** (with the extension **.ses**). Session configuration file stores major CPI2-B1 settings, and includes the name of the most recently used project; it resides in the ChipProg-02 folder. The new session settings will be used by the ChipProg-02 when invoked from command line.
-O<file>	This key replaces the default option configuration file **UPROG.opt** with a new one named **<file>** with the extension **.opt**. Option configuration file stores target device type, file options, etc.; it resides in the ChipProg-02 folder. The new options will be used by the ChipProg-02 when invoked from command line.
-D<file>	This key replaces the default desktop configuration file **UPROG.dsk** with a new one with name **<file>** and extension **.dsk**. Desktop configuration file stores computer screen configuration, i.e., positions, dimensions, colors and fonts for all open windows; it resides in the ChipProg-02 folder. The new desktop configuration will be in force when ChipProg-02 is invoked from command line.
-ES<file>	This key executes a script whose file name follows the **-ES** key, immediately after starting the ChipProg-02 application. If the command line does not include the **-ES** key, the ChipProg-02 application searches for the script file named '**Start.cmd**' in the working folder and, if such script exists, executes it.

3.5   On-the-Fly Control Interface

The **On-the-Fly Control** interface is very similar to command line control interface. However, it can control a CPI2-B1 programmer that has already been started and is running, without restarting it. **On-the-Fly Control** interface can be used to start any operation available for target device, such as Read, Program, load project, execute script, etc. **On-the-Fly Control utility** can be used to control a running CPI2-B1 programmer by Windows batch files coming with third-party graphical packages such as National Instruments LabVIEW.

The **On-the-Fly Control utility** is an alternative to a more advanced Application Control Interface (DLL control); using the latter requires some programming skills.

The **OFControl.exe** executable resides in the ChipProg-02 installation folder. We suggest you keep it in that folder and start it from there. Once started, the utility does not modify its working directory..

After completion, **On-the-Fly Control utility** issues return codes. The code is 0 (zero) in case of success. Error codes are listed in the **UPControl return codes** section. The program writes error messages to the Console window and, optionally, to log file and/or Windows clipboard.

After the **On-the-Fly Control** process has exited, CPI2-B1 keeps running unless **On-the-Fly Control** utility had been started with the **-X** key. You may re-launch the **On-the-Fly Control utility** to control the same device programmer. However, please keep in mind that only one On-the-Fly Control utility can control a running device programmer at the same time. In case you launch a second instance of the **On-the-Fly Control** while the CPI2-B1 device programmer is being controlled by previously launched instance, the second instance will not "find" the programmer.

The On-the-Fly Control command line format is as follows.

OFControl.exe [Options] [@<Option File>] [Options]

Each option starts with either '**/**' (slash) or '**-**' (hyphen) character, followed by a name. Valid names are listed below. The '/' (slash) and '-' (hyphen) can be used interchangeably. *For example, '/L', '-P'.*

The order of **options** in the command line is not important. Operations specified by options are performed in logical order. For example, operations on target device will be performed after loading a project and executing a script, regardless of the order in which options appeared on command line. However, the -**F<device operation list>** and -**A options** are exceptions. These options define an order of operations on target device, therefore they are always performed according in the order they are appear on the command line.

Note. Brackets [] in option descriptions denote optional parameters; brackets should not be used when specifying actual parameters. Angle brackets <> are used to denote entities and are not part of the option notation. For example, replace -G[+] with -G+; replace -G[+][<C:\Temp\UPC.log] with -G +C:\Temp\UPC.log.

If a file name used in an option includes spaces, full name with the path should be used. Any additional part of an option should not be separated by spaces. For example, -L"H:\Program Files\ChipProg-02 \6_00_20\UprogNT2.exe /g". Here the file name and path is enclosed in quotation marks ("") and there are no spaces between the /L and the rest of the option

The **@<Option File>** construction specifies a text file containing additional options for **On-the-Fly Control utility**. Each option in such file must be listed on a separate string. For example: :

```
UPControl.exe -D @response.txt -WK
```

In the option file, lines starting with semicolon (;) are treated as comments and are ignored. A commented example file **response.txt** is shown in the **Option File example.**

3.5.1   On-the-Fly Command Line Options

On-the-Fly Control command line has the following format:

OFControl.exe [Options] [@<Option File>] [Options]

The following table provides detailed descriptions of available options.

Option	Description
-D	Debug mode: include additional information in console log and in log file. This option is helpful for debugging **On-the-Fly Control** program.

Option	Description

-G[+][<log file name and path>]	Send the **ChipProg-02** Console window output also to a log file. If **-G** is followed by a **+** sign output will be appended to the log file if it exists. If the + sign is omitted a new log file is created. By default the log file is called **OFControl.log** and resides in the **ChipProg-02** working folder; you can specify a new file name and location if desired.
	
	Examples:
	  -G - create a new log file, named OFControl.log, in the OFControl.exe working folder.
	  -G+ - append records to OFControl.log file if it exists; otherwise create the file.
	  -G+C:\Temp\OFC.log - append records to C:\Temp\OFC.log file if it exists; otherwise create it.

-WK	Keep **On-the-Fly Control** program running until a key is pressed on the keyboard. This allows perusing messages in the Console window before it terminates.
-L< ChipProg-02 executable file name and command line options>	Launch the **CPI2-B1** device programmer if it is not running. If it has been already launched the option is ignored. The **On-the-Fly Control** program executes the **-L** option before all other options on command line, that is before loading a project, executing scripts, or performing any operations with the device. The **-L** cannot be used together with **-R** option (see below). Example: -L"UProgNT2.exe /g1"
-R<device programmer's serial number>	If more than one **CPI2-B1** device programmer is controlled by the PC in the gang mode, connect to the unit whose serial number is given by this option. -R cannot be used in a combination with -L option. If more than one programmer is controlled by the PC and **On-the-Fly Control** command line does not contain an **-R** option, the program terminates with error code #14.
-C	Copy error message to the Windows clipboard. Whenever **On-the-Fly Control** program terminates with a return code other than 0 (except when **-T** option is used, see below), it means that an error has occurred. If the the -C option is given, the error message will be copied to the clipboard; otherwise the clipboard contents remain unchanged. If more than one operation specified on **On-the-Fly Control** command line results in an error, error messages of all operations will be copied to Windows clipboard if the command line also contains the **-I** option (ignore errors).
-M[=<timeout in seconds>]	Specifies timeout in seconds when waiting for device programmer to become ready before performing certain operations. The operations include loading a project, running a script, programming target device, and terminating execution triggered by the **-X** option. If **-M** option is not specified, **On-the-Fly Control** program does not check whether

Option	Description
	ChipProg-02 is ready to perform the next operation. In case it is not, an attempt to perform a programming operation will result in program terminating with an error. If the **-M** option is not accompanied by a **[=<timeout in seconds>]** parameter, **On-the-Fly Control** program will wait for the programmer ready state indefinitely. In this case you can interrupt program execution and make it quit by pressing **Ctrl+C** on the keyboard.
-B	Stop an operation with the device. If **CPI2-B1** performs a programming function (Read, Program, Verify, etc.) on target device, it will be interrupted. This action takes place prior to performing all actions specified by the options **-P, -S, -F, -X** options. It is possible, however, that the **-B** option does not interrupt operation on target device. This happens when the program displays an error dialog that requires operator response. In this case **On-the-Fly Control** program exits with an error code.
-P<project file>	Load the specified project file. Project files with **.UPP** extensions contain all information and settings defining a programming session (device type, file(s) to be written to the device, customized device and algorithm parameters, interface settings, device serialization options, scripts, etc.). Before loading the project file, **On-the-Fly Control** program waits for the programmer to stop operations on device (see the **-M** option). If the **-P** option is specified on **On-the-Fly Control** command line along with **-S** and/or **-F** options, the project file will be loaded before running scripts or performing any operations on target device. Example: -P"C:\Prog\Projects\Antenna-01 Test.upp"
-S<script file>	Run the specified script. Before running the script **On-the-Fly Control** program waits for the programmer to stop operations on device (see the **-M** option). By default **On-the-Fly Control** program waits for the script to complete. To allow **On-the-Fly Control** program to continue operations while the script is still running, add the **-NWS** option to the option list. Example: -S"D:\Prog Scripts\Checksum.cmd"
-NWS	Do not wait for completion of the script specified by the **-S** option.
-F<function list>	Execute listed operations (functions) on the target device. Names of the functions in the list must be separated by semicolons **(;)**. In order to execute the **Auto Programming** function the **-F** option should be followed by an asterisk character **(*)**. If command line has more than one **-F** option, functions will be executed in the order in which they are specified on the command line.

Option	Description

	If one or more **-F** options is specified in the command line along with **-P** (load project) and/or **-S** (launch script) options, all functions specified by **-F** option(s) will be performed after loading the project file and/or running the script.
	
	By default **On-the-Fly Control** program waits for function to complete before proceeding. To enable the program to proceed while function specified by the **-F** option is still executing, add the **-NWF** option to command line. In this case you may specify only one **-F** option on the command line.
	
	If an -F option specifies a sub-function displayed in the drop-down menus of the **Program Manager** function tree, use both menu name and function name separated by the caret **'^'** character. For example: -FProgram (for the Code Memory chip layer) but -FData Memory^Program (for the Data Memory) .
	
	Examples:
	
	  -F* - launch the Auto Programming function.
	  -FErase;Blank Check;Program;Verify - erase the device, check if it is blank, write the file from the programmer buffer and compare the buffer and device memory contents.
	  "-F*;Verify;Device Parameters^Program HSB and XAF" - execute the Auto Programming function, then compare the buffer and device memory contents, then launch the function Program HSB & XAF from the Device Parameters sub-menu.

-NWF	Do not wait for completion of the function specified by **-F** option. This option is incompatible with **-X**.
-I	Ignore errors during programming operations. By default **On-the-Fly Control** program stops operations on target device in case of any error. The **-I** option enables the operations to continue regardless of error conditions; this allows logging of all errors that occurred.
-T[+][W=<delay in milliseconds>]	Wait for programmer status ["Ready" or "Busy"]. **On-the-Fly Control** program returns code 0 (zero) when **CPI2-B1** stops and becomes ready to perform a programming operation ("Ready"), or 1 if an operation on target device is underway ("Busy"). In addition, if '**+**' sign follows the -T and the programmer status is busy, current function name (Read, Program, etc.) will be output to the console window along with the completion percentage of the function being executed. For example: Program, 87%. Optional **[W=<delay in milliseconds>]** parameter sets a delay before getting the programmer status. Delays allow checking programmer status within a settable period of time. Examples:

Option	Description
	-T - get the programmer status "Ready" or "Busy" -TW=1000 - wait for 1 sec, then get the programmer status "Ready" or "Busy" -T+ - get the programmer status "Ready" or "Busy" then output to the Console window the name of currently executed function and percentage of its completion. An example of the function status string: Read 56%.
-V=[0 | 1]	Hide (-V=0) or make visible (-V=1) the **ChipProg-02** main window. If **ChipProg-02** main window is hidden, the program will not be present among other open applications in the **Applications** tab of the **Windows Task Manager**. In order to stop a running **ChipProg-02** program you will have to go to the **Process** tab of the Task Manager, then locate and highlight the programmer executable name (**UprogNT2.exe**) and click the **End Process** button.
-X	Stop the programmer and quit the program. To quit the **ChipProg-02** program, the programmer must complete all current operations on the device. The **On-the-Fly Control** program waits for completion of the current programming operation for the period of time specified by **-M** option. If this option is omitted or the timeout period has expired, **On-the-Fly Control** returns an error.
-? or **-H**	Show a brief description of the **On-the-Fly Control** program options and exit.

3.5.2   On-the-Fly utility return codes

Upon completion **On-the-Fly Control** program returns code 0 (zero) in case of success. Otherwise it returns one of the error codes listed below. There is one exception related to the use of option **–T**. If -T option is specified **On-the-Fly Control** returns 0 if the programmer is stopped and 1 if an operation on the target device is underway.

Error messages are set to the **Console** and, optionally, to a log file and/or Windows clipboard.

Return codes:

0	Successful completion.
1	The –T option was specified and the programmer is busy performing an operation on taget device.
2	Invalid option or parameter on command line.
3	Error calling a Windows API function; it could be caused by an abnormal exit of the programmer software.

4	The programmer application was closed while the **On-the-Fly Control** utility has been waiting for response. Possibly the operator has forced closing of the program.
5	Timeout set by an **-M** option occurred.
6	The programmer was launched in the gang mode but an option in the **On-the-Fly Control** utility tried performing a function not applicable to multiple CPI2-B1 running in the gang mode.
7	Failure to perform requested action because programmer is busy performing anoter operation on the target device.
8	Failure to load project file specified by **-P** option.
9	Failure to run script specified by **-S** option.
10	General error.
11	Programming function specified by the **-F** option is not applicable to current target device.
12	An error occurred while programmer performed operation on the target device.
13	Programmer could not complete an operation and closed the program after receiving the **-X** option request.
14	More than one device programmer is running. **-R** option must be used.

3.5.3   On-the-Fly Control Examples

; Launch programmer in diagnostic mode unless it is already in use
-L"C:\Phyton\ChipProg-02\6_00_21\UProgNT2.exe /g1"

; Append records to the log
-G+

; If programmer is busy, wait for 30 seconds max
-M=30

; Load project file. The FuelPump-08.upp project file is in D:\Projects folder
-PD:\Projects\FuelPump-08.upp

; Execute csm-16.cmd script located in the D:\Scripts folder
-SD:\Scripts\csm-16.cmd

; Execute auto programming using parameters defined by the FuelPump-08.upp project
-F*

4   Operating Procedures

This chapter describes typical operations with a CPI2-B1 device programmer running in the **Single-programming** **control** mode. The description refers to the operation made withing the ChipProg-02 GUI, only.

4.1   How to check if device is blank

1. Select the target device type: press the **Select Device** button in the Main toolbar or select command **Main menu** > **Configure** > **Select device**.

2. Connect a CPI2-B1 programmer to the device.

3. a) Click the **Check** button on the main toolbar, or
   b) Double click on the **Blank check** function line in the **Function** list of the **Program Manager** window, or
   c) Select the **Blank check** function line in the **Function** list of the **Program Manager** window and click the **Execute** button, or
   d) Select the **Main** menu > **Commands** and click on the **Blank check** line.

   Wait for the message **Checking … OK** in the **Program Manager** window, or for the warning message if the device is not blank.

4.2   How to erase a device

1. Make sure the device is electrically erasable. Some devices are not erasable; these may be programmable once or over-writable – in this case the **Erase** button is disabled (grayed out).

2. If the device is electrically erasable:
   a) Click the **Erase** button on the main toolbar or
   b) Double click on the **Erase** function line in the **Function** list of the **Program Manager** window or
   c) Select the **Erase** function line in the **Function** list of the **Program Manager** window and click the **Execute** button or
   d) Select the **Main** menu > **Commands** and click on the **Erase** line.

   Wait for the message **Erasing … OK** in the **Program Manager** window or for the warning message if the device is not blank after erasing.

4.3   How to read data from device

There are several ways of reading device content into the active buffer:

- click the **Read** button on the main toolbar, or
- double click on the **Read** function line in the **Function** list of the **Program Manager** window, or
- select the **Read** function line in the **Function** list of the **Program Manager** window and click the **Execute** button, or
- select **Commands** > **Read** menu command.

In every case above, wait for the message **Reading … OK** in the **Program Manager** window or for the warning message if the device could not be read.

4.4 How to program a device

In order to write (program) a device you need to perform a few consecutive operations:

- load the file that you want to write to the device;
- edit the file (if necessary);
- configure the device to be programmed (if necessary);
- write the prepared information into the device and verify the programming.

4.4.1 How to load a file into a buffer

1. In the main menu select **File** > **Load** or click the **Load** button on the local toolbar of the **Buffer** window.

2. In the pop-up dialog box that appears enter file name, select file format, addresses, buffer and sub-level to load the file to.

3. Wait for the message **File loaded: "......"** in the **Program Manager** window, or for a warning message if the file cannot be loaded for some reason.

4.4.2 How to edit data before programming

1. If you need to modify source data before writing it into the target device, open the **Buffer Dump** window. Please keep in mind that the **View** button must be released to enable editing.

2. Make necessary changes using **Modify** dialog or select the data to be modified and type new data over old data.

4.4.3 How to configure target device

1. Parameters displayed in the **Device and Algorithm Parameters** window that can be modified are shown in blue.

2. Click on the name of the parameter to be changed to open a dialog. Set a new value for the parameter or check/uncheck appropriate boxes and click OK. Modified parameter will be displayed in red.

3. Repeat the above procedure for other parameters that you wand to modify.

Note. All changes above will become effective in the target device only upon programming by the **Program Parameters** function in the **Program Manager** window.

4.4.4 How to write information into the device

1. Click on the **Options** tab in **Program Manager** **window**. Check the options you need. We recommend you always check **Blank check** before programming and **Verify** after programming to ensure reliable programming.

2. Click on the **Program Manager tab**. Select the **Program** line in the **Function box** and double click on it to start programming of the primary memory layer (**Code**). Click on the **Execute** button to launch the process. Alternatively, you can do the same by clicking on the big **Program** button or by selecting the menu command **Commands** > **Program**.

3. Wait for the message **Programming … OK** in the **Operation Progress box** of the **Program Manager tab**. If an error has occurred, ChipProg-02 issues an error message.

4. Execution of the main **Program** function (always shown at the top of the **Function list**) writes the specified buffer layer to the **Code** memory of the device. However, other buffer layers may exist for the selected device (**Data**, **User**, etc.). If more than one buffer layer exists for the selected device, go down in the list of functions, expand those that are collapsed and execute the **Program** functions for as many types of memory as device has (**Data**, **User**, etc.). Skip those steps if only the **Code** layer exists in the device.

5. **IMPORTANT.** If any options in the **Device and Algorithm Parameters Editor window** have been modified, you have to program the options set after programming all memory layers (**Code, Data**, **User**, etc.). Go down to the **Device parameters & ID** line, expand it if collapsed, select the **Program** function and double click on it. Continue until every parameter that has been changed in the **Device and Algorithm Parameters window** is successfully programmed.

6. Some microcontrollers can be protected against unauthorized reading of the code stored in them by setting **Lock bits**. You can selectively lock only certain parts of the device memory. Go down to the **Lock bits** line, expand it if collapsed and double click on the lock bit# lines one by one. Continue until every lock bit you want is set.

7. After every operation described above make sure that you see **Ok [xxxxx... Ok]** message in the **Operation Progress** box of the **Program Manager tab**. In case you get an error message stop programming and troubleshoot the issue.

4.5   How to verify programming

There are several ways to check if device was programmed correctly:

- click the **Verify** button on the main toolbar, or
- double click on the **Verify** function line in the **Function** list of the **Program Manager** window, or
- select the **Verify** function line in the **Function** list of the **Program Manager** window and click the **Execute** button, or
- select the **Commands** > **Verify** menu command.

Wait for the message **Verifying … OK** in the **Program Manager** window or for a warning message if the device verification has failed.

4.6   How to save data to disk

1. After you have read device content into the Buffer or specified Buffer layer you may want to save the data to a PC hard drive or other media. To save the data:
a) Click the **Save** button on the local toolbar of the **Buffer** window, or
b) Select menu command **File** > **Save.**

2. In the pop-up dialog enter destination file path and name, format, start and end addresses in the buffer, source sub-level, then click OK.

4.7 Multi-Target Programming

Multi-target device programming

CPI2-B1 programmers can be used for concurrent programming multi-PCB panel assemblies with multiple identical boards, each of which may carry more than one programmable devices of different types (for example, two MCUs and a configuration EEPROM). To speed up the production by parallel flashing all programmable devices on the panel you can organize as many virtual programmer clusters as many devices are set on each boards; each cluster should want by running multiple copies of ChipProg-02 program. For example:

- A panel has four identical boards;
- Each board carries three devices of different types;
- Each device should be written with its own file.

Then you will need as many as 12 CPI2-B1 device programmers. A typical scenario of use:

1. Split 12 programmers in 4 groups by 3x CPI2-B1 programmers in each. Each group will independently and concurrently program three different devices one board. In a result, all 12 devices will be independently programmed in parallel.

2. Prepare a matrix of the CPI2-B1 programmers' serial numbers assigned to programming a particular target board and a particular device on each board. Connect the programmers to a USB hub or a LAN switch, connected to a PC.

3. Make tree programming projects - one for each target device. Save their **.upp** files that includes device types, file names and other options. It is supposed that you have preliminary debugged these projects on a CPI2-B1 programmer working in a single-programming mode.

4. Launch three copies of the ChipProg-02 program in the gang mode. In the command line of the startup dialog specify serial numbers of the programmers - four numbers per a project. The program itself will "connect" appropriate device programmers to appropriate USB or LAN ports and to appropriate target devices and load appropriate files to appropriate buffers.

5. Then place the first panel into the fixture and start device programming either by the ATE signal or manually by executing the Auto Programming command in the GUI. Then replace target panels upon successful programming of all 12 devices.

5 Integration with NI LabVIEW

The National Instruments LabVIEW™ (hereafter LabVIEW) is a popular graphical development environment that makes possible integration of a variety of design, production, and testing tools. CPI2-B1 programmers can be controlled by LabVIEW using two methods:

- ChipProg-02 **Command Line**;

- Application Control Interface (**ACI**).

Each method is described in a section below.

5.1   LabVIEW Integration Using Command Line

This is the most simple way to integrate ChipProg-02 with LabVIEW that involves two steps.
- Set up a programming session using ChipProg-02 user interface.
- Operate device programmer using LabVIEW user interface.

Here is an example:

1) Create a folder for controlling ChipProg-02 software from LabVIEW user interface, for example **C:\LabView\1**.

2) On Windows desktop make a copy of ChipProg-02 icon. Rename it for use exclusively with LabVIEW. The path to the program referred to by this icon is usually "**C:\Program Files\ChipProg-02 \x_xx_xx\UprogNT2.exe**", where the **'x_xx_xx'** is the version of ChipProg-02 software. Right-click on the icon, select **Properties**, **Shortcut** tab, and in the **Start in** field change path to **C:\LabView\1** as in the following figure:
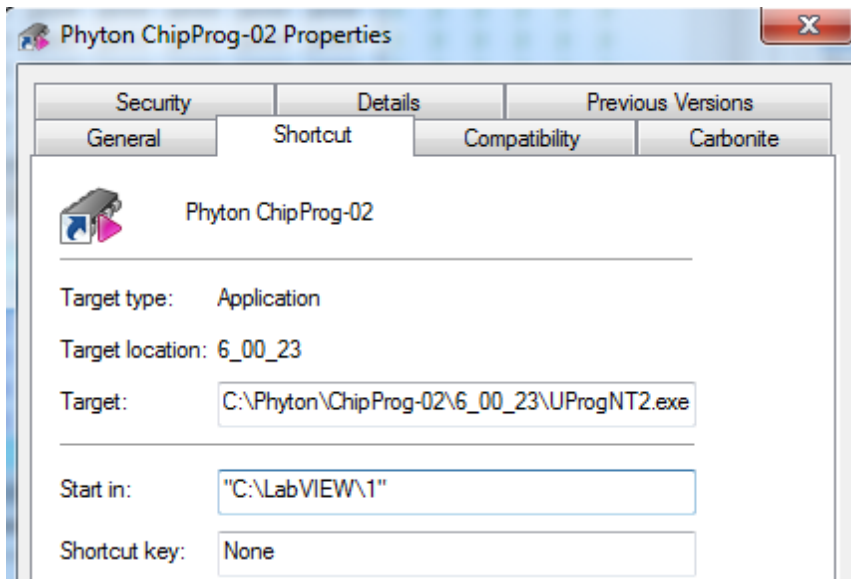
3) Power on CPI2-B1 device programmer, connect it to a USB port on your PC, and launch the ChipProg-02 program by clicking the icon in **C:\LabView\1** folder. When programmer user interface opens, start setting programming session options by choosing the target device (for example by pressing the **F3** hot key). After choosing the device set up programming options and parameters using ChipProg-02 windows, menus, and dialogs if these options **differ from default** ones. The following options can be set within the ChipProg-02 GUI:

- Settings in the **Program Manager** window, such as selecting functions to be included into the **Auto Programming** batch (button **Edit Auto...**); these include Split data, Insert test, Auto Detect, and other settings in the **Options** tab; the number of chips to be programmed during a programming session and other options in the **Statistics** tab.
- Settings in the **Device and Algorithm Parameters Editor** window that are device-specific, such as boot vectors, fuses, lock bits, Vcc voltage, oscillator frequencies, etc.

- Settings in the dialogs accessible via **Serialization, Checksum, Log file...** menu, such as algorithms for writing serial numbers and custom signatures into the devices being programmed, buffer checksum calculation, custom shadow areas, dumping data to log files, etc.
- Miscellaneous settings in the dialogs accessible via **Preferences** and **Environment** menus, such as color, fonts, sounds, etc.

Complete the definition of programming session by including appropriate **command line keys** into command line pattern:

- Specifying method of control through the programming session (key **/S**);
- Choosing target device (key **/C<manufacturer>^<device>**);
- Loading the file to be programmed and its format (key **/L<file name> /F<file format>**);
- Specifying the Auto Programming mode (key **/A**);
- Launching programmer in hidden mode, when the ChipProg-02 GUI is hidden (key **/I2**).

Notes:
- Device specified by the **/C** key on command line must be the same as chosen in the ChipProg-02 user interface.
- Specifying **/I2** key on command line hides ChipProg-02 application main window, suppresses display of error messages but copies them to the Windows clipboard. If the session terminates successfully ChipProg-02 application returns exit code 0; in case of errors exit code 1 is returned.

For example, if you want to program a HEX file **myfw1020.hex** located in the **Program Files (x86) \ChipProg-02\6_00_21** folder into the flash memory of a number of **NXP MK20N64VFT7 [ISP EzPort Mode]** devices, then the command line should have the following format:
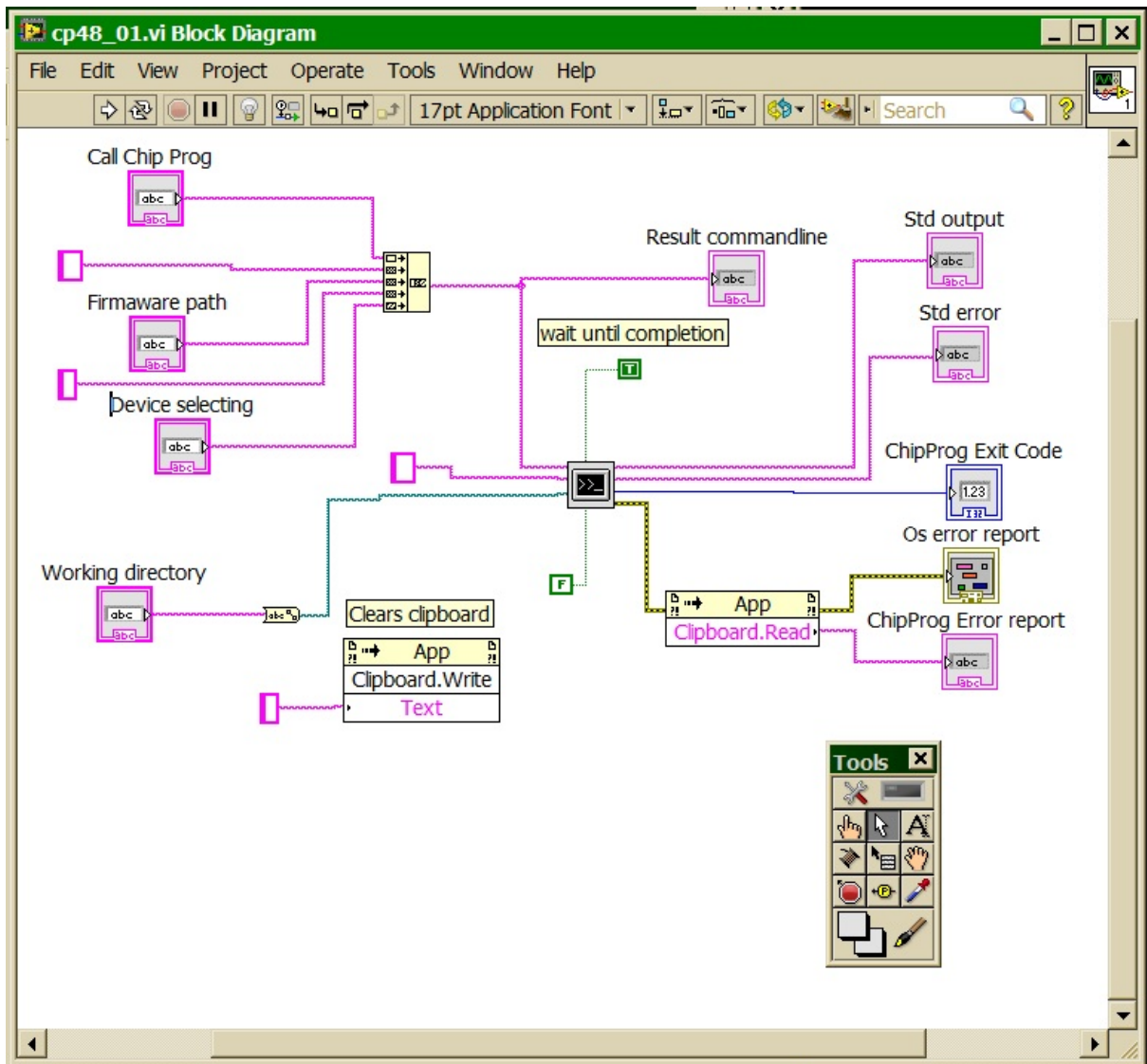
"C:\Program Files (x86)\ChipProg-02\6_00_21\UprogNT2.exe" /L"Program Files (x86)\ChipProg-02\6_00_21\myfw1020.hex" /FH /C"NXP^MK20N64VFT7 [ISP EzPort Mode]" /A /I2

4) To start CPI2-B1 in command line mode use the standard LabVIEW module **SystemExec.**

The figure below shows a screen shot of LabVIEW GUI front panel with the cp48_01.vi module loaded.
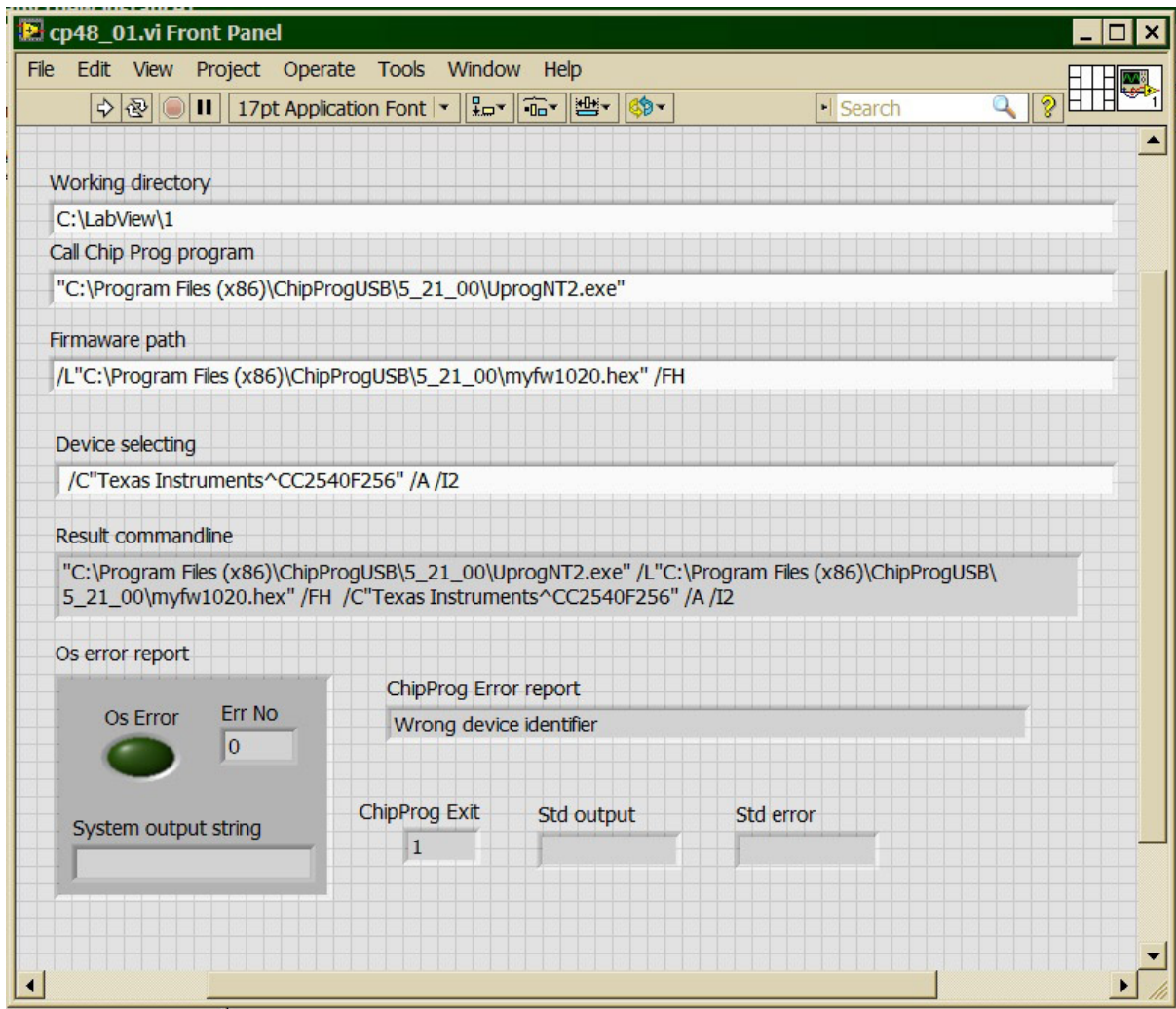
And below is the same module block diagram:

The <CPI2-B1 starts in hidden mode, its GUI remains invisible during the programming session. If no errors occur, the **ChipProg Exit** box returns exit code 0, otherwise exit code 1 is returned. The error is displayed in the ChipProg Error box report.

5.2   LabVIEW Integration Using ACI

The ChipProg-02 software package includes the **Virtual Instruments** (VI) library developed in the National Instruments' LabVIEW™ graphical development environment. It also includes a few usage examples of these virtual instruments. The library files reside in the LabVIEW folder located in the ChipProg-02 installation directory. The library is created using the 2013 SP1 version of LabVIEW.

The DLL control is based on use of the Application Control Interface. Each VI is a wrapper over the appropriate function exported by the ACI.DLL library. You should be quite familiar with the Application Control Interface in order to use the Virtual Instruments library.

Because of limitations imposed by LabVIEW on passing parameters to functions exported from DLLs, the virtual instruments do not call the ACI.DLL functions directly. Instead, they call functions exported

from the intermediate DLL - the ACI_LV.DLL. This DLL packs parameters into structures required by ACI.DLL and then calls its functions. The declarations of functions exported by ACI_LV.DLL are placed in the C/C++ header file named ACIProgLabVIEW.h.

Each virtual instrument has its own front panel. It allows calling an appropriate Application Control Interface function. In order to do this, before launching this function, you should launch the CPI2-B1 by means of the VI with the name ACI Launch. Each virtual instrument has input and output terminals for inputting and outputting parameters of the ACI function served by the virtual instrument.

The VI folder contains a sub-folder called Examples with two usage examples for virtual instruments. The "Device Programming Example" demonstrates use of all major ACI functions, namely:
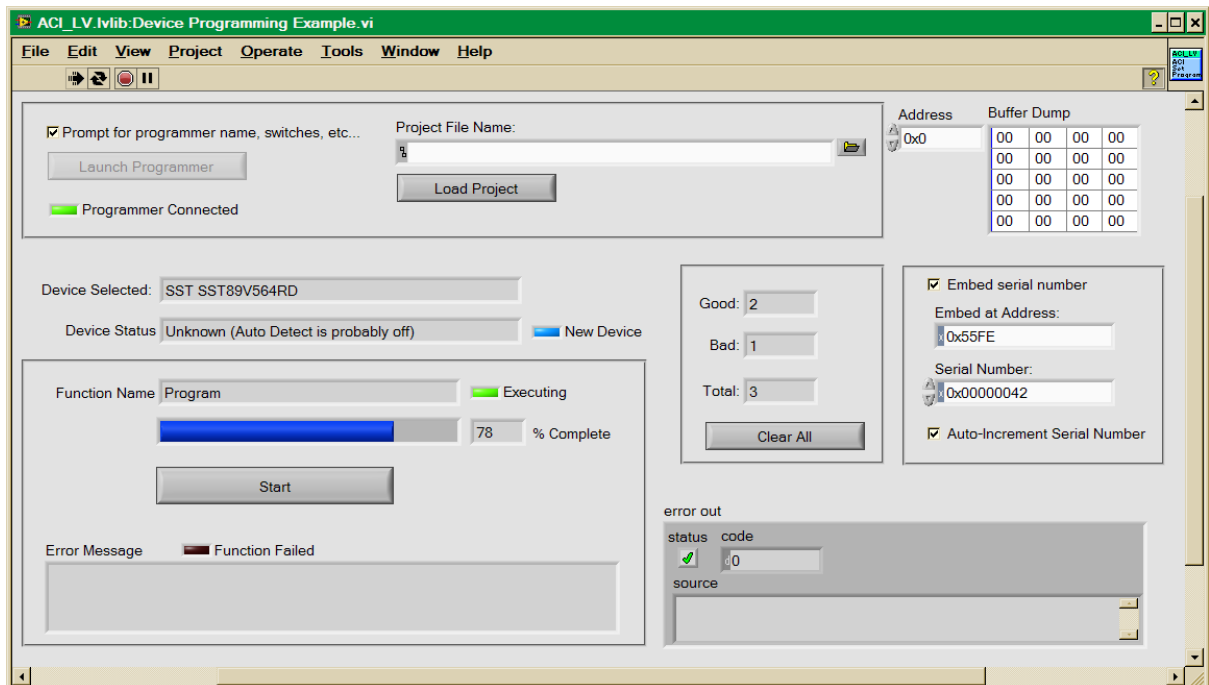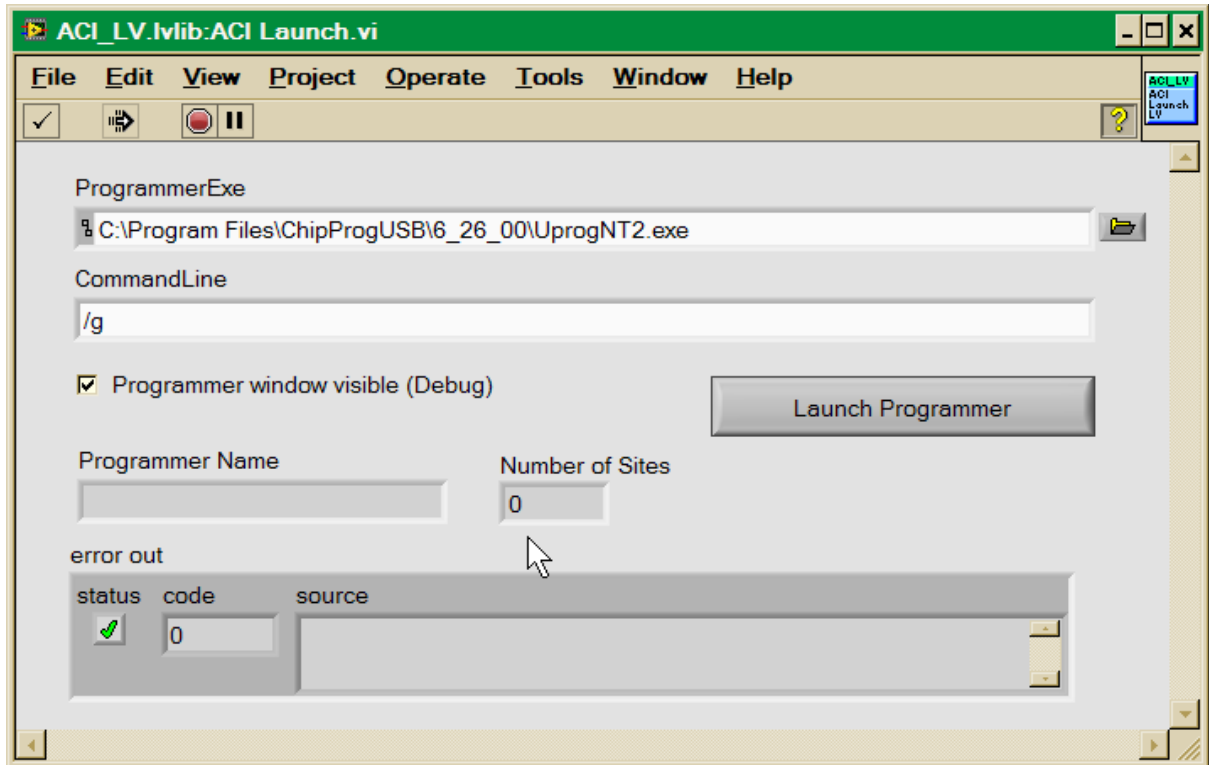
• launch a device programmer;
• load a project;
• display the device programmer buffer content in the GUI;
• display a chosen device in the GUI;
• display the device programmer socket's status (if a chosen programmer type supports this feature);
• write a serial number and increment it automatically in the device programmer buffer;
• perform programming functions on target device and display the results in the GUI;
• count numbers of successfully programmed and failed devices, and display them in the GUI;

To evaluate the example, start the CPI2-B1 and launch **Device Programming Example** by clicking **Run continuously** button in the LabVIEW GUI. Then click the **Launch Programmer** button on the VI front panel. This will open front panel of the virtual instrument **ACI Launch**. Enter full path to the ChipProg-02 executable file, for example: "C:\Program Files\ChipProgUSB\6_00_00\UprogNT2.exe" and (optionally) specify the command line parameters. To avoid prompts to restart programmer you can specify the path to the UprogNT2.exe in a constant string in the virtual instrument diagram and un-check the **Prompt for programmer name, switches, etc...** box on the front panel (see the diagram below).
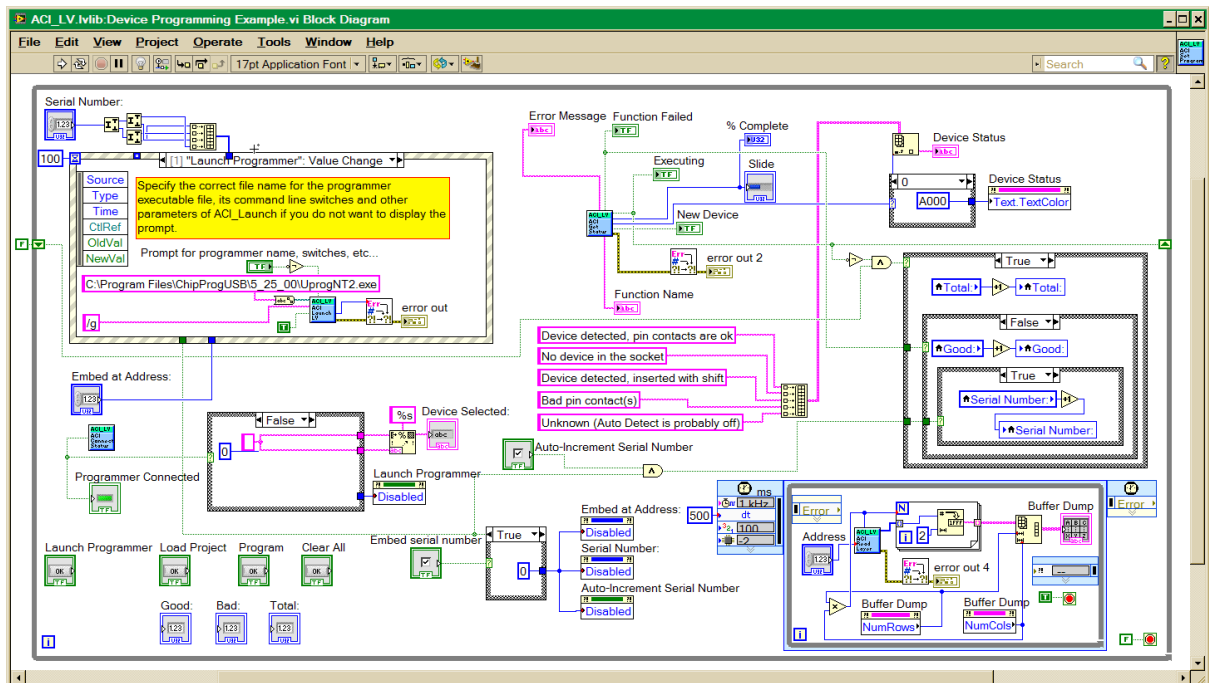
After launching the programmer its current status becomes visible in the virtual instrument's front panel.

Clicking the **Start** button launches the operation with the name that you can enter into the **Function Name** field, for example: **Blank Check**. If the **Function Name** field is left blank, the programmer will execute **Auto Programming** function. This process is illustrated in figures below.

6   Standalone Operation Mode

CPI2-B1 device programmers can operate in standalone mode that does not require control by a computer. This control mode provides the fastest CPI2-B1 control. It is especially convenient for mass production without human intervention.

6.1   Overview

CPI2-B1 device programmer contains internal memory card (or SD card) that can hold all information sufficient for maintaining programming sessions without further interaction with a PC. This CPI2-B1 mode of operation is called **standalone mode**. (The capacity of internal memory card is 8 to 64 GB; it can be upgraded by Phyton without prior notice.)

To configure standalone mode, CPI2-B1 operator puts the following information into the internal memory card:
- unique serial number of the device programmer;
- data to be programmed into target device;
- settings and programming options;
- pre-generated parameters for device serialization (optional).
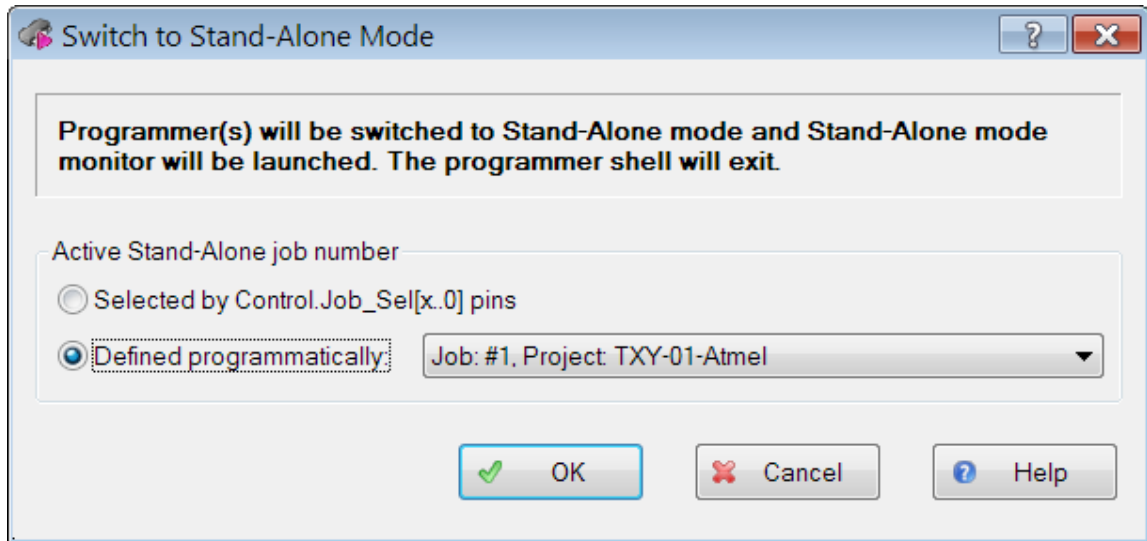
The above information, called Standalone Project, completely defines a programming session executed by CPI2-B1 programmer in standalone mode. A standalone project is quite similar to projects used for other methods of controlling ChipProg-ISP2 programmers. A standalone project may include multiple Standalone Jobs. CPI2-B1 memory card can store up to 255 independent jobs, provided the total volume of data to be programmed is does not exceed memory card capacity.

Operator prepares standalone jobs within the ChipProg-02 GUI. Further description assumes that standalone jobs were prepared in accordance with guidelines described below.

6.2   Switching to and from Standalone Mode

After power-up CPI2-B1 device programmer enters idle mode. It can then be switched into either standalone or computer controlled mode. Switching a CPI2-B1 programmer into the standalone mode can be done in one of the following ways:

- Using ChipProg-02 GUI, by selecting menu command **Commands -> Standalone Mode** (see figure below).

Switch to Stand-Alone Mode

Programmer(s) will be switched to Stand-Alone mode and Stand-Alone mode monitor will be launched. The programmer shell will exit.

Active Stand-Alone job number

○ Selected by Control.Job_Sel[x..0] pins

● Defined programmatically: Job: #1, Project: TXY-01-Atmel ▼

✓ OK   ✗ Cancel   ❷ Help

- By launching a Standalone Mode Monitor program,
- By applying a logical 1 signal to the SAMODE pin of CONTROL connector at a moment of the programmer powering.
- By applying a logical 0 signal to the Start pin of CONTROL connector for at least 2 seconds.
- By pressing and holding Start button for at least 2 seconds.

Once CPI2-B1 switches to standalone mode, the green (GOOD) and red (ERROR) LEDs start blinking. These LEDs will keep blinking until the programmer is switched to computer controlled mode. When programmer is in standalone mode, a standalone job can be launched by either applying logical 0 to the Start pint of CONTROL connector for at least 1 ms, or by pressing Start button on the programmer.

ChipProg-02 software allows real-time monitoring of CPI2-B1 device programmer activity (or multiple programmers running in the gang mode). Special utility, Standalone Mode Monitor, is provided for such monitoring . Monitor displays status of device programmer (if multiple CPI2-B1 device programmers if running in gang mode, statuses of multiple programmers) along with current Standalone Job number, device counters, statistics of failures, and other useful information.

To interact with ATE or other equipment, CPI2-B1 device programmer running in the standalone mode provides logical signals Good, Busy, Error (log. 0 means active) on corresponding pins of the CONTROL connector. These signals are duplicated by Green, Yellow and Red LEDs on the top surface of CPI2-B1 unit.

If a session involves programming of different data into two or more devices of different types, by means of the same CPI2-B1 programmer, standalone jobs must be switched by external ATE or other equipment. For this purpose, the CONTROL connector contains six pins (Job_Sel [5..0]) used to select a standalone job. For example, if Job_Sel code = 000001B the programmer will run the Job #1, if the the code = 100001B -- Job #33 (or 21H). When no electrical signals are applied to these pins, Job #0 will be selected.

When multiple CPI2-B1 device programmers run in gang mode, ChipProg-02 program takes care of synchronizing Standalone Mode Jobs in all device programmers in the gang cluster.

6.3 Preparing Standalone Mode Projects

Use CPI2-B1 device programmers in the Standalone Mode requires using Standalone Mode Projects. These projects store the file names and paths, device part numbers, programming options, serialization parameters, etc. Operator prepares standalone mode jobs using the ChipProg-02 Graphical User Interface (GUI).

To create a standalone project, first a regular project must be created by using menu command **Project > New**.

6.3.1 Data Caching

Data Caching is a process of copying data to CPI2-B1 device programmer internal memory card by the ChipProg-02 program. After caching has been performed,
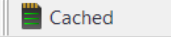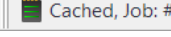- the programmer uses data from internal memory card rather than from a computer. Copying data from memory card to the target device is significantly faster than copying it from a PC;
- if a project is open within the ChipProg-02 application, data caching automatically copies to memory card all information necessary to run this project in the standalone mode.
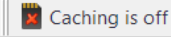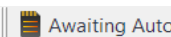
Data Caching function is off by default. To increase programming speed, caching data on memory card is only useful with target devices that have large capacity.

Data Cashing status is displayed in the ChipProg-02 main window toolbar:
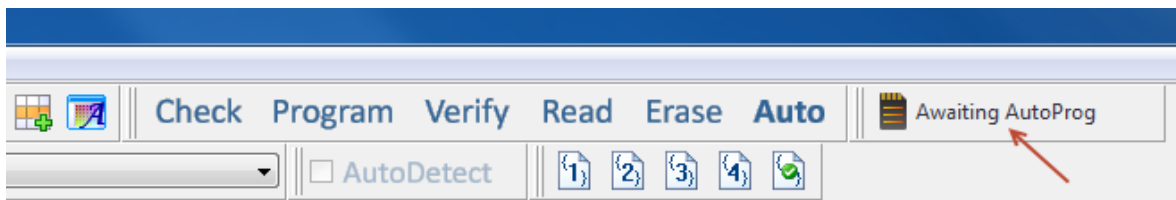
Data caching states are:

No SD card(s)	Either programmer type used is not CPI2-B1, or SD card is not present or is malfunctioning. Data caching is not possible.
Caching is off	Data caching is turned off.
Awaiting AutoProg	Data caching ready. To perform caching, start Auto Programming of the target device.
Cached	Data caching complete. subsequent programming operations will use data on SD card. If a project is open, it is ready to be prepared for standalone mode.
Cached, Job: #2	Data caching complete. Project is ready for standalone mode and is assigned to standalone job with selected number.

To bring up settings for caching, standalone jobs, and serialization, click on the image of caching status, or use the command **Configuration -> Data Caching, Standalone Jobs... .**
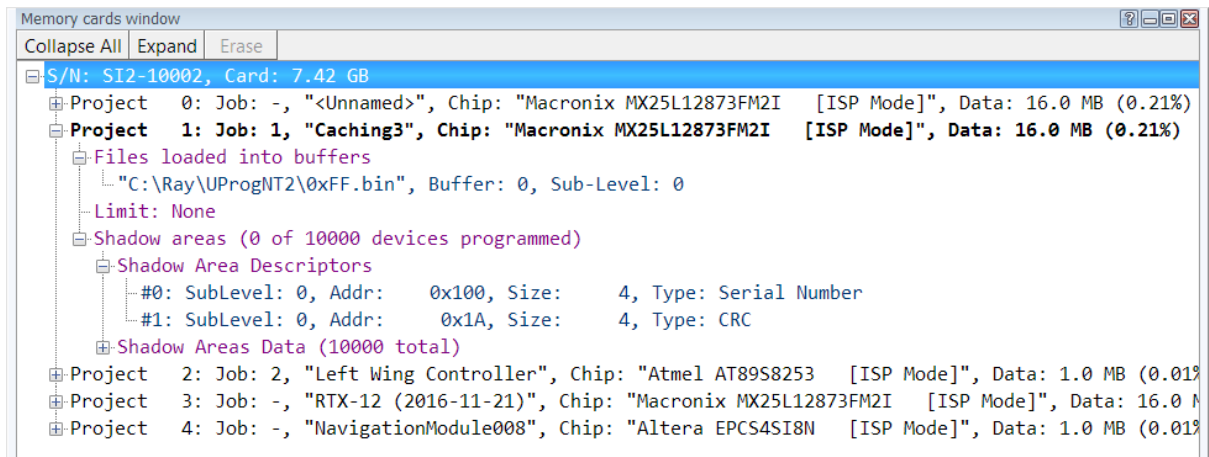
How Caching is Performed

Before launching, data caching must be enabled in settings. Caching happens during the first automatic programming of a target device. The ChipProg-02 program records all buffer layers read operations performed by programmer.

After automatic programming has finished, SD card will contain cached data which will be used for subsequent programming operations. SD card can hold up to 255 projects, provided the size of all buffer layers does not exceed SD card capacity.

Projects stored on SD card are assigned numbers. The program uses the following rules to determine the number assigned to a project when storing it on SD card.

- If there is a project open in ChipProg-02, then a project on SD card with the same name is looked up; if such project exists, it is overwritten. If there is no project with this name on the card, the first unused number is assigned. If there are no unused numbers, the project is assigned the first lowest number used by an unnamed project. If there are no unnamed projects, an error is reported.
- If there is no open project in ChipProg-02, then the cached data are considered to be an "unnamed project." (project 0 in the figure). First, the program checks if an unnamed project with these data already exists on SD card, and if yes, the procedure stops. Otherwise, the unnamed project is stored on SD card and is assigned the first available number. If no numbers are available, the data are written into the oldest unnamed project and its number is reused. If there are no such projects on the card, an error is reported.

SD Card Window can be used to examine information stored on the card, as shown on the figure below.

```
Memory cards window                                                          ? _ □ X
Collapse All | Expand | Erase
⊟ S/N: SI2-10002, Card: 7.42 GB
  ⊞ Project   0: Job: -, "<Unnamed>", Chip: "Macronix MX25L12873FM2I    [ISP Mode]", Data: 16.0 MB (0.21%)
  ⊟ Project   1: Job: 1, "Caching3", Chip: "Macronix MX25L12873FM2I    [ISP Mode]", Data: 16.0 MB (0.21%)
    ⊟ Files loaded into buffers
       └ "C:\Ray\UProgNT2\0xFF.bin", Buffer: 0, Sub-Level: 0
    ┈ Limit: None
    ⊟ Shadow areas (0 of 10000 devices programmed)
      ⊟ Shadow Area Descriptors
         ┈ #0: SubLevel: 0, Addr:    0x100, Size:    4, Type: Serial Number
         └ #1: SubLevel: 0, Addr:     0x1A, Size:    4, Type: CRC
      ⊞ Shadow Areas Data (10000 total)
  ⊞ Project   2: Job: 2, "Left Wing Controller", Chip: "Atmel AT89S8253    [ISP Mode]", Data: 1.0 MB (0.01%
  ⊞ Project   3: Job: -, "RTX-12 (2016-11-21)", Chip: "Macronix MX25L12873FM2I    [ISP Mode]", Data: 16.0 M
  ⊞ Project   4: Job: -, "NavigationModule008", Chip: "Altera EPCS4SI8N    [ISP Mode]", Data: 1.0 MB (0.01%
```

During subsequent programming operations, the programmer uses buffer layers data from SD card. ChipProg-02 application tracks changes in the settings that can cause modification of data on the card, and if need launches data caching again. Actions that will trigger re-caching are:

- writing data into buffer memory by reading in a file, manually using dump window, by a script or using ACI;
- change in target settings;
- change in serialization settings;
- change in parameters of automatic programming etc.

6.3.2 Projects and Jobs

Because of data caching, internal SD card in the programmer will store a set of projects that can be used in standalone mode. ChipProg-02 application automatically assigns a number to each stored project.

When using programmer in standalone mode, logic signals on Job_Sel [5..0] pins of the CPI2-B1 CONTROL connector are used to select a *standalone job*. The number of such job is uniquely defined by these signals. Each job is simply a reference to a project number. Assignment of specific projects to job numbers is performed using the standalone mode settings dialog, as shown in the figure.
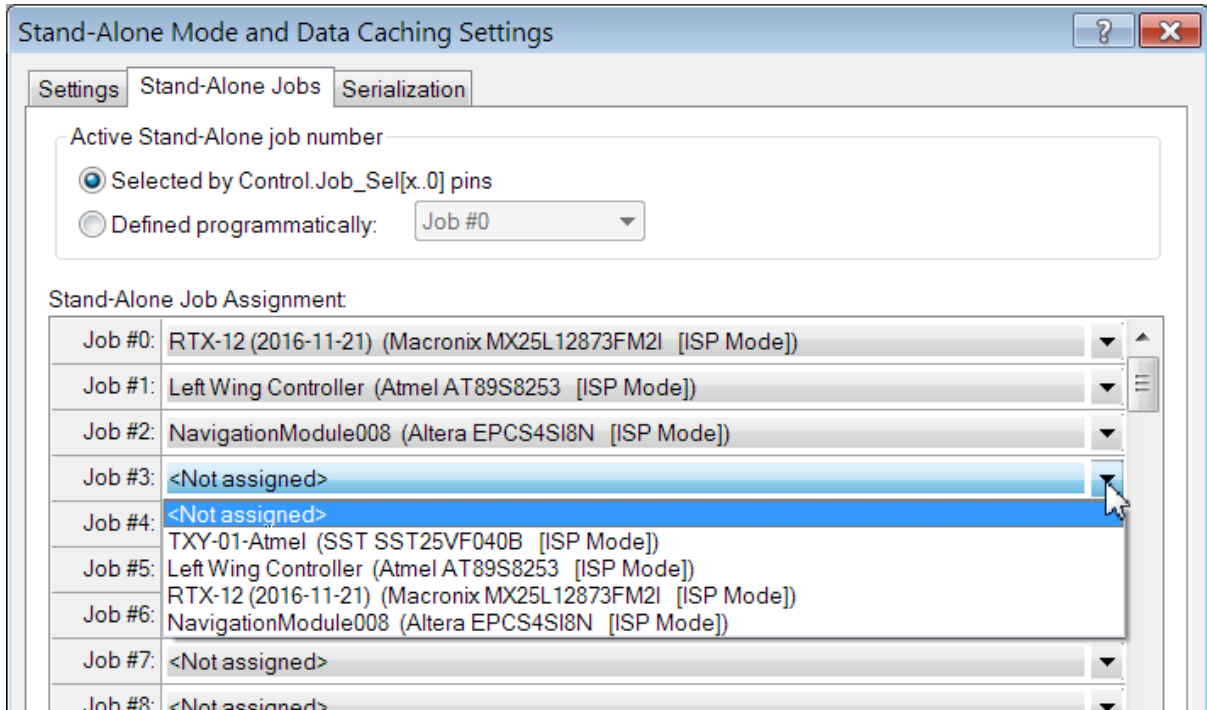
Only a named project can be associated with with an autonomous job. Each project can only be associated with a single job. The list of projects available for association contains projects stored on programmer internal SD card.

When using programmer in standalone mode, a job can be selected by using logic signals on Job_Sel [5..0] pins of the CPI2-B1 CONTROL connector or using Standalone Mode Monitor. A job can also be selected when using ChipProg-02 application to switch programmer to standalone mode.

6.3.3 Device serialization

Standalone mode allows writing into target device generated serial numbers, checksums, and other information. When using computer-controlled mode (GUI), serial numbers and other parameters of serialization can be managed using serialization settings brought up by menu command **"Configuration" -> "Serialization, Checksum, Log."** Each project has its own serialization settings. These settings must be done *before* generating serialization information for standalone mode. See figure below.
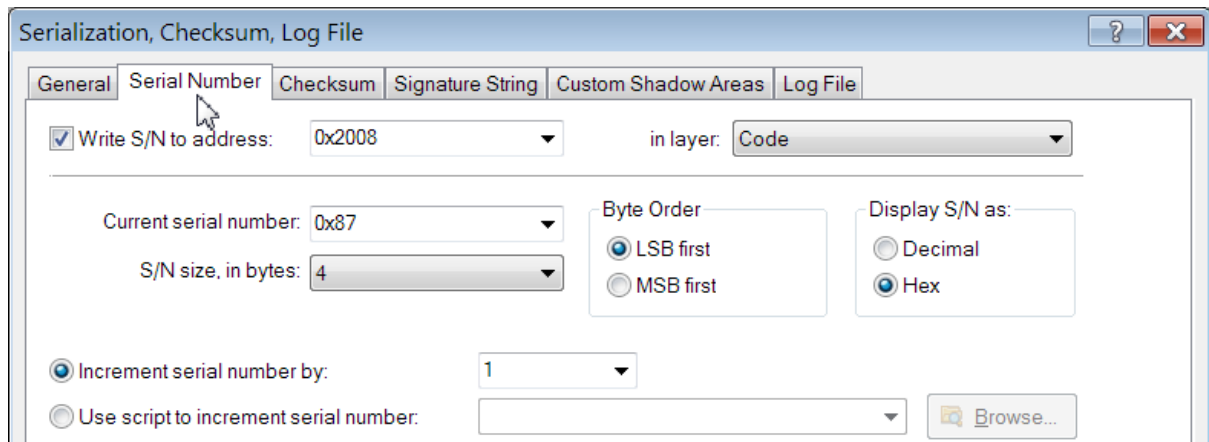
Serialization information for a project must be generated beforehand. Settings that control generation can be done in a dialog brought up by clicking on the image of serialization status, or by menu command **"Configuration" -> "Data Caching, Standalone Jobs..."** as shown of the figure below.
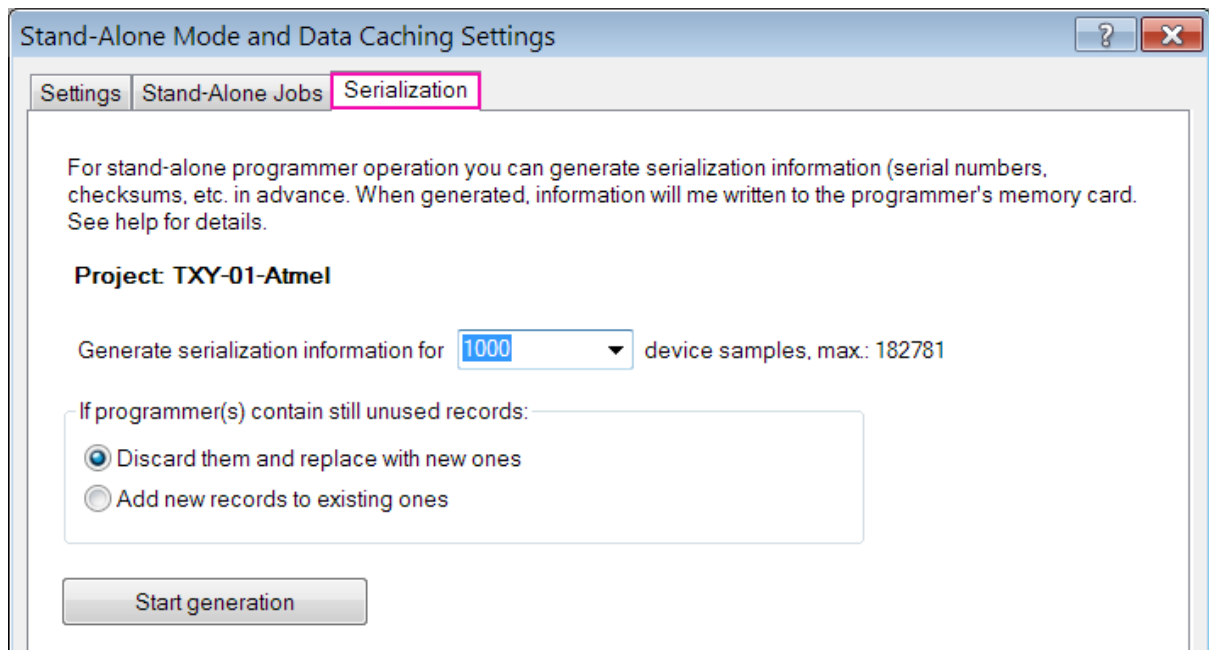
Serialization information is stored in a fixed portion of SD card memory. The maximum number of target devices that can be accommodated depends on what information is generated. In the figure the maximum number of devices is 182781.

When operating in standalone mode, programmer fetches serialization records one by one, and programs them into target devices. The number of the next record to be fetched is preserved even if the programmer is powered off. Once all records have been programmed, the programming process is terminated and an error occurs. To continue programming process, additional serialization info must be generated.

If unused records remain in the programmer, a dialog can be used to choose how they should be dealt with. If new records are added to existing ones, serial numbers of target device will be consecutive.

Serialization records can only be generated after the data has been cached. This is because before caching the amount of SD card memory available for storing serialization information cannot be known.

> ***To perform generation, the button "Start Generation" must be clicked. The OK button at the bottom of the dialog does not start generation.***

In the gang mode, serialization records are equally distributed among all CPI2-B1 device programmers in the gang cluster since it is assumed that all these programmers are simultaneously writing target devices.

Current serialization information can be viewed in the Memory Maps window; in this window serialization records are called "shadow areas" (which they actually are), as illustrated in the following figure.
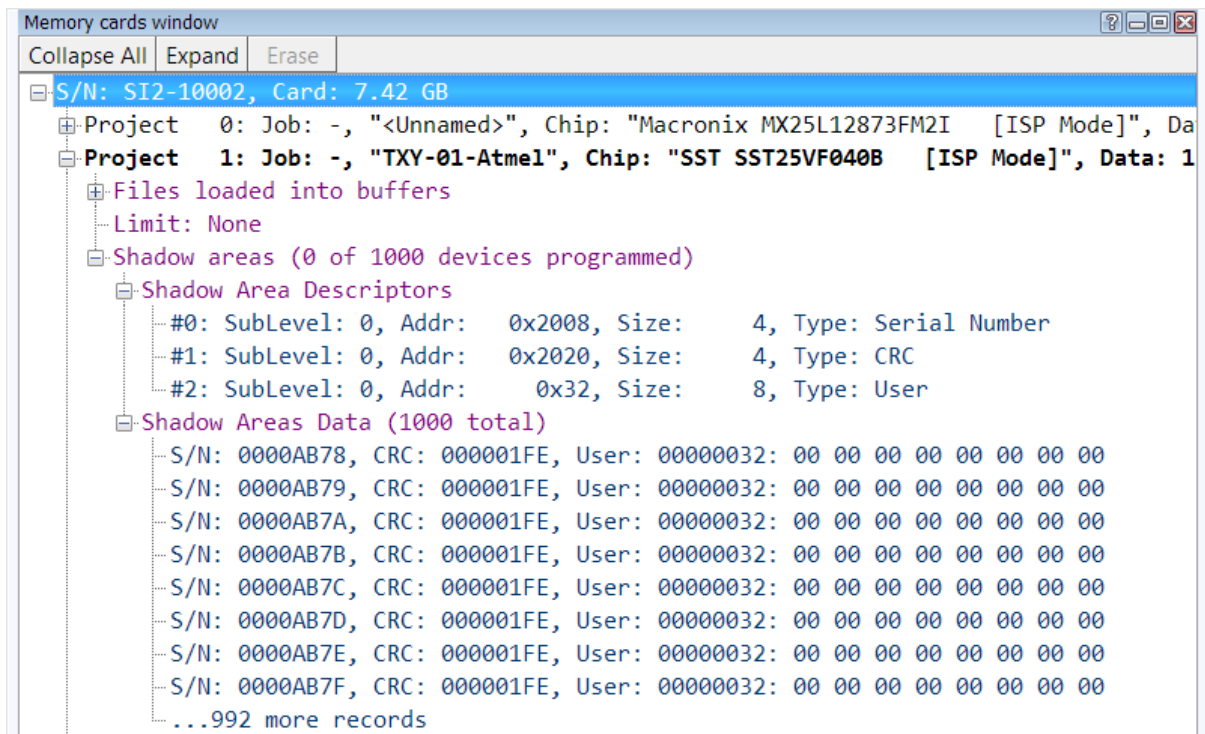
Limitations of Serialization in Standalone Mode

Besides obvious limitations related to the need of periodically add in serialization records, the following should be kept in mind:

- If programming of a target device ended up in an error, serialization record is still used up, in spite of settings in the application program. In such case serial numbers of targets will not be consecutive - it will consist a gap.

- If scripts are used for generation of serial numbers, checksums, etc. it is important to realize that in the GUI control mode scripts launch immediately *before* programming of the next target device. However, when generating records for standalone mode, scripts launch immediately *after* generation of the next record. If a script includes some real-time related parameters, such script will not work correctly. If the scripts modify the data to be written into target device, that is not going to work either.

6.3.4 Permissions and setting limits

A limit can be set on the number of target devices to be programmed in standalone mode. This may be useful to physically control the quantity of devices.

The limit is specified in project settings by clicking on "Permissions" button in the dialog, as shown in the figure.
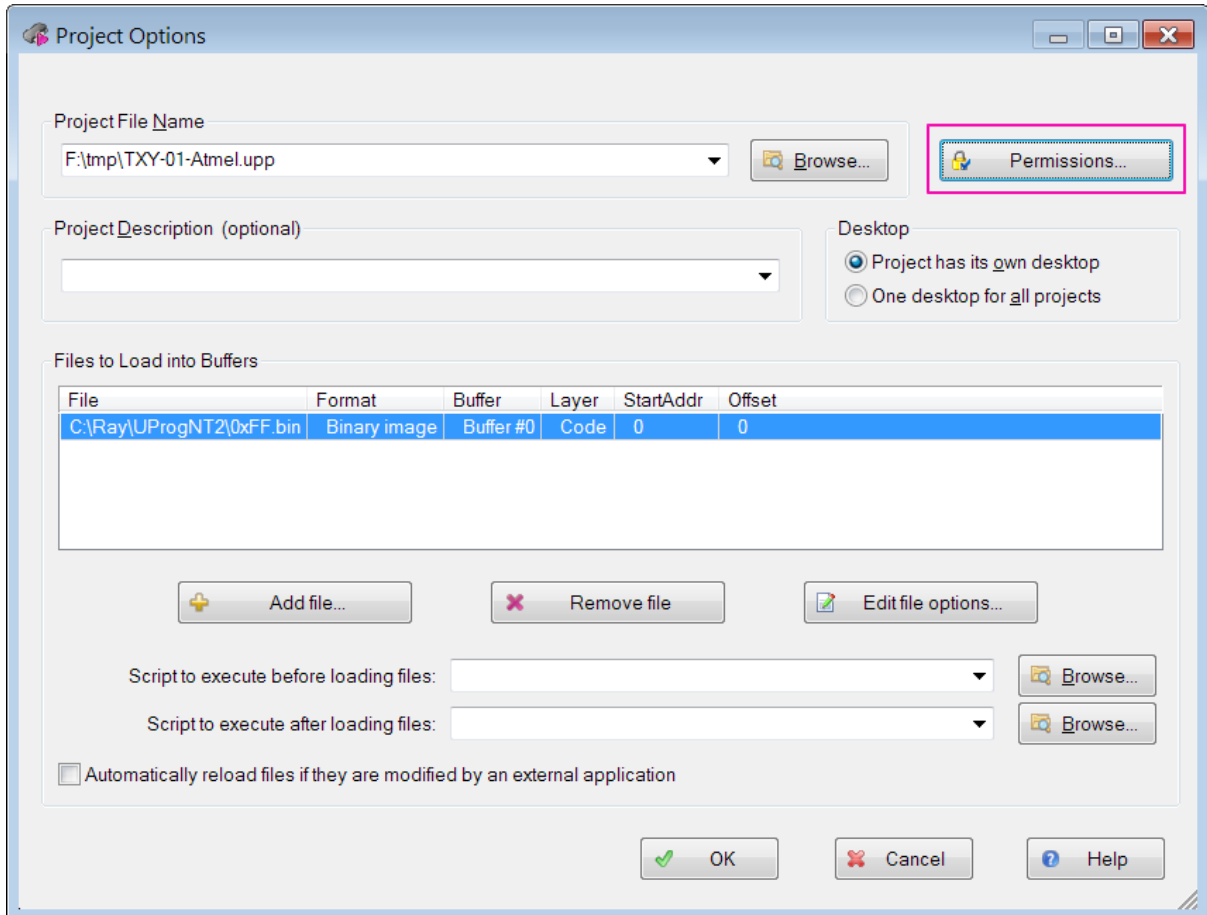
The button brings up permissions dialog in which you can specify the number of devices to be programmed. (The option of protecting the project with a password must be turned on.)
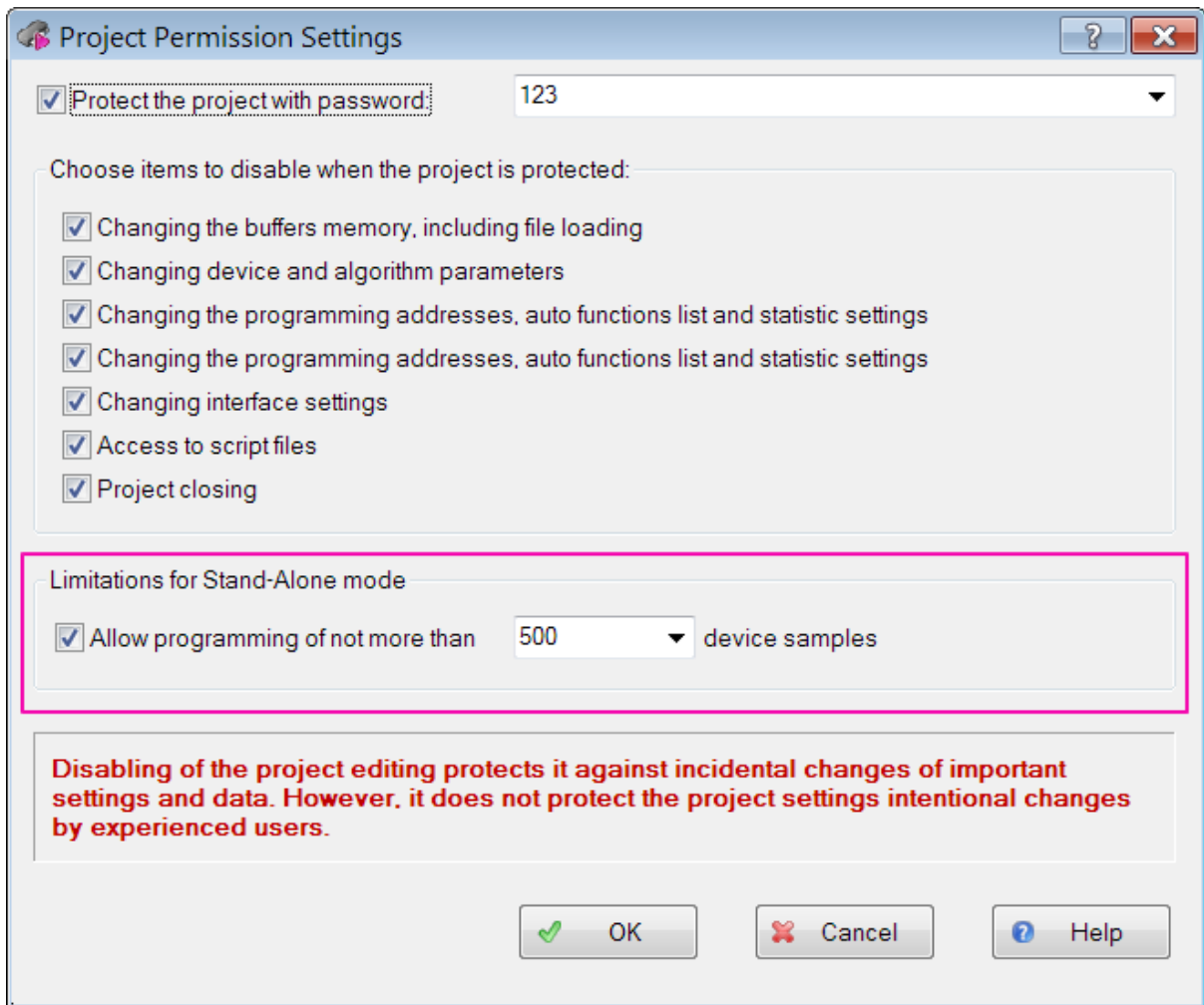
Current state of limitations counter can be monitored in the **Memory Map** window.
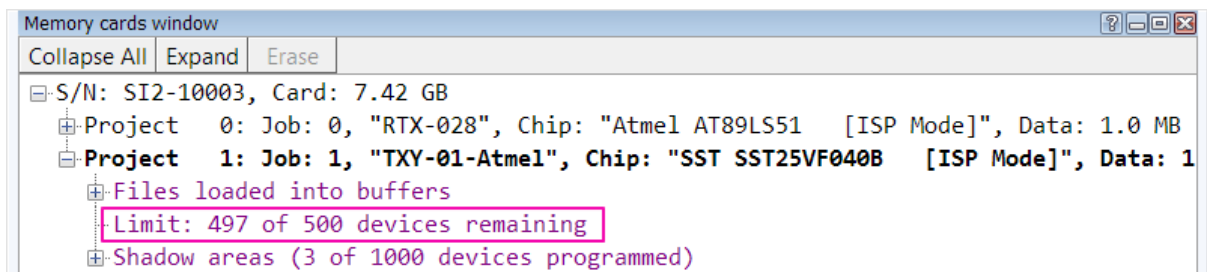
Once the limit was achieved ChipProg-02 issues an error warning and the programming stops. To continue programming, it is required to confirm or remove limitation using **Project Permission Settings** dialog.

6.4 Standalone Mode Monitor

Standalone Mode Monitor is an application program for watching the states of programmers operating in standalone mode. The application also can perform certain operations with programmers.

The application can be launched in the following two ways:

- by clicking on its icon in the launcher application as shown on the figure below:
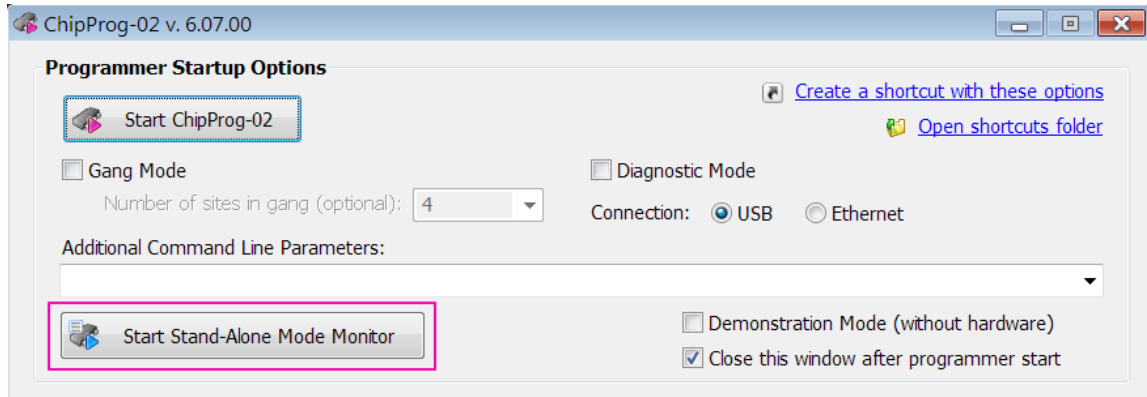
- by using ChipProg-02 GUI application menu command "**Commands" -> "Switch to standalone mode.**"

When launched, **Standalone Mode Monitor** switches all programmers it can communicate with to standalone mode, unless they are in that mode already. The Monitor can "see" only those programmers which are not being used at the moment by ChipProg-02 application; this is because at any given time a programmer can only be used with one controlling application. On the other hand, ChipProg-02 application does not "see" programmers operating in standalone mode.

Standalone Mode Monitor does not affect operation of the programmers, it only displays their current state, as shown below.
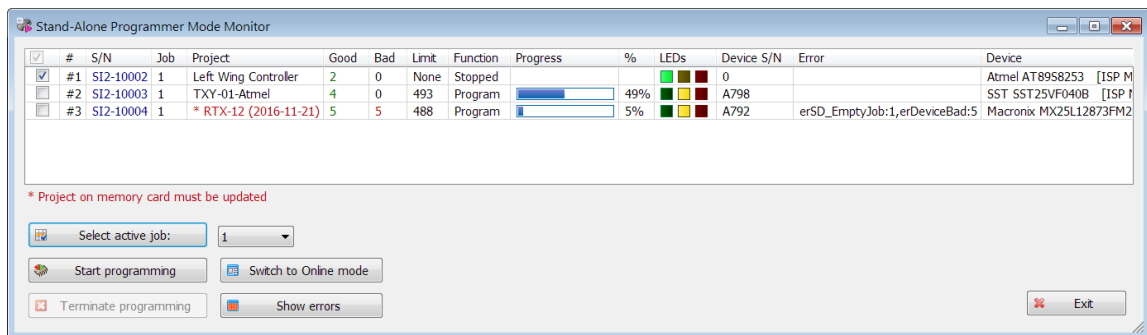
Information is displayed as follows.

#	Programmer number in the list.
S/N	Programmer serial number.
Job	Number of the active standalone job.
Project	Name of the project whose data is being written to target device.
Good	Counter of successfully programmed targets. This counter is reset to zero when programmer is powered off or when a different job is selected.

Bad	Counter of targets programmed unsuccessfully.
Limit	Number of targets remaining before the limit count is achieved. The count is set in project settings. Limit counters are preserved in powered off stat of programmer.
Function	Name of programming function being performed currently.
Progress	Indicator of function execution process.
%	Percent completeness of the function.
LEDs	State of programmer LEDs.
Device S/N	Current target device serial number, if it is defined in the standalone mode serialization settings.
Error	Error codes with error counts. Programmer keeps up up to 8 types of errors.
Device	Target device selected for the project. This information is shown last as least priority.

The Monitor window size can be changed in the same way as sizes of other windows.

All buttons in the dialog affect programmers marked by checkboxes.

Select Active Job: For selected programmers, set active job number to the number selected in the list to the right of the button. Selecting a job number by itself does not activate the job; clicking the button activates it. If selected job is not associated with any project, attempt to start programming aborts with an error **erSD_EmptyJob**. The button is enabled only if all selected programmers are stopped.

Start Programming: Start target device programming on selected programmers that are currently in the stopped state.

Interrupt Programming: Abort all target device operations on currently selected programmers. Completing of this command can be delayed for a while.

Switch to Online Mode: Switch selected programmers into online (computer-controlled or GUI) mode. This may be used to allow programmers to be controlled by ChipProg-02 application (the application does not "see" programmers operating in stand-alone mode). Once programmers are switched into online mode, Monitor is no longer able to communicate with them. For the Monitor to re-establish communications it has to be restarted.

Show Errors: Show table of errors for all selected programmers. Error counters are reset to zero when programmer is powered off. When switching active job, error counters are not reset.

If a project name is displayed in red, it means that the project data was written by an earlier version of application and have to be refreshed. In many cases this is critical, because with new versions of programmer software its firmware is also updated. To refresh a project launch the ChipProg-02 application, load the project into it, make sure data caching is turned on, and run Auto Programming once. This will refresh the project in programmer SD card.

6.5   Example of Setting Up Standalone Mode

Let's take a look at the minimal set of actions needed to prepare programmer for standalone mode operation.

- Target device: Atmel AT89LS51 [ISP Mode].
- File C:\Work\Monitors\RTX-028.hex (in standard hex format) has to be loaded into memory buffer.
- A 32-bit serial number has to be written into each target device at address 0x200. Serial numbers are increased by 1 for each device.
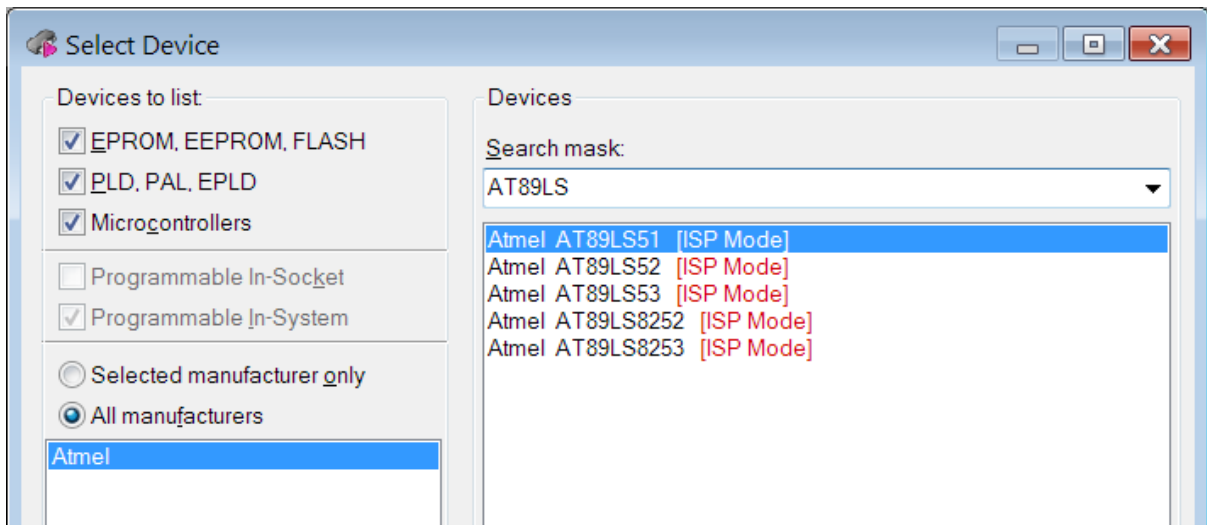
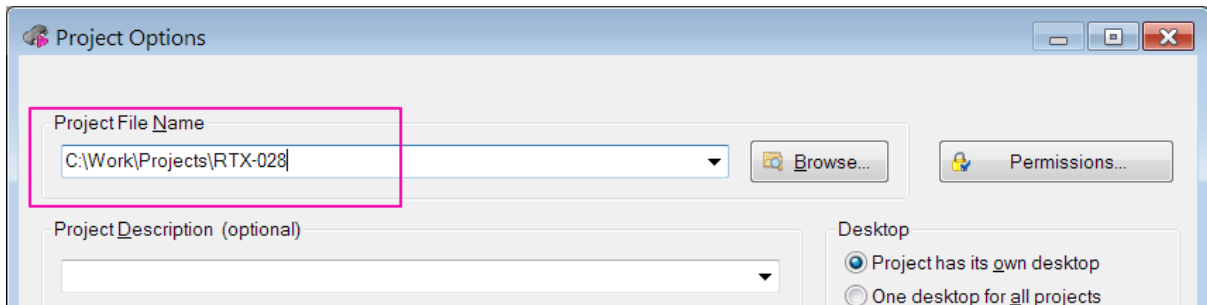Connect programmer to a computer and start application using icon:
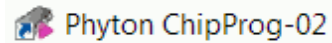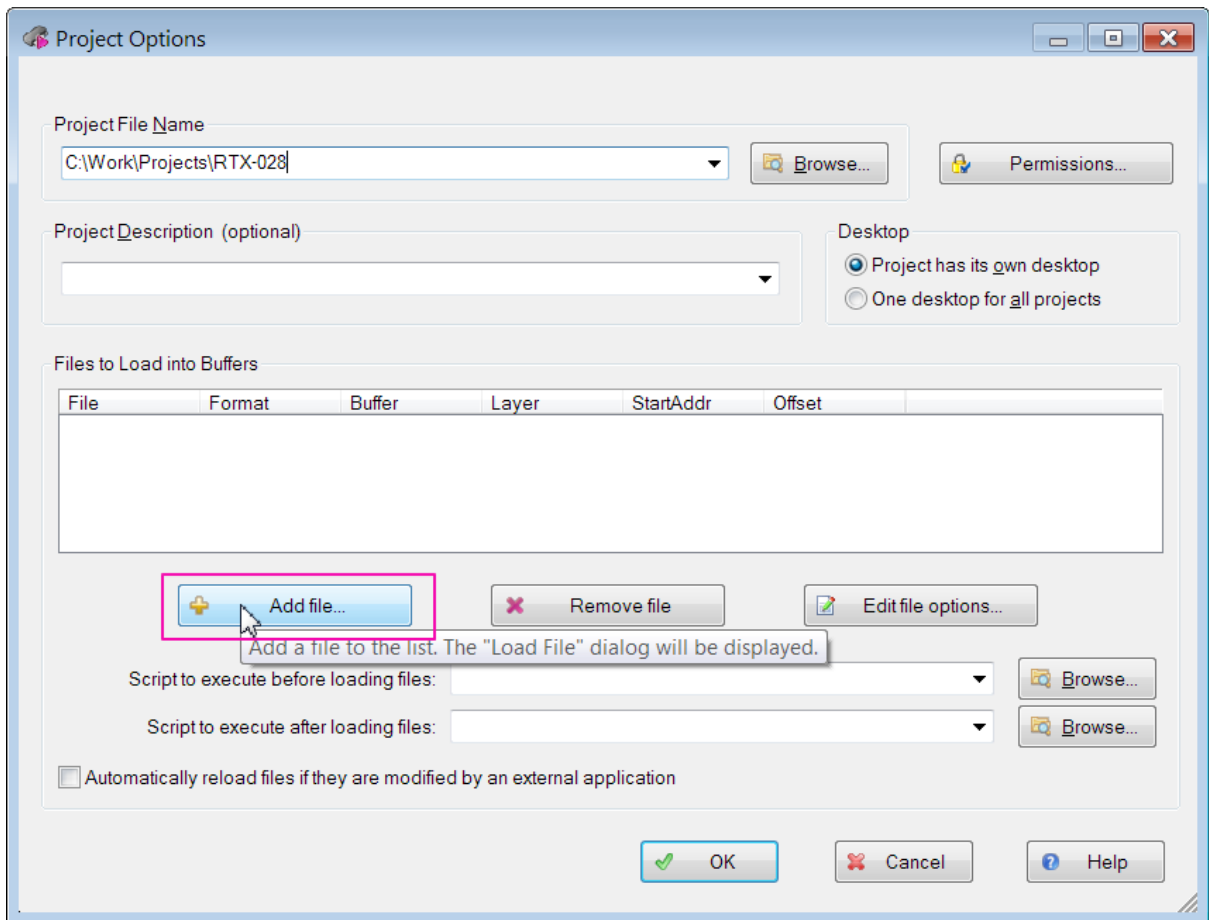
Click on "Select device" button:
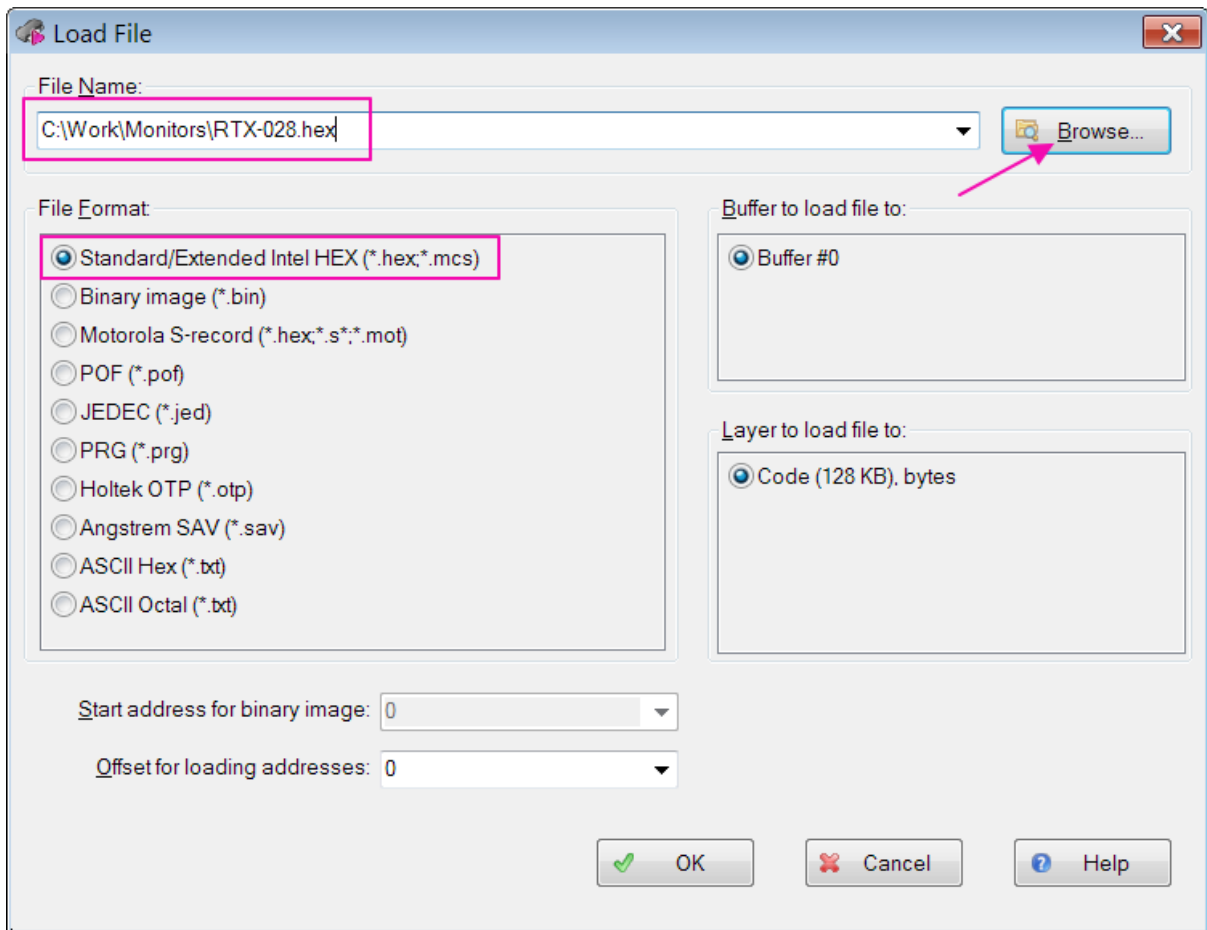

Select device type Atmel AT89LS51   [ISP Mode]:

Execute menu command **Project -> Create New**:

This brings up project creation dialog. In the field "**Project file name**" enter the name of the project file. Alternatively, click on **Browse** button and select folder and file using standard Windows dialog:

Select file C:\Work\Monitors\RTX-028.hex to be loaded:

In file selection dialog enter C:\Work\Monitors\RTX-028.hex, or use **Browse** button. Select "Standard/ Extended Intel HEX":
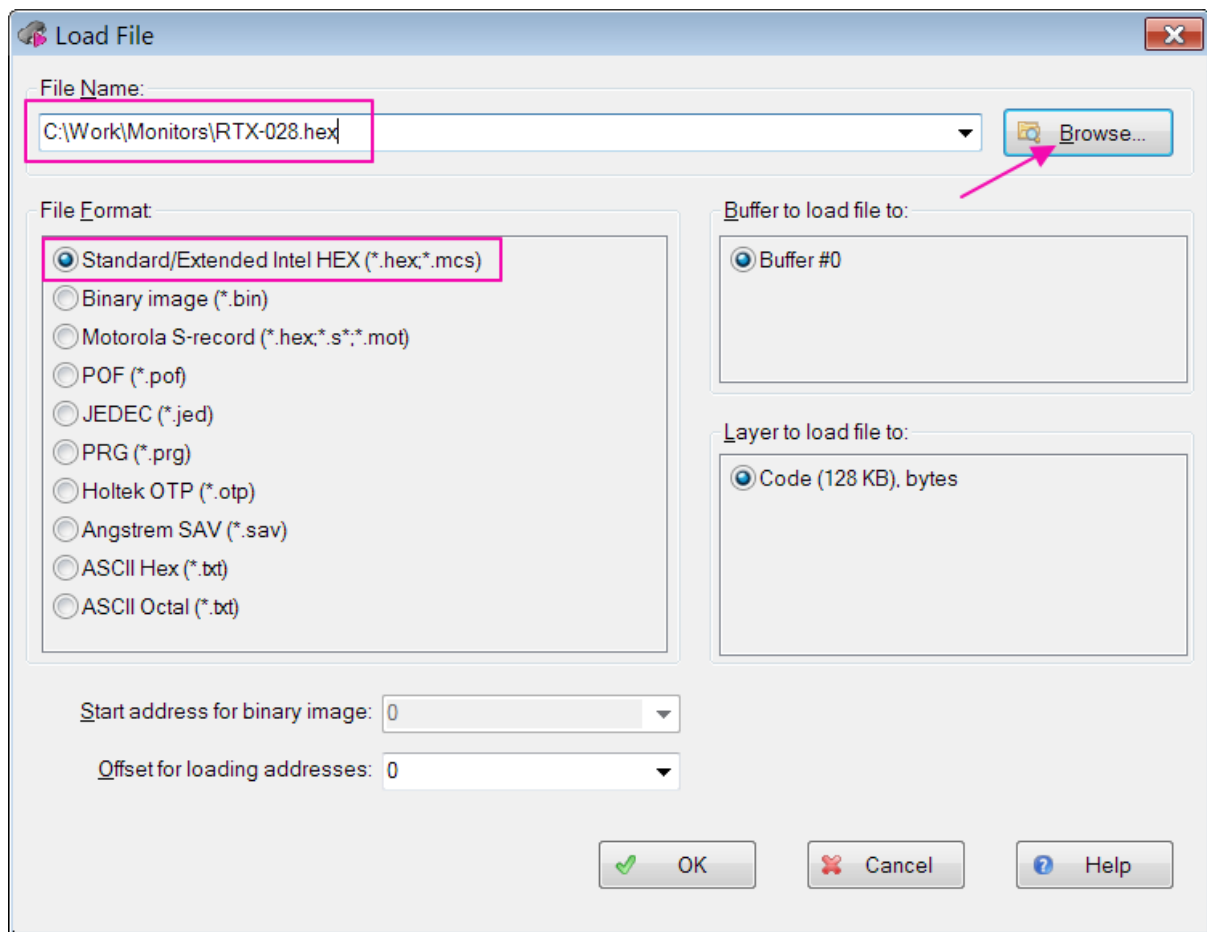
Confirm file selection by clicking OK, and the settings dialog will show the name of selected file. Confirm project settings by clicking OK; the project will be saved in the selected file C:\Work\Projects\RTX-028.upp. If folder :\Work\Projects does not exist, the program will ask if it should be created.
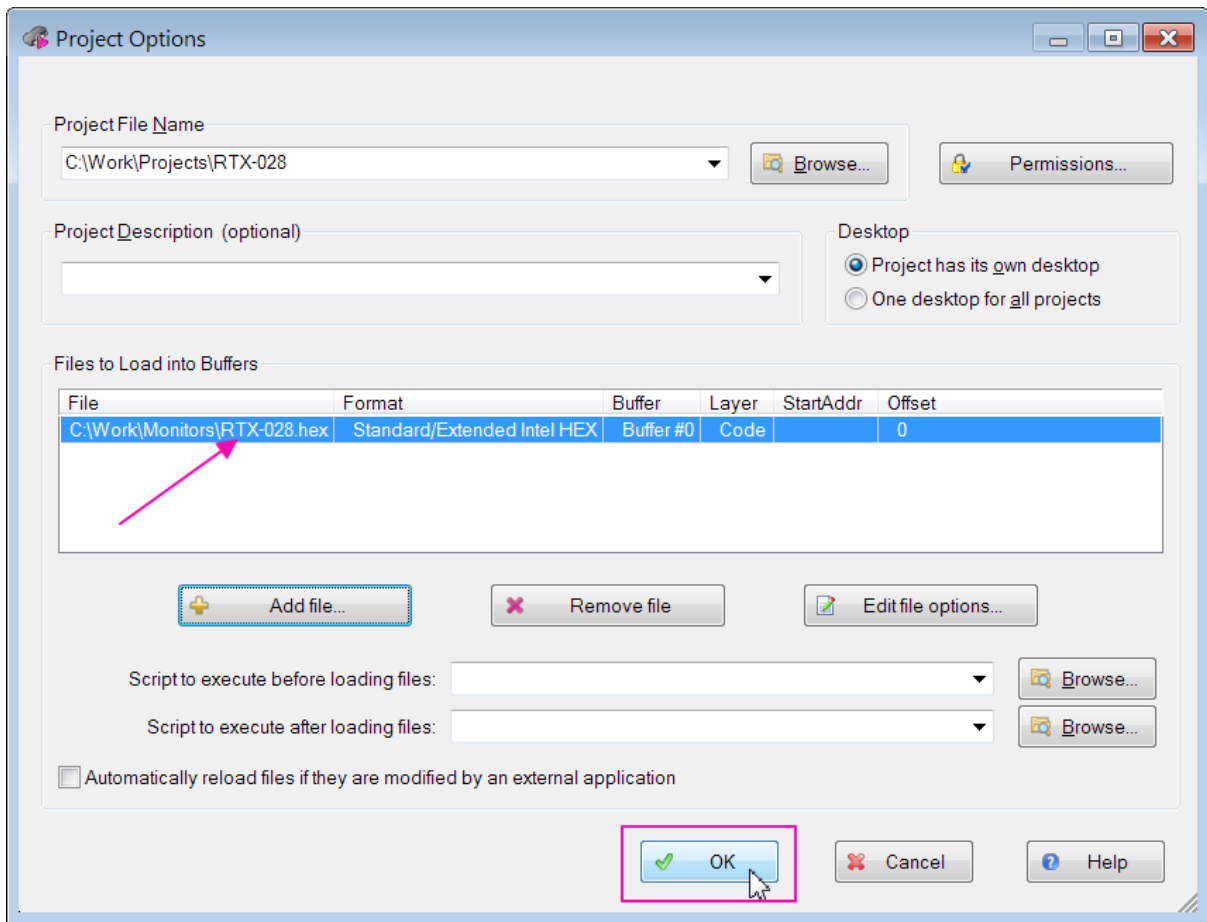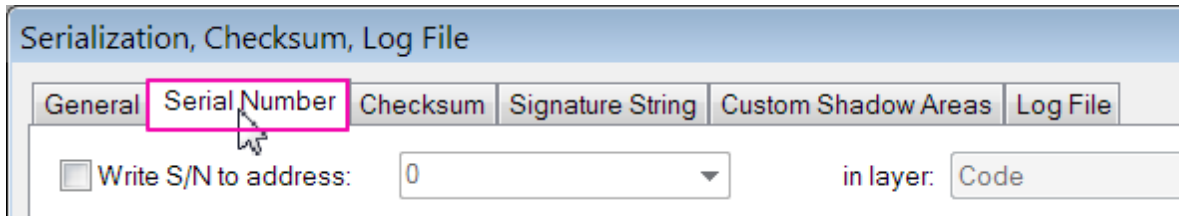
Now we are working with a project, as shown in the window title:
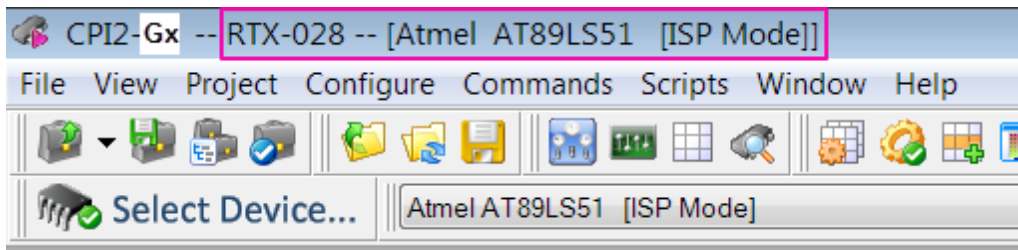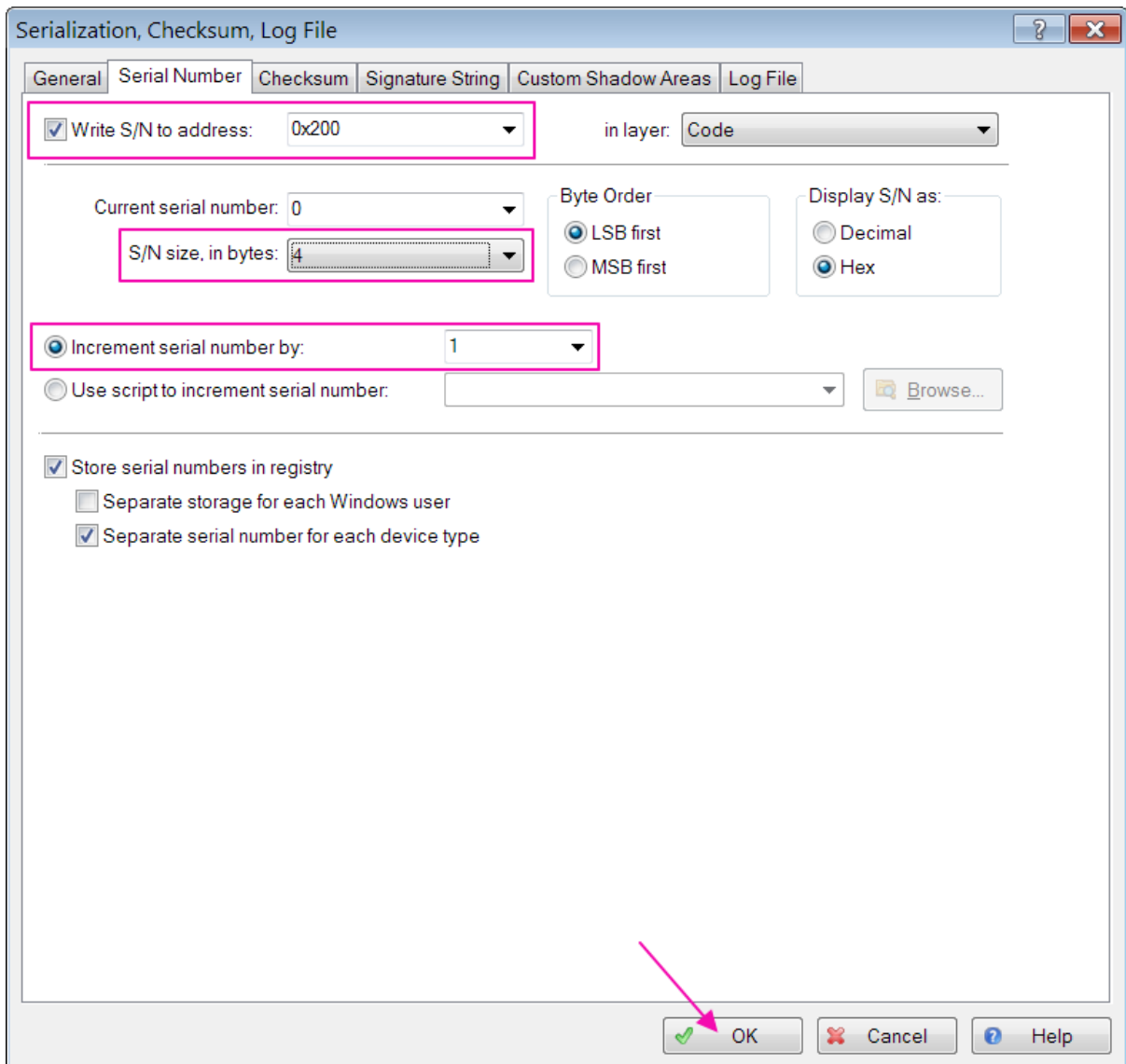

Now we need to set parameters of serial numbers written to each target device. To do this, open serialization settings:
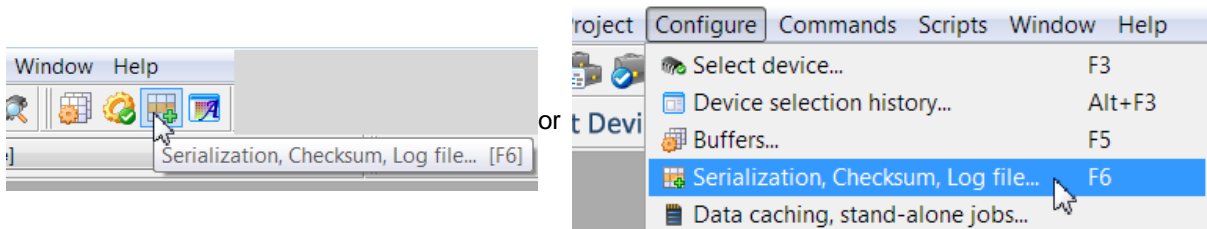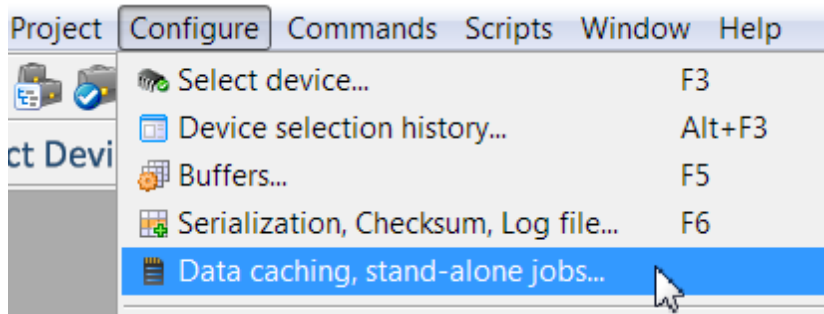

In the dialog that appears, select the **"Serial Number"** tab:

Check off the **"Write S/N address"** check box, and enter 0x200 into the address field. Set serial number size equal to 4 bytes, set increment to 1, then click **OK**:

We are almost done setting project options. We need to turn on data caching; to do this, run menu command **Configure -> Data Caching...:**

This brings up a dialog for serialization parameters. Check off the **Enable Caching** check box, then click **OK:**
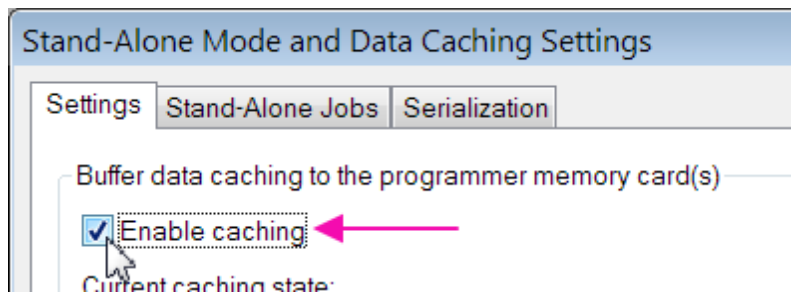
Data caching status now looks like this:
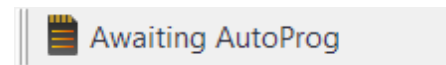
Now the project has to be saved, because projects are not saved automatically:

Connect your <%CPI2-B1%> to a target device and launch the **Auto Programming** command in the **Program Manager** window:
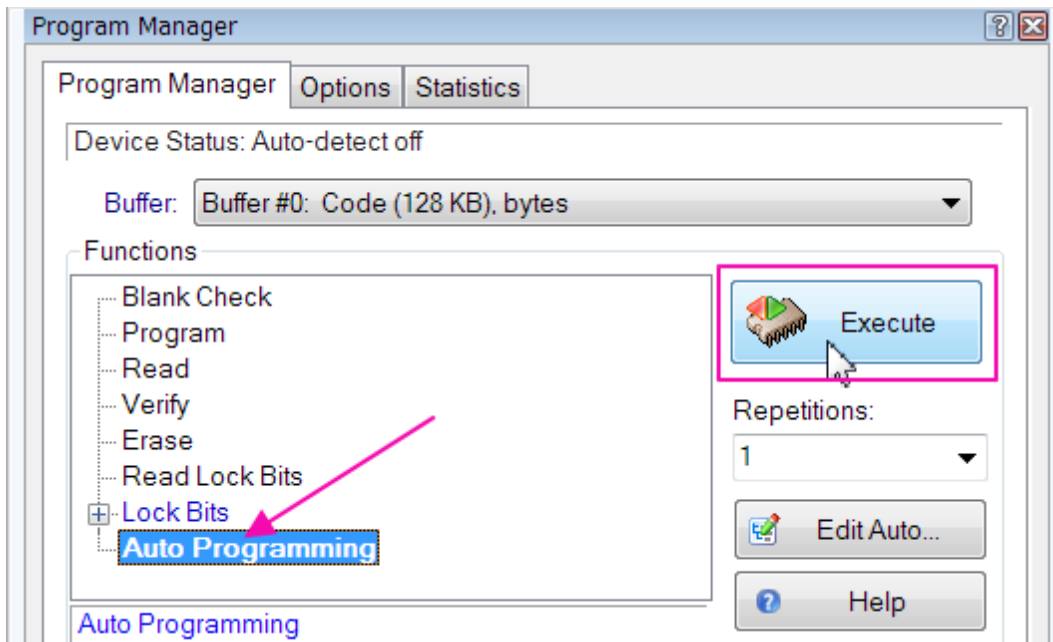
If the Auto Programming operation completed successfully, the history field will display a line saying **"Caching data to the programmer SD card enabled";** caching status will read **"Cached."**
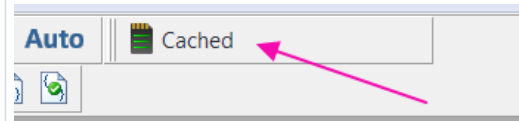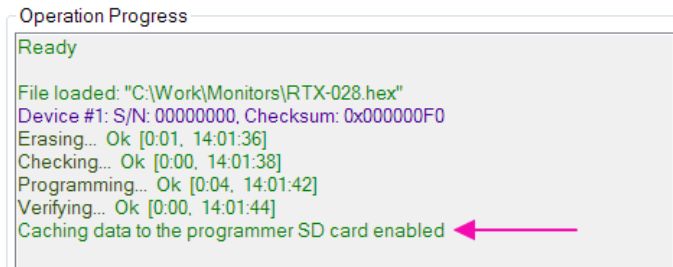
Now we need to generate serial number information for writing them into target devices. To do this, either click on caching status field or select menu command **Configure -> Data Caching...:**
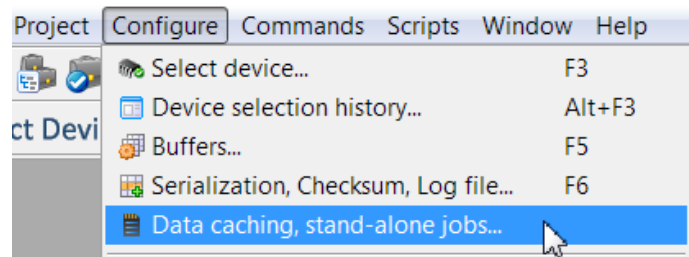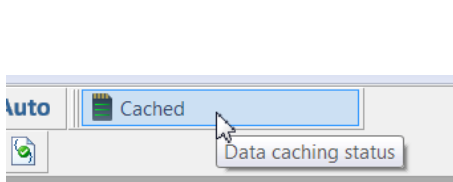
This brings up standalone mode options dialog. Select the **Serialization** tab and specify amount of 10000 devices to generate serial numbers for. Then click **Start Generation** button:
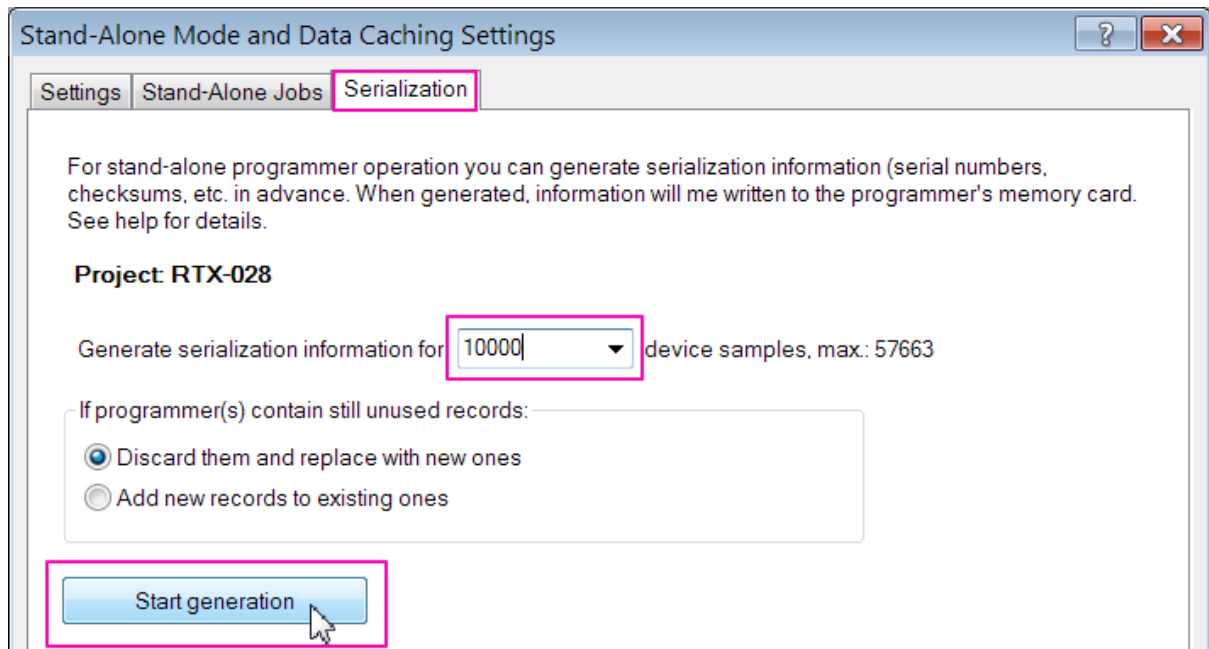
Assign our project to a standalone job number zero by selecting **"Standalone Jobs"** tab and selecting project RTX-08 for job 0:
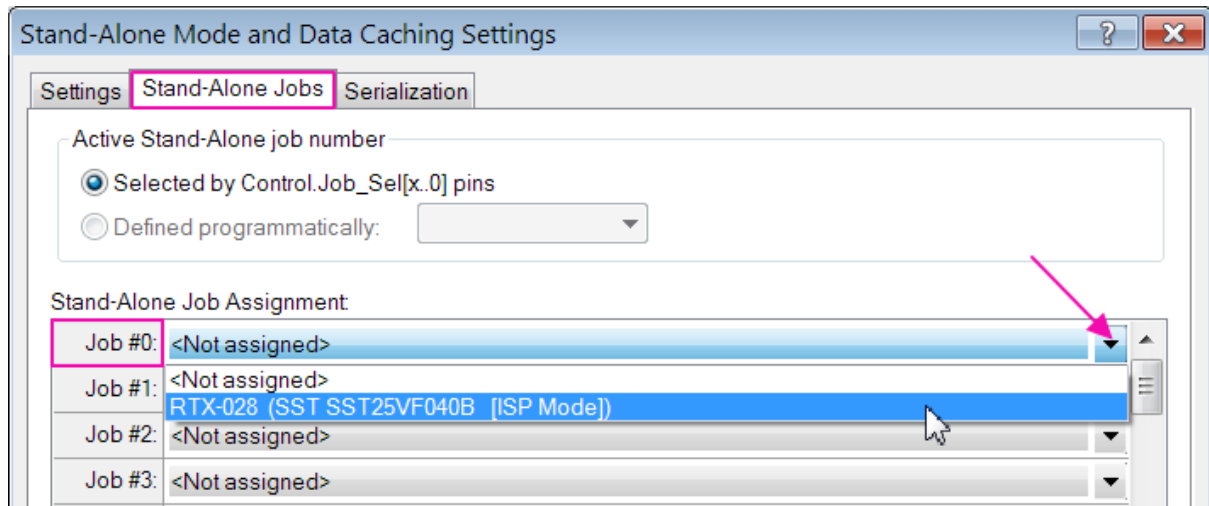
The dialog now looks like this:
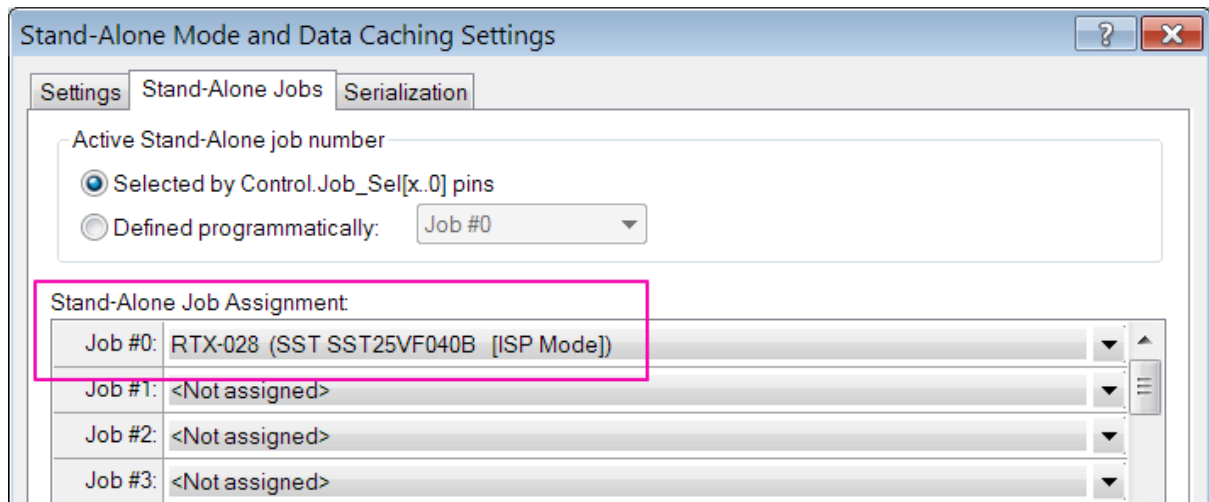
Confirm settings by clicking "   ".

This ends preparation of the project for standalone mode. Contents of memory buffer and all settings have been stored as a project on programmer internal SD card, information for 10000 serial numbers has been generated, the project has been associated with standalone job 0.

One way to switch programmer into standalone mode is by selecting menu command **Commands -> Switch** to **Standalone Mode:**
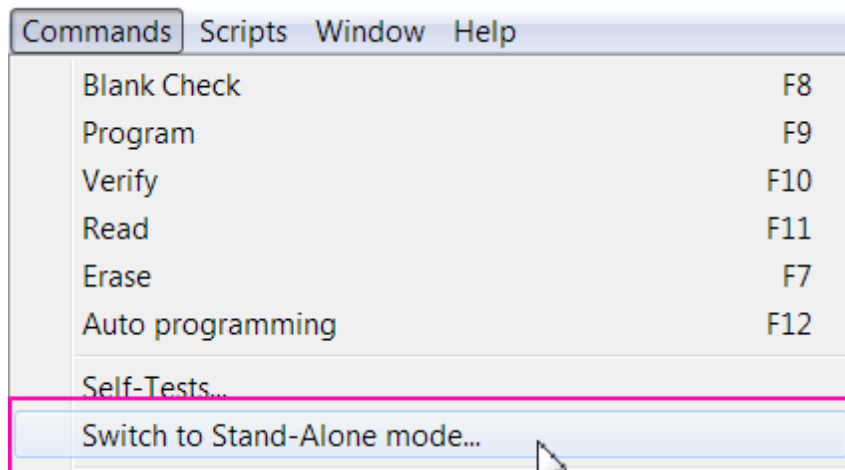
This brings up a dialog which is used to select job 0 to be the active job:
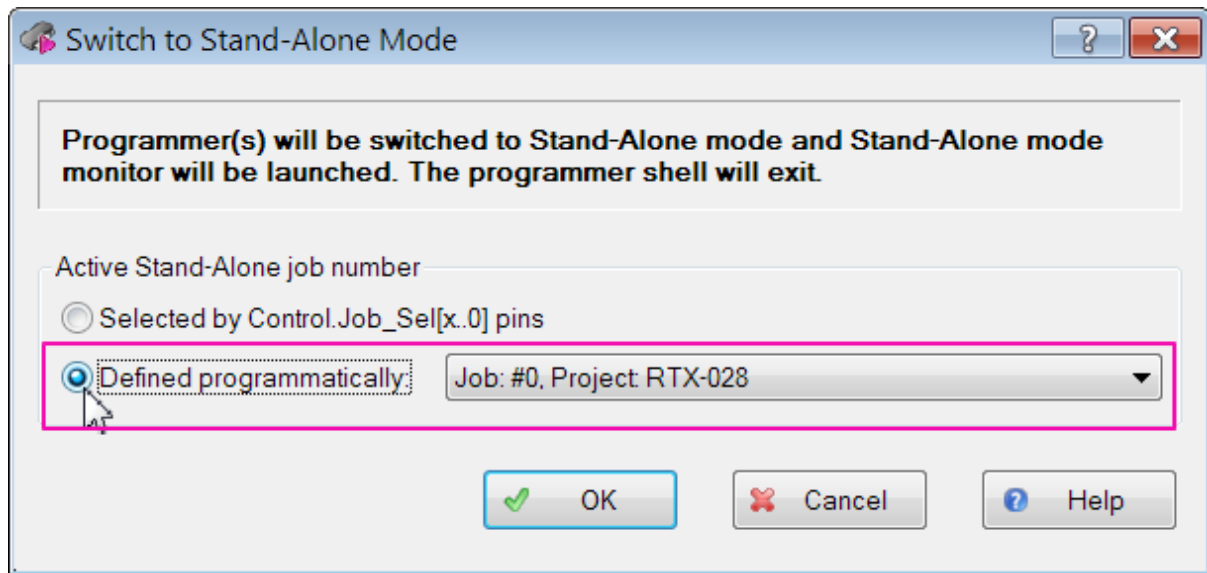
7 Software Development Kit (SDK)

This section describes Phyton ChipProg-02 Software Development Kit (SDK) called ChipProg-02 Application Control Interface.

Developers can use Application Control Interface to control CPI2-B1 programmers by means of their own software.

Application Control Interface provides a comprehensive set of features to control the programming process, including selection of device type, accessing data buffers, loading files, launching programming procedures (also in gang mode), and more.

7.1 ACI Components

Application Control Interface Files

The CPI2-B1 SDK includes the following components:

1. **ACI.DLL** dynamic-link library which implements Application Control Interface functions.

2. ACI.lib export library.

3. Header file **aciprog.h** to be included in user software written in C/C++ programming language. The header contains declarations of all ACI functions, structures and constants. The **windows.h** file must be included in user program before the **aciprog.h**.

4. A set of example files illustrating the use of Application Control Interface.

Platform Requirements

1. Phyton Application Control Interface requires Windows 7, 8 or 10 operating system.

2. ChipProg-02 software must be installed on the computer that controls the CPI2-B1 hardware. The latest ChipProg-02 software version is available for free download from the http://www.phyton.com/

htdocs/support/update.shtml webpage.

Usage with 32- and 64-bit Applications

- 32-bit applications must use the ACI.DLL dll and the ACI.lib export library.

- 64-bit applications must use ACI64.DLL and ACI64.lib.

- Otherwise, there's no difference between 32- and 64-bit applications.

- There's no need to develop 64-bit applications for use with 64-bit operating system: both 32- or 64-bit applications can be used in such case.

Programming Languages

Developers can use any programming language of his choice when working with Application Control Interface; ACI.DLL exports its functions according to the standard rules for Windows operating system.

7.2   Using ACI

To control a CPI2-B1 programmer, user program calls functions in the ACI.DLL. When user program calls the **ACI_Launch()** function, ACI.DLL launches ChipProg-02 executable **UProgNT2.exe** and then controls its operations.

ChipProg-02 GUI can be made hidden or visible. In most cases there is no need to display GUI windows or daialogs; however, this may be used for debugging purposes. User program can also use ChipProg-02 partially, for example to bring up dialogs that show settings, target device selection, file loading and others. Once the programming environment has been set up, the ChipProg-02 GUI can be hidden to free more screen space for the controlling application.

All ACI functions, when called, take either no parameters or one parameter which is a pointer to a structure. Each such structure has its first field set to the structure size; this ensures compatibility of different ACI.DLL versions. The only exception is the **ACI_IDECommand()** function; this sacrifices uniformity in favor of simpler pseudo-function declaration. The aciprog.h header file provides declarations of the parameter-carrying structures.

Names of all the ACI objects (functions and structures) conform to the same naming convention. All names begin with **ACI_** prefix. Names of the parameter structure patterns end with **_Params** suffix.

Numeration of all memory buffers and layers of memory buffers startins with zero. All addresses are 64-bit long and consist of two 32-bit parts (lower and upper), to make them compiler-independent. For example, if the compiler recognizes the **uint64** type, then the structure ACI_Memory_Params can be initialized as follows:

```
ACI_Memory_Params mparams;
*((uint64 *)mparams.AddressLow) = 0x123456789ABC;
```

Note. All addresses in the structures are shown in the format specified by the device manufacturer, i.e. in Bytes, Words, etc. For example, for any 16-bit microcontroller the address format is always a word not a byte.

ChipProg-02 automatically allocates buffer number 0 so that it always exists and does not have to be

explicitly created.

All ACI functions provide return code to the calling application. The return code constants - **ACI_ERR_xxx** - are defined in the aciprog.h file included into the ACI software set.

7.3 Controlling Multiple Programmers via ACI

ACI can be used to launch any number of programmers and control each of them individually. When launching a programmer, ACI creates internal object called *connection* that identifies the programmer. The ACI_SetConnection function is used to select a particular programmer. Once a connection (programmer) is selected, all further calls to ACI functions will use that connection (i.e. they all will affect only the selected device programmer). If there's only one programmer, the connection is selected automatically.

It is important to keep in mind that more than one ChipProg-02 can be launched in either the **Single-programming** or the **Gang-programming** mode. If, for example, a cluster of six CPI2-B1 programmers is launched in the gang mode, a whole cluster driven by the ACI will represent a single connection, and not six connections.

7.4 ACI Functions

This section provides an overview of Application Control Interface functions. Detailed description of each function can be found in the ACI Fuctions reference section.

Calling some functions requires filling in and passing structures that specify memory locations, pointers and other objects associated with the called function, while other functions do not take any parameters.

Table below shows ACI functions grouped by functionality. Most functions are grouped in "bidirectional couples" (In-Out or Get-Set).

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
1. ACI functions that start and stop programming sessions			
ACI_Launch	Starts the ChipProg-02 program. This function must always be the very first in the chain of other Application Control Interface functions that form the programming session.	NA	ACI_Launch_Params
ACI_Exit	Closes the ChipProg-02 program. This function must always be the last one in the chain of other Application Control Interface functions. It completes the external control session.	NA	NA
2. ACI functions that configure the programmer or get its current configuration			
ACI_LoadConfigFile	Loads the programmer configuration parameters from the host computer to the programmer.	NA	ACI_Config_Params
ACI_SaveConfigFile	Saves the programmer's current configuration parameters to the host computer.	NA	ACI_Config_Params
3. ACI functions that get the target device properties or set them			
ACI_GetDevice	Gets the manufacturer's name (brand) and the part number of the device currently being programmed from the programmer to the host computer.	Select Device	ACI_Device_Params
ACI_SetDevice	Sets the manufacturer's name and the part number of the device to be programmed in the programmer.	Select Device	ACI_Device_Params
4. ACI functions that get current parameters of the buffers and layers or configure them			
ACI_GetLayer	Gets the parameters of a specified memory buffer and layer from the programmer to the host computer.	Buffer Dump	ACI_Layer_Params
ACI_CreateBuffer	Creates a memory buffer with specified parameters in the programmer.	Buffer Dump	ACI_Buffer_Params
ACI_ReallocBuffer	Changes a size of the layer #0 in a specified memory buffer in the programmer.	Buffer Dump	ACI_Buffer_Params
5. ACI functions that read the content of the buffer layer or write into it			
ACI_ReadLayer	Reads data from a specified memory buffer in the programmer to the host computer.	Buffer Dump	ACI_Memory_Params
ACI_WriteLayer	Writes data into a specified memory buffer of	Buffer Dump	ACI_Memory_Params

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
	the host computer to the programmer memory buffer.		
ACI_FillLayer	Fills a whole selected layer of a specified memory buffer with a specified data pattern.	Buffer Dump	ACI_Memory_Params
6. ACI functions that get programming parameters from the programmer or set them in the programmer			
ACI_GetProgrammingParams	Gets current programming parameters from the programmer to the host computer.	Program Manager > Options	ACI_Programming_Params
ACI_SetProgrammingParams	Sets programming parameters from the host computer to the programmer.	Program Manager > Options	ACI_Programming_Params
7. ACI functions that get device-specific programming options from the programmer or set them in the programmer			
ACI_GetProgOption	Gets current programming options from the programmer to the host computer.	Device and Algorithm Parameters	ACI_ProgOption_Params
ACI_SetProgOption	Sets programming options from the host computer to the programmer.	Device and Algorithm Parameters	ACI_ProgOption_Params
ACI_AllProgOptionsDefault	Sets default programming options and programming algorithms in the programmer.	Device and Algorithm Parameters	ACI_ProgOption_Params
8. ACI functions that control programming operations			
ACI_ExecFunction	Initiates a specified programming operation, keeping under control its successful completion or failure. It controls a single programmer.	Program Manager	ACI_Function_Params
ACI_StartFunction	Initiates a specified programming operation and then does not check the operation result. It controls a single programmer.	Program Manager	ACI_Function_Params
ACI_GangStart	Used to control multiple device programmers. Initiates auto programming in the gang (gang-programming) mode.	Program Manager	ACI_GangStart_Params
ACI_GetStatus	Gets a current programmer status information.	Program Manager	ACI_PStatus_Params
ACI_TerminateFunction	Terminates a current programming operation.	Program Manager	NA
ACI_GangTerminateFunction	Terminates a current programming operation	Program	ACI_GangTerminate_Para

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
	on a specified site of the gang programmer.	Manager	ms
9. ACI functions that save files from the programmer and load files to the programmer			
ACI_FileSave	Saves a specified file from a specified buffer's layer of the programmer into the instrumental computer.	Buffer Dump	ACI_File_Params
ACI_FileLoad	Loads a specified file from the instrumental computer to a specified buffer's layer in the programmer.	Buffer Dump	ACI_File_Params
10. ACI functions that display programmer's windows and dialogs for setting up and debugging external programming sessions			
ACI_SettingsDialog	Displays the programmer Preferences dialog.	Configure > Preferences	NA
ACI_SelectDeviceDialog	Displays the Select Device dialog.	Select Device	NA
ACI_BuffersDialog	Displays the memory buffers setting dialog.	Buffer Dump	NA
ACI_LoadFileDialog	Displays the file loading dialog.	Buffer Dump	NA
ACI_SaveFileDialog	Displays the file saving dialog.	Buffer Dump	NA

7.5   ACI Structures

This section provides an overview of the structures used in calls to ACI functions. Detailed description of each structure can be found in the ACI Structures reference section.

Structure	The ACI function that uses the structure
ACI_Launch_Params	ACI_Launch
ACI_Config_Params	ACI_LoadConfigFile, ACI_SaveConfigFile
ACI_Device_Params	ACI_GetDevice, ACI_SetDevice,
ACI_Layer_Params	ACI_GetLayer
ACI_Buffer_Params	ACI_CreateBuffer, ACI_ReallocBuffer
ACI_Memory_Params	ACI_ReadLayer, ACI_WriteLayer, ACI_FillLayer
ACI_Programming_Params	ACI_SetProgrammingParams, ACI_GetProgrammingParams
ACI_ProgOption_Params	ACI_GetProgOption, ACI_SetProgOption
ACI_Function_Params	ACI_ExecFunction, ACI_StartFunction
ACI_PStatus_Params	ACI_GetStatus
ACI_File_Params	ACI_FileLoad, ACI_FileSave
ACI_GangStart_Params	ACI_GangStart, ACI_GetStatus
ACI_GangTerminate_Params	ACI_GangTerminateFunction

Here is an example of the structure syntax:

```
typedef struct tagACI_Buffer_Params
{
  UINT    Size;           // (in)  Size of structure, in bytes
  DWORD   Layer0SizeLow;  // (in || out) Low 32 bits of layer 0 size, in bytes
  DWORD   Layer0SizeHigh; // (in || out) High 32 bits of layer 0 size, in bytes
                          //       Layer size is rounded up to a nearest value
supported by programmer.
  LPCSTR  BufferName;     // (in)  Buffer name
  UINT    BufferNumber;   // For ACI_CreateBuffer(): out: Created buffer number
                          // For ACI_ReallocBuffer(): in: Buffer number to realloc
  UINT    NumBuffers;     // (out) Total number of currently allocated buffers
  UINT    NumLayers;      // (out) Total number of layers in a buffer
} ACI_Buffer_Params;
```

Each structure includes a number of parameters (here Size, Layer0SizeLow, NumBuffers, etc.). The parameter's name follows its format (UINT, DWORD, LPCSTR, CHAR, BOOL, etc.). The comment to the parameter begins with a symbol in parentheses showing the direction in which the parameter is passed, as follows:

- **(in)** - the parameter is sent from the instrumental computer **to** the programmer;
- **(out)** - the parameter is sent **from** the programmer to the instrumental computer;
- **(in || out)** - the parameter can be sent in **either direction,** depending on the ACI function context.

7.6 Examples

Phyton ChipProg-02 SDK comes with several usage examples of Application Control Interface functions and structures. Examples reside in the **ACI\Programmer ACI Examples** subdirectory of CPI2-B1 installation directory.

Examples are written in the C language and are projects that can be built using Microsoft Visual Studio® 2008. Project sources can also be compiled using other C/C++ compilers, sometimes with minor adjustments. Building a project creates a Windows console application executable.

To adjust an example project (or a part of it) for use in your application, in the **main()** function adjust paths to the ACI functions. This includes paths to the CPI2-B1 executable file, to file loaded into programmer memory buffer or saved from buffer to disk. You also have to specify real target device type. Sample main() function fragment is shown below.

```
/*+          main                    °          01.07.09 17:37:24*/
......

// Launch the programmer executable
if (!Attach("C:\\Program Files\\ChipProg-02\\6_00_01\\UPrognt2.exe", "", FALSE)) return -1;

// Select device to operate on
if (!SetDevice("Microchip", "PIC16C505 [ISP HV Mode]")) return -1;

// Load .hex file to buffer 0, layer 0
```

```
        if (!LoadHexFile("C:\\Program\\test.hex", 0, 0)) return -1;
```

....

All examples use the ACI.DLL file, therefore that file must be located in the same folder where the example executable file resides, or in a folder listed in the PATH environment variable. For provided examples, ACI.DLL file has already been copied to the folder in which Microsoft Visual Studio creates executable files.

Description of the Examples

Each example has an opening comment briefly describing the program purposes; more comments are added to the code. All examples start with calling the ACI_Launch() function that launches the programmer. You will have to adjust the path the CPI2-B1 executable that is passed as parameter to the Attach() function. After that, target device type is selected; you will need to modify that accordingly.

AutoProgramming.c

This is the simplest and most frequently used example of the CPI2-B1 control by an external program. User program launches the programmer, selects the PIC18F242 target device, loads the test.hex file into programmer buffer, sets default programming options, and then executes a preset Auto Programming batch of functions: Erase, Blank Check, Program, Verify.

SaveMemory.c

This example shows how to save a binary image of a device to a file on disk. First, the user program makes sure a device is insertion into the programmer socket by calling the ACI_GetStatus(&Status) function. After detecting correct and reliable insertion, the program reads data from the specified address range of SST89V564RD device's memory and saves it to the file test.bin on disk.

Checksum.c

This example shows how to calculate a checksum of data read from a device. First, user program verifies device insertion into programmer socket by calling the ACI_GetStatus(&Status) function. After detecting correct and reliable placement, the program calculates the real size of the SST89V564RD device flash memory by executing the ACI_ExecFunction function. It then allocates the buffer 'buf' in the host computer memory for holding data read from the device, reads the data into this buffer, and calculates buffer content checksum.

LongProgramming.c

This example shows how to monitor the AutoProgramming procedure that takes a long time. Programming is launched by calling the ACI_StartFunction. Completion percentage of the operation is then checked by calling the ACI_GetStatus function. If the operation fails, the programmer issues an error message; otherwise operation is continued.

## ProgrammingOptions.c

This example shows how to read, display, and change options set in the Device and Algorithm Parameters Editor window. First, the program checks device insertion in the programmer's socket by calling the ACI_GetStatus() function. After detecting correct and reliable insertion of the device, the program reads current set of options by calling the ACI_GetProgOption() function, and prints them the options. Then the program changes the Vpp from default value to 10.5V and disables device Brown-out

Reset feature.

7.7   API Explorer

API Explorer is a GUI application program that allows experimentation with ACI functions without writing custom code. You can vary ACI function call parameters, study return codes, and see code in C programming language recommended for performing function calls. API Explorer is shipped as part of Phyton ChipProg-02 package. Figure below shows API Explorer window.
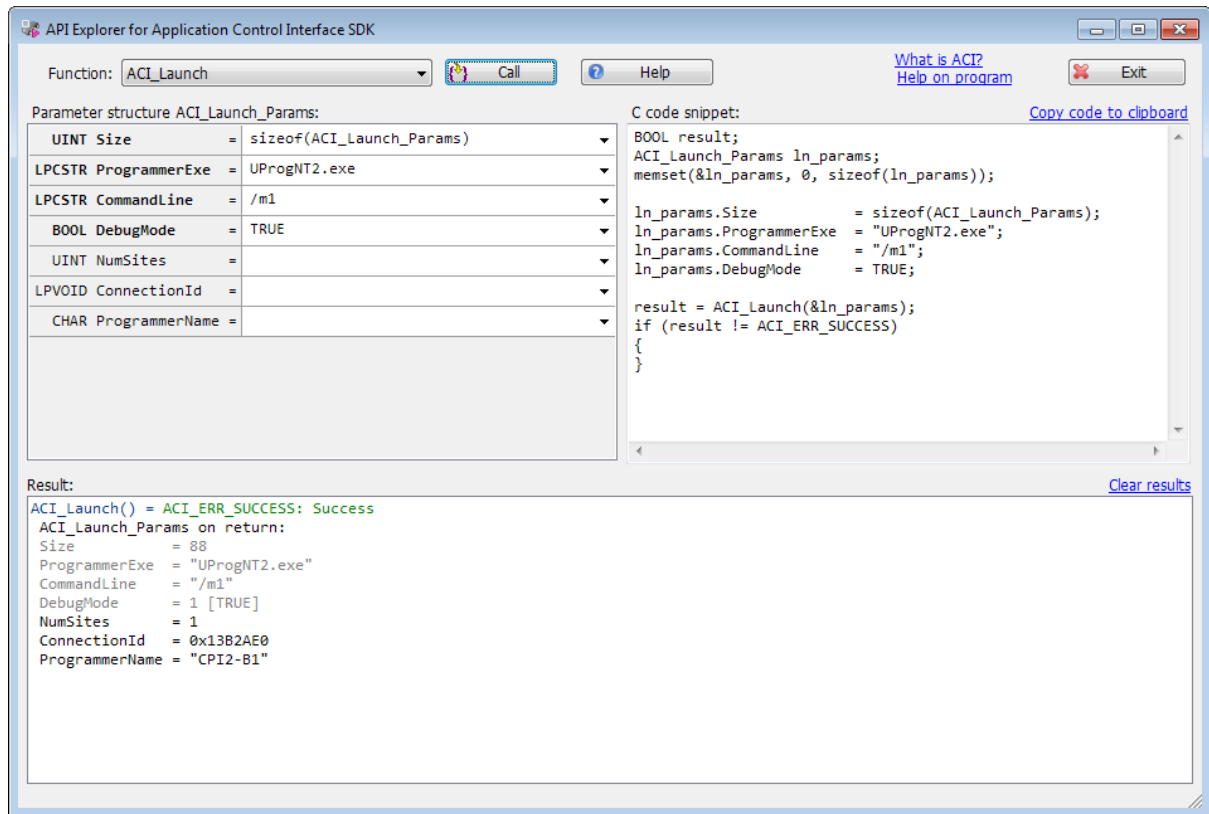
The name of ACI function to call is shown In the upper left corner of the window. In the figure the function is ACI_Launch. The drop down list contains names of other functions. Help button brings up description of the selected function.

Below function name is the title of the structure used to pass parameters to the function. in the figure this is ACI_Launch_Params structure. Structure body follows its name and contains field names and types. Each field can have its value set for the function call. Input parameters are shown in bold type; on the figure these are Size, ProgrammerExe, CommandLine   Debug.

To the right of the list of structure fields is sample code in C programming language that performs the call with parameter passing. This code can be copied to clipboard to be pasted into user program.

To call the function, press the Call button. Results pane will show the return code, a string describing the result, and structure field values. Output parameters that are results of the function call are shown in black, input parameters that the function does not change are gray. To get the string description of the result, the program automatically calls ACI_ErrorString function once the selected function returns control.

How to set values for structure fields.

The first field of each structure is Size which is the size of the structure itself. When a function is selected, API Explorer sets this value to the 'sizeof' of the structure; in the figure it is sizeof (ACI_Launch_Params). This field should be left as is; while experimenting, a number can be entered here.

If a field type is string, the text in the field can be quoted. The program missing quotation marks automatically. The special string NULL is treated literally, as a null pointer.

If a field type is int or Boolean, you can enter 1 or TRUE, and 0 or FALSE which will be placed as is into generated code. In the figure TRUE value is entered in the DebugMode field.

Numeric values may be entered in decimal or hexadecimal format according to C language conventions. An example of hexadecimal number is 0xFFF0.

Fields left blank will be set to zero. This is true also for fields of type string; for example, LPCSTR pointers will be set to NULL, and function call will result in error.

Generated Code Fragment

As shown in the figure, the parameter structure initially is filled in with zeros:

memset(&ln_params, 0, sizeof(ln_params));

Then follows the code to set values of structure field for which values are non-empty. All other fields will contain zeros because the structure has already been zero-filled.

Specifics of ACI_ReadLayer, ACI_WriteLayer functions

When calling ACI_ReadLayer the program allocates its own data buffer. If data size specified in ACI_Memory_Params.DataSize field exceeds 128, the program will impose size limit if 127 cells.

To define data to be written by ACI_WriteLayer call, ACI_Memory_Params.Data must contain hexadecimal numbers without the 0x prefix, for example: C0 03 FF. Value of the ACI_Memory_Params.DataSize field must be equal to the count of specified numbers.

Using API Explorer

All function call are carried out and not simulated. API Explorer allocates and fills in structures and actually calls functions in the ACI.DLL library.

When API Explorer is started, the ACI_Launch function is automatically selected because without calling it first other functions cannot be activated. Filename of the CPI2-B1 executable is specified without full path since it resides in the same directory as API Explorer executable. The CommandLine field contains option /1 which launches programmer in demo mode. If you would like to use one or more real programmers connected to the computer, option /1 must be removed.

When developing custom programs that controls programmers using ACI, please be sure to update the

library ACI.DLL and aciprog.h header file in the directories where you executables reside. The ACI.DLL may be updated in future CPI2-B1 releases.

8 Scripting

8.1 Scripting Overview

ChipProg-02 application can execute commands contained in **script files**. Scripting is a convenient way to automate programming process when using CPI2-B1 programmers.

Scripts can be used to perform various operations, such as automatically load data into memory buffers, calculate checksums, initiate device programming, pause programming in case of an error, manipulate windows, and others.

For the purpose of customizing CPI2-B1 user interface (and for debugging purposes) scripts can create additional windows of two types: the **User window** and the **I/O Stream window**. Scripts can also create custom menus.

Scripts can send messages to **Console window** or to User window created from within the scripts. User windows can display text and graphical data.

ChipProg-02 scripting language is similar to C programming language; most C language features are supported, except structures and pointers. However, there are some differences. The scripting subsystem supports many built-in functions, such as printf(), sin() and strcpy().

Scripts are stored in files with filename extension **.CMD**.

The **scrips** controls and associated dialogs and windows are concentrated under the **Script menu**. The major dialog that controls scripts is the **Script Files dialog.**

How to write a script file

Script is similar to a in C language program. You can use the ChipProg-02 built-in editor or any other text editor to create or edit scripts. You can store script files in your working directory or in the ChipProg-02 installation directory.

Note that you must not use special characters (braces, dash, etc.) in the script file names.

How to run a script file

To start, stop, restart, and debug a script file use the **Script Files** dialog.

The Reference section contains detailed information about scripting.

8.1.1 Simple example

This sample script loads a file, performs automatic programming, and displays the result.

```
#include <system.h>
#include <mprog.h>
```

```
 void main()
 {

   LoadProgram("test.hex", F_HEX, SubLevel(0, 0));              // load file "test.hex" that is an Intel HEX
file                                                                         //
to buffer 0, sub-level 0
   InsertTest = TRUE;                                          // set testing of chip presence to "on"
   if (ExecFunction("Auto Programming") == EF_OK)             // perform an automatic programming
   {
    if (ExecFunction("Verify", SubLevel(0, 0), 10) != EF_OK)  // verify 10 times
    {
     printf("Verify failed: %s", LastErrorMessage);           // display error message if verify failed
     return;                                                  // terminate script
    }
    printf("Verify ok.");                                     // display Ok result
   }
   else
    printf("Programming failed: %s", LastErrorMessage);       // display error message
 }
```

8.2   The Startup Script

When the ChipProg-02 application starts, it automatically runs the **start.CMD** script if it exists. This is similar to execution of the autoexec.bat file in Windows. ChipProg-02 first looks for **start.CMD** file in the current directory; if it is not found, ChipProg-02 then looks for **start.CMD** in its installation directory. If the **START.CMD** is not found, the default CPI2-B1 GUI shell will open.

8.3   Running Scripts

Scripts can be started and restarted in several ways. The easiest one uses the commands of the Script Files dialog. A script can be also be started by calling the **StartCommandFile() function** from another script.

8.3.1   The Script Files Dialog

This dialog is used start, stop, and debug scripts.
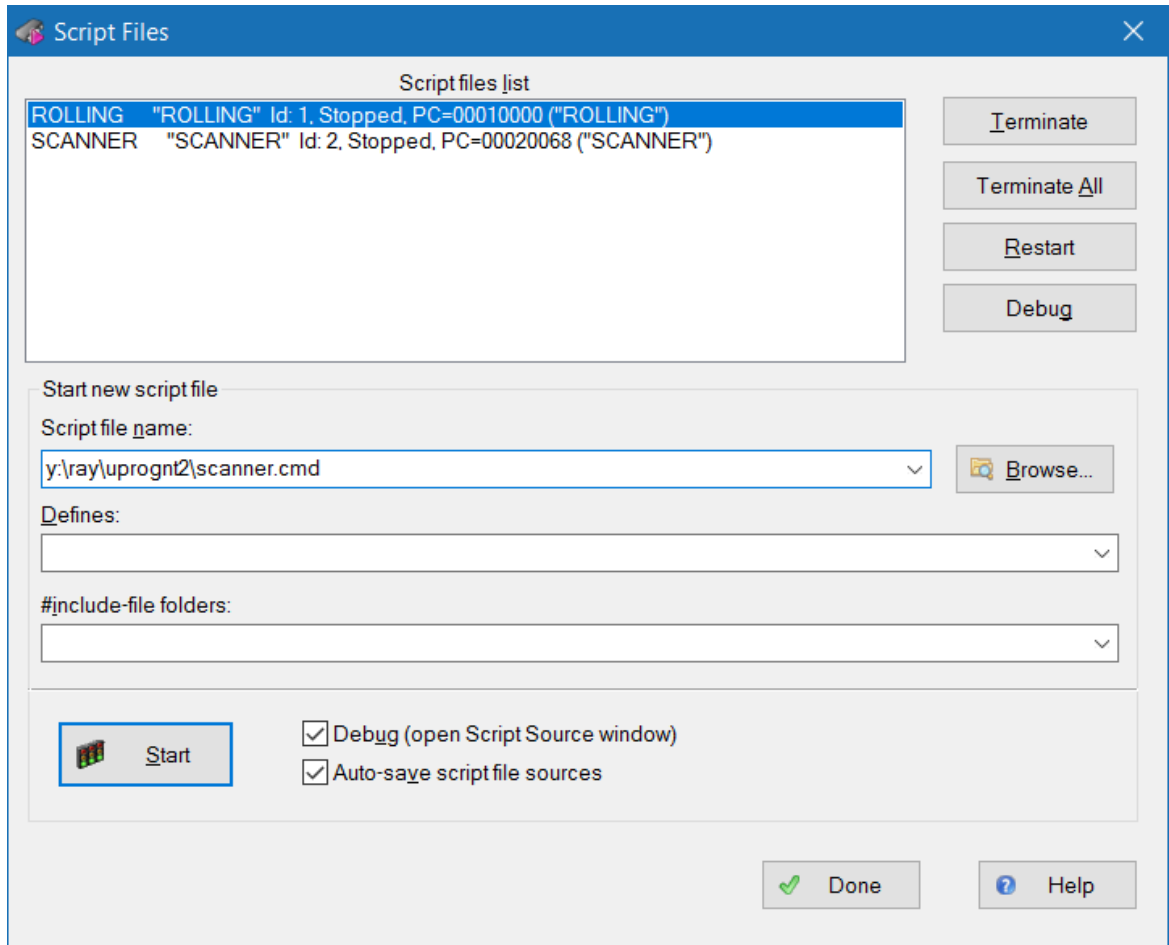
In the top pane of this dialog you see the list of loaded script files along with the state of each script. A script can be in one of the following states:

State of Script	Description
Stopped	Execution of the script file is temporarily stopped.
Running	The script file is being executed.
Waiting	The script is waiting for an event. This state is initiated by calling certain wait functions in the script file text (for example, **Wait**).
Cancelled	The script execution is terminated, but the script file is not yet unloaded from the memory.

To select a script highlight its name in the window. The four buttons on the right of the list affect the highlighted script:

Button	Description
Terminate	Unloads the selected script file if it can be unloaded. Otherwise, it sets up the **Unload Request** flag for the selected script that then goes to the **Canceled** state.
Terminate All	Unloads all script files visible in the window.
Restart	Restarts the highlighted script.
Debug	Switches to the Debug mode for the highlighted script. This command stops execution of the script and opens it in the Script Window for debugging. If the script is in the wait state, execution will be stopped immediately after the script returns from the **Waiting** state.

When you use several script files simultaneously and unload or restart some of them, remember that script files can share global data and functions. If one script accesses data or functions belonging to a script that is already unloaded, the script interpreter will issue error messages and the active script will also be unloaded (terminated).

The buttons and fields in the lower part of the dialog box determine how scripts are run:

Dialog Control	Description
Script File Name	Specifies the filename of the script for loading. You can either type in file name with full path, select it from the drop-down history list, or browse files on disk.
Browse	Opens the **Load/Execute Script File** dialog for locating and loading script files into the **Script File Name** box.
Defines	Defines preprocessor variables. For more information, see **Preprocessor Variables** below.
#include-file Directories	Specifies directories to search for files specified in the **#include <file_name>** directive(s). To specify more than one directory separate them by semicolons. The current directory is searched as well.
Debug (open Script Source window)	If this box is unchecked, a script file automatically start execution upon the file loading. If the box is checked, then upon loading script file a window for debugging is opened. See also How to Debug a Script File.
Auto-save Script File Sources	If this box is checked, clicking the **Start** button automatically saves the source texts of all script files visible in the **Script Source** windows.
Start	Starts the script file specified in the **Script File Name** box.

Preprocessor Variables

The content of the **Defines** text box is equivalent to the #define directive in C language. For example, if you type **DEBUG** in this text box, the result will be as if the **#define DEBUG** directive is placed in the first line of the script source.

You can use Defines to specify values for variables. For example, **DEBUG=3** is equivalent to **#define DEBUG 3**.

You can list several variables in a line, separated by semicolons. For example:

```
DEBUG;Passes=3;Abort=No
```

Also, see Predefined Symbols at the Script File Compilation.

8.3.2 The User Window

User window is a window created by calling built-in **OpenUserWindow** function from within a script. **User** window provides the following functionality**:**

- displaying text;
- displaying graphics (indicators, LEDs, buttons, arrows, etc. by calling built-in graphic functions);
- responding to events (see **WaitWindowEvent**).

These capabilities allow write scripts working in interactive mode.

All functions working with windows (including **User** windows) take window identifier (handle) as a parameter. Because of this you can have several windows of the same type open at the same time.

User window does not have context menu. However, it provides a toolbar with 16 buttons (**0**...**F**), and each button can be programmed to perform a certain function. Pressing a button generates the **WE_TOOLBARBUTTON** event.

8.3.3 The I/O Stream Window

 I/O Stream window is created by calling built-in **OpenUserWindow** function from a script. Script use windows of this type to display text I/O streams. The most common examples of I/O streams are the characters input from PC keyboard and text messages output by the script. Also, you can assign I/O streams to files and input data from those files.

Functions that operate on windows (including the **I/O Stream** window), receive window identifier (handle) as a parameter. Therefore, several windows of the same type can be open simultaneously.

When a function sends some text to this window, the text is appended at the current cursor position. To start the next line the function outputs '\n' (line feed character).

 I/O Stream window features two text display modes, with or without automatic line advance (wrap). In automatic line feed mode, text that does not fit into current line is wrapped to the next line. If auto wrapping mode is off, if the line does not fit in the window it is truncated. The **Wrap** button in the toolbar toggles the this modes. The **Clear** button clears the window contents.

Windows of this type do not have context menu.

8.4   Debugging a Script

A script can be started in **Debug mode**. This is usually necessary while you master the script and need to check if it properly works and make necessary corrections. To start a script in debug mode, highlight its name in the **Script Files dialog** and click the **Debug button.** This brings up the Script Window.

The ChipProg-02 application is designed for source-level debugging. Scripts are debugged in the same way the programs are debugged, executing script step-by-step or up to cursor, setting breakpoints, watching variable values, etc. Debugging process uses Script Source and **Watches** windows. If the **Debug** option is set in the **Script Files** dialog, the Script Source window opens automatically when starting the script.

When the StartCommandFile() function in a script is called to start another script, you can specify parameter instructing it to start the new script in debug mode and open the Process window.

To view the value of a script variable in the **Watches** window, use the **Add Watch** command in the **Script window** menu or the **Add Watch** toolbar button. This can also be done manually in the **Watches** window.

For example, if you need to view the value of the **addr** variable, which is used in a script named TEST, place the **#TEST#addr** construct in the **Watches** window. If **addr** is declared public, that is, outside the function, then it should be written as **##addr**.

8.4.1   The Script Window

The Script window is divided into two panes; the left pane displays the script source, while the right pane is the **AutoWatches pane**.

Syntax constructions and the lines that correspond to the current Program Counter (PC) value (blue strip) and the breakpoints (red strips), are highlighted in the script file text (for more information, see Syntax Highlighting).
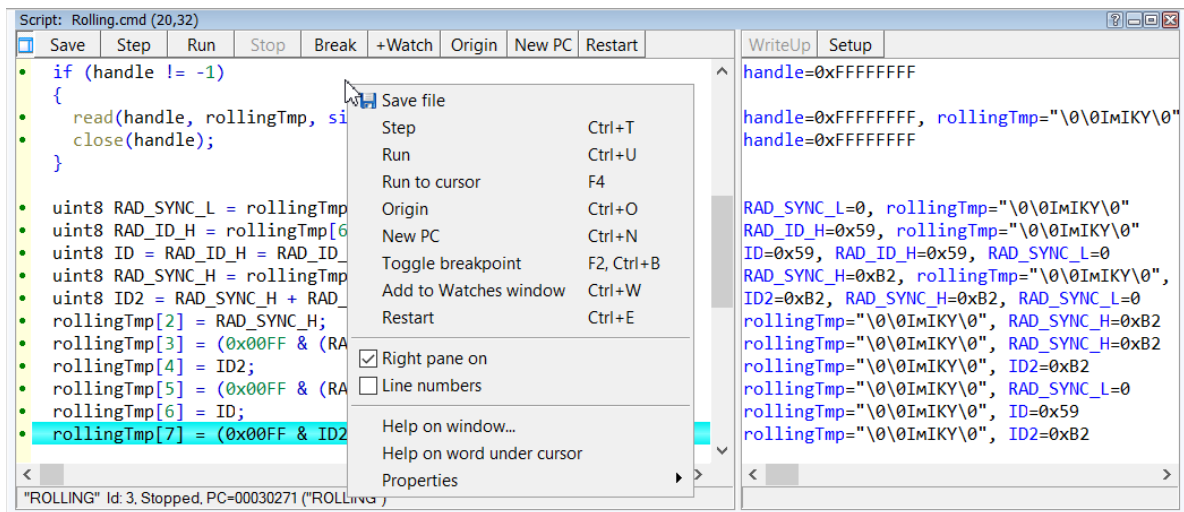
Note. To get help on a function or a variable, click mouse button on the function or variable name in the script source.

8.4.1.1   Menu and Toolbar

The context menu contains the following commands, most of which are duplicated by the toolbar.

Menu Command	Toolbar Button	Description
Step	**Step**	Executes one operator of the script.
Run	**Run**	Starts continuous execution of the script in the window. The script execution can be stopped either by reaching a breakpoint or by the executing **Stop** command.
Run to Cursor		Executes the script up to the line containing cursor. Alternatively, you can double-click the line to carry out this command.
	Stop	Stops the running script.

Origin	Origin	Shows script source from the line whose address corresponds to the script file Program Counter. This operation is not available when source lines do not exist for the program addresses.
New PC	New PC	Sets the script's Program Counter to the address corresponding to the line containing cursor.
Toggle Breakpoint	Break	Sets or clears breakpoint at the address corresponding to the line containing cursor. When you execute the **Run** or **Run to cursor** command, the program execution will be stopped at the breakpoint.
Add to Watches Window	+Watch	Opens the **Watches** window (if not already open) and places the name at the cursor into it.
Restart	Restart	Restarts execution of the highlighted script.

8.4.1.2 The AutoWatches Pane

The ChipProg-02 application displays the visible portion of the script in the **Script** window. The names of variables, called **AutoWatches,** which belong to the visible script lines, are listed along with their values in the right pane of the window. When you scroll through the **Script** window, contents of the **AutoWatches** pane refreshes automatically.

The **AutoWatches** can be displayed in binary, hexadecimal, decimal or ASCII format. To select a format, click on the **Setup** toolbar button or right click anywhere in the pane to open context menu.

8.4.2 The Watches Window

AutoWatches pane of the Script window displays values of currently visible script variables. In addition, you may want to monitor other explicitly specified script variables and expressions. To do so, ChipProg-02 provides the **Watches** window. For each variable, the window displays its name, value, type and address, if any.

A newly opened **Watches** window has one **Main** tab. You can add custom tabs (using **Display Options** command in context menu) or rename any existing tabs. The tabs operate independently of each other, each tab being functionally equivalent to a separate **Watches** window. However, if desired, you can open several **Watches** windows.

Each **Watches** window has the **+Watch** toolbar button. Clicking on this button opens a dialog for adding a selected object to the **Watches** window.

Grids in the Watches Window

For better readability, the Watches window can be divided into cells by vertical and horizontal grid lines. Enable the grid by checking corresponding boxes in the **Configure** menu > **Environment** > **Fonts** tab.

The window context menu contains the following commands, most of which are duplicated by toolbar buttons.

Command	Description
Add Watch	Adds one or more objects to the window. Opens the **Add Watch** dialog to choose an object by name. Also, you can enter an expression as a name.
Delete Watch	Deletes a selected object from the **Watches** window.
Delete All Watches	Deletes all watches from the window.
Modify	Opens the **Modify** dialog to set a new value for a selected variable. Alternatively, just enter the new value.
Move Watch Up	Moves selected watch up the list.
Move Watch Down	Moves selected watch down the list.
Display Options	Opens the **Display Options** dialog to change the display settings for selected object and also to add/delete tabs to/from the window.

8.4.2.1   The Display Watches Options Dialog

Use this dialog to set the display options for the selected variable or expression in the **Watches** window.

Dialog Control	Description
Watch Expression	Contains selected expression. The drop–down list contains the previously used expressions.
Display Format	Specifies the format for displaying selected expression (binary, hexadecimal, decimal, or ASCII).
Pop-up Description	Contains check boxes that choose format for displaying pop-up SFR descriptions.
Display Bit Layout	If this box is checked the SFR bits will be displayed in the pop-up layout descriptions.
Display Bit Descriptions	Checking this box enables displaying the pop-up descriptions for the SFR bits, if any.
Auto-size Name Field	When this box is checked and when vertical grid is visible (see note below), the window automatically adjusts the **Name** column width to fit the longest record in the column.
Tabs	Lists all tabs present in the window.
Add Tab	Opens the **Add New Tab to Watches Window** dialog for entering a new tab name. The window adds the new tab upon pressing **OK**.

Remove Tab	Removes the tab selected in the **Tabs** list.
Edit Tab Name	Opens the **Edit Watch Window Tab Name** dialog for editing tab name.
Global Debug/ Display Options	Opens **Debug Options** dialog.

Note. To make grids visible in the **Watches** window, open **Configure** menu, the **Environment** dialog, the **Fonts** tab and check corresponding boxes in the **Grid** field.

8.4.2.2   The Add Watch Dialog

Use this dialog to add symbol names (for example, a variable name or an expression) to the **Watches** window. The dialog contains a list of symbol names defined in, or known to, the program.

Dialog Control	Description
Name or expression to watch:	Enter the symbol name or expression to be added. You can specify several names and expressions either manually (separated with semicolons) or by selecting from the list with the **Ctrl** key pressed.
History	List of previous names and expressions.

8.5   Script Editor

A **script** is similar to a source program written in C programming language. Scripts can be created and edited using ChipProg-02 built-in editor described below or using any other text editor. Scripts can be stored as files in your working directory or in the directory where the ChipProg-02 is installed.

To open a built-in editor select **Script menu > Editor window.** The Editor toolbar that contains all buttons related to editing is normally hidden. To customized editor toolbar right click on a blank area in the main toolbar, select Customize in the drop-down menu, and check the boxes for editor functions that you want to make visible.

To create a new script file and open it for editing, select **Script menu > Editor window > New**. This will open a blank window shown below. Right clicking in the window brings up the **Editor menu** with buttons you can add to the local **Editor toolbar**. On the figure the toolbar is shown above the window.
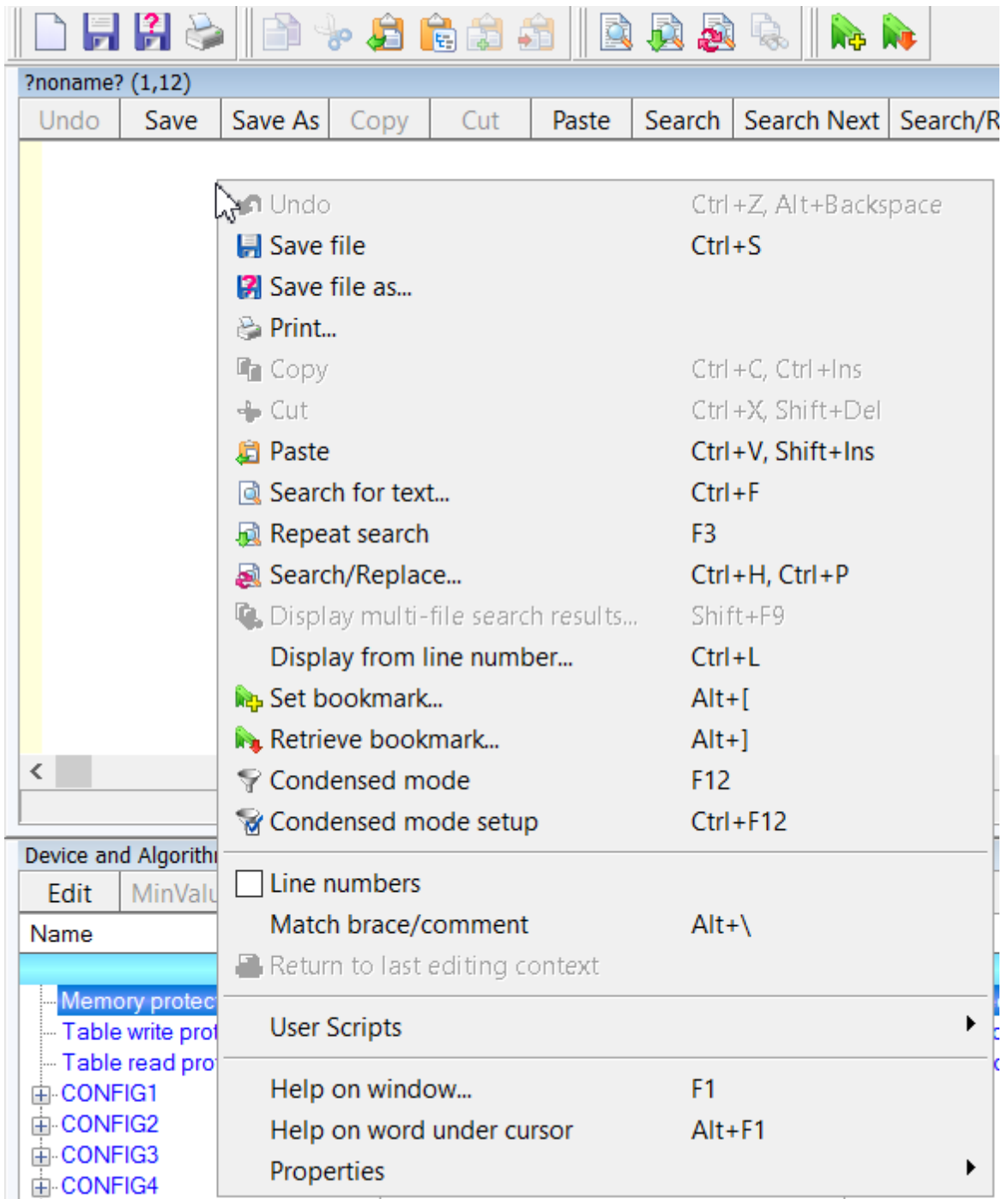
Now you can edit the script in the window.

Note that you should not use the punctuation characters (braces, dash, etc.) in the script file name.

To finish editing click on the **Save** button in the **Editor toolbar,** the program will prompt you for script filename and location.

8.5.1 The File Menu

Commands in this menu act on the currently active **Edit** window.

Button	Command	Description
	New	Opens the **Editor window** for a new script file.
	Open...	Brings up **Open file** dialog to load a script file for editing. The file name and path can be either entered or browsed here.
	Save	Saves contents of the active window to a file on disk.
	Save As...	Opens the **Save as...** dialog.
	Print	Opens standard **Print** dialog for default printer. You can print entire file or just the selection.
	Properties..	Common properties for open files.

8.5.2 The Edit Menu

Commands of this menu act on the active **Edit** window.

Button	Command	Description
	Undo	Undoes the last text editing action performed in this window. For example, if the last action deleted a line, then deleted line will be restored. The number of steps provided by the Undo function is set in the of the **Configure > Editor Options > General** tab.
	Copy	Copies selection to clipboard. The text format in the clipboard is standard and the copied block is accessible to other programs.
	Cut	Moves selection to clipboard..
	Paste	Pastes text from clipboard, starting at the cursor position.
	Clipboard History/ Repository	Opens the **Clipboard History/Repository** dialog.
	Append to Clipboard	Copies and appends selection to clipboard contents.
	Cut & Append to Clipboard	Cuts selection and appends it to clipboard.
	Fast Copy	Copies selection to a specified position in the same window.
	Fast Move	Moves a block from one position in a window to another position in the same window.
	Block Off	Unmarks a marked text block.

	Search	Opens the **Search for Text** dialog.
	Next Search	Repeats search with parameters used in the previous search.
	Replace	Opens the **Replace Text** dialog.
	Display Multi-file Search Results	Re-opens the last multi–file search results in the **Multi-File Search Results** dialog.
	Display from line number...	Opens the **Display from Line Number** dialog for you to specify a line number. Source text will be displayed from this line.
	Set bookmark...	Opens the **Set Bookmark** dialog to set a local bookmark.
	Retrieve bookmark	Opens the **Retrieve Bookmark** dialog to retrieve a local bookmark.
	Condensed mode	Toggles Condensed display mode on and off.
	Condensed mode setup	Opens the **Condensed Mode Setup** dialog.
	Line numbers on/off	Toggles line numbers on and off.
	Return to last editing context	Activates the most recently edited **Source** window, and places the cursor in its final position during the edit.

8.5.3   Block Operations

Block operations are operations on blocks of text. The script **Source** window supports persistent blocks and performs a full range of operations with standard (stream), vertical (column) and line blocks of text.

Non-persistent blocks In this mode, once a block is marked, you have to immediately carry out an operation with it (delete, copy, etc.). Any movement of cursor turns the marking off. If a block is marked, then any entered text will replace the block with the typed text.

Persistent blocks In this mode, the block remains marked until the marking is explicitly removed (hot key **Shift+F3**) or the block is deleted (**Ctrl+X**). The Paste operation for persistent blocks has certain specifics. Two additional block operations are available for persistent blocks: fast copy and fast move. These operations do not use clipboard and require fewer keyboard manipulations.

To enable persistent block mode check corresponding box in the **Main menu > Configure>Editor Options> General tab.**

Standard blocks A standard (stream) block contains a "text stream" that begins at the initial line/column of the block and ends at the final line/column.

The **Standard blocks** mode is enabled by default.

Line blocks A line block consists of lines of text. To mark a line block, put the cursor anywhere in the first line and press **Alt+Z**; then put the cursor anywhere in the last line of the block and press **Alt+Z** once more (the latter is not necessary if the block is to be immediately deleted or copied to the clipboard).

Line blocks are always available.

Vertical blocks A vertical block contains a rectangular text fragment. Characters within the block that go

beyond line ends are considered to be spaces.

 Vertical blocks are convenient in cases like the following:

```
char Timer0 far ;
char Timer1 far ;
char Int0  far ;
char Int1  far ;
```

Assume the word "far" is to be moved to the place right after the word "char" in each line. The stream blocks are of little help here. However the task can be easily done with one vertical block. Mark the persistent vertical block containing the word "far" in each line, place the cursor on the first letter of word "Timer0" and press **Shift+F2** (fast move the block):

The **Vertical Blocks** checkbox in the the **Main menu > Configure>Editor Options> General tab** toggles between the vertical block and the stream block modes. Standard blocks are enabled by default; i.e. the **Vertical Blocks** checkbox in the **Editor Options** dialog is unchecked by default. Line blocks are always accessible, independent of the state of the **Vertical Blocks** checkbox.

To mark a block, move the mouse while pressing its left button or use the arrow keys on the keyboard while holding the **Shift** key. To unmark the block press **Shift+F3**.

Copying / moving blocks

A marked block can be copied or moved in two ways within the same **Source** window: directly (fast copying, fast moving) or using clipboard (Copy/Cut/Paste). Copying and moving blocks across **Source** windows or to another application is always done using clipboard.

Note. The result of copying a stream or vertical non-persistent block depends on the INSERT mode. If the mode is enabled, the block is inserted into the text starting at the cursor position; otherwise the copied block overwrites the text in an area of equivalent size.

Fast copying / moving

Fast copying or moving of the blocks in the same window happens without the use of clipboard. It is convenient because it requires pressing the keys only once per operation. Mark a persistent block, then place the cursor to the destination position and press **Shift+F1** to copy, or **Shift+F2** to move the block.

8.5.4   Condensed Mode

In the Condensed mode, only lines that satisfy a specific criteria are displayed in the window. There are two available criteria:
- line must contain the given substring;
- the first non-space character in a line must be at a specified position (column).

Examples:
(a) with the substring criterion and the substring set to "counter," only the lines containing the word "counter" are displayed;

(b) with the second criterion and the position set to four, only the lines in which text starts at column 4 will be displayed.

Condensed mode brings lines having some common feature to "one place." If you attentively follow the rule to begin a declaration of data at position 2, procedures at position 3, and interrupt handlers at position 4, Condensed mode will help you find necessary declaration. If you comment certain lines with the same or similar comments and use the Condensed mode with substring, you will be able to benefit from your composing style. In Condensed mode, you can move the cursor just the same way as in normal mode.

The criterion for display is set in the **Main menu > Script > Text Edit > Condensed Mode Setup** dialog. To toggle Condensed mode on/off, use the **Edit** menu command, the **Condensed Mode** command of the local menu or the **F12** hot key. To exit Condensed mode, press **Esc**; at exit the cursor returns to the position at which it was before the mode was turned on. To exit condensed mode leaving cursor in the same line as while in the mode, press **Enter** or begin editing the line.

8.5.5   Syntax Highlighting

When the **Source** window displays script source, it marks certain language constructs with different colors. This feature improves readability. The following constructions are highlighted:
- Punctuation and special characters: **( ) [ ] { } . , : ;** etc.
- Comments starting with **//** are highlighted.
- Comments enclosed in the **/* */** pairs are highlighted only if the opening and closing pairs are placed in the same line.
- Strings enclosed in double or single quotation marks.
- Keywords of the scripting language (**for**, **while**, and so on).
- Type names of the language (**char**, **float**, and so on).
- Library function names (**printf**, **strcpy**, and so on).

You can disable syntax highlighting through the **Main menu > Configure>Editor Options> General tab>Syntax Highlighting** flag. In addition, you can change the color of each construction; to do so use **Main menu > Configure> Environment > Colors tab.**

8.5.6   Automatic Word Completion

It is normal for words (labels, names of variables) to be repeated within a some part of a file; the **Source** window helps you typing such word.

When the cursor is at the end of line being composed, upon typing a letter the editor scans the text above and below the current line. If a word beginning with the letters you just typed is found, the editor will "complete" this word for you by writing the remaining part of the word from the current cursor position. To accept the completion press **Alt+Right** (**Alt+<right arrow>**) and the editor will append the remaining part of the word to the text as if you have typed it yourself. To discard completion, just continue typing and the editor will accept whatever you type. At any point during typing you may press **Alt+Right** to accept editor's completion suggestion.

You can press **Alt+Right** at any time (not only when the editor offers you to complete a word). In this case the editor will open a list of words that begin with the typed letters. If the list does not contain an applicable word, just ignore the prompt. The right pane of the Source window, if it is open, also displays the word completion list.

To disable automatic word completion, uncheck the **Automatic Word Completion** box in the **Main menu > Configure>Editor Options> General tab.** When the box is checked, a number placed in the **Scan Range** box defines the number of lines for the editor to scan. The default is 24 lines below and 24 lines above the current line. When this parameter is greater than the total number of lines in the file (for example, 65535), then program composing will become slower because the whole file will be scanned.

8.5.7   The Quick Watch Function

The **Quick Watch** function works as follows: if you roll the mouse pointer over a variable name in the **Source window** or the **Script Source** window, a small box containing the value of the variable will be opened. This box disappears upon moving the mouse off the object.

8.5.8   Dialogs

This section describes dialogs used by Script Editor.

8.5.8.1   The Search for Text Dialog

This dialog sets criteria to search for text in files. This dialog and the **Replace Text** dialog have a number of features in common. To specify file names, you can use one or several wildcards. Also, the names may contain paths. You can search more than one file by using parameters of the **Multi-File Search** area.

Dialog Control	Description
String to Search for	Text to search for.
Case Sensitive	Unchecked by default. Checking this box makes the search case sensitive.
Whole Words Only	Unchecked by default. If checked, the editor will search only for whole words: the string will be found only if it is enclosed between punctuation characters or delimiters (spaces, tabs, commas, quotation marks, etc.).
Regular Expressions	Unchecked by default. Checking off this box specifies that the search string is a regular expression.
Global	Search entire file for the string. Enabled by default.
Selected Text	Search for string in the selected block.
From Cursor	Search from the current cursor position.
Entire Scope	Search from the beginning or end of the file (depending on the search direction). Enabled by default.
Perform Multi-File Search	If checked, the editor will search in all project files (see the notes below). If unchecked, the search will be performed in current **Source** window only.
Search All Source Files in Project	If checked, the editor will search in all the source files included in the project.
Include Dependency Files	If checked, the editor will search in all the source files included in the project and all files on which the source files depend, whether explicitly or implicitly.
Search Wildcard(s)	Check this box to search for one or several wildcards specifying the files to be searched. Separate wildcards with semicolons. No quotes are required to denote Windows-style long names. Example: `*.txt;*.c;c:\prog\*.h`. This option and the **Search All Source Files in Project** option act independently of each other: you can search in all files of the project AND in other files that comply with the specified wildcard(s).
Search Subdirectories	If checked, the editor will search in subdirectories of all directories specified by the **Search All Source Files in Project** option and by wildcards.
Starting Path	Begin search from the directory specified in this text box. This directory serves as the common path and is useful when there are several wildcards such as the following ones:

```
c:\prog\text\source\*.txt;c:\prog\text\source\*.doc
```

In this case, make use of wildcards (`*.txt;*.doc`) and common path (`c:\prog\text\source`).

Notes

1. When you search in the file opened in the **Source** window, then only the window buffer will be searched, not the file on disk.

2. Multi-file search is performed in all source files of the project. Upon finishing, the **Multi-File Search Results** dialog remains open.

8.5.8.2   The Replace Text Dialog

This dialog sets parameters for search-and-replace operation. This dialog and the **Search for Text** dialog have a number of common parameters, which function in the same way in both dialogs. To specify file names, you can use one or several wildcards. Also, the names may contain paths. You can search in more than one file at once by using parameters of the **Multi-File Search** area.

Element of dialog	Description
Text to Search for	Specifies the text string to look for (search string).
Replace with	Specifies the text string to replace the found one.
Case Sensitive	Unchecked by default. Checking this box specifies that the case of the string is to be matched.
Whole Words Only	Unchecked by default. If checked the editor will search only for whole words: the string will be found only if it is enclosed between punctuation or separation characters (spaces, tabulation symbols, commas, quotation marks, etc.).
Regular Expressions	Unchecked by default. Checking of this box specifies that the search string is a regular expression.
Prompt at Replace	Checked by default. If checked, the editor will always pop up the **Confirm Replace** dialog requiring your permission to replace the found text. If unchecked the editor will automatically replace the searched-and found text.
Global	Search entire file for the string. Enabled by default.
Selected Text	Search in selected block.
From Cursor	Search from current cursor position.
Entire Scope	Search from beginning or end of the file (depending on the search direction). Enabled by default.
Perform Multi-File Search and Replace	Checked by default. If checked, the editor will search in all project files (see the notes below). If unchecked, the search will be performed in the current **Source** window only.
Search All Source Files in Project	If checked, search in all the source files included in the project.
Include Dependency Files	If checked, search in all the source files included in the project and all files on which the source files depend, whether explicitly or implicitly.
Search Wildcard(s)	Check this box to search for one or several wildcards specifying the files to be searched. Separate wildcards with semicolons. No quotes are required to

denote Windows-style long names. Example: `*.txt;*.c;c:\prog\*.h`. This option and the **Search All Source Files in Project** option act independently of each other: you can search in all files of the project AND in other files that comply with the specified wildcard(s).

Search Subdirectories	If checked, search in subdirectories of all the directories, which are specified by the **Search All Source Files in Project** option and by wildcards.
Starting Path	Begin search from the directory specified in this text box. This directory serves as the common path and is useful when there are several wildcards such as the following ones:

`c:\prog\text\source\*.txt;c:\prog\text\source\*.doc`

In this case, make use of wildcards (`*.txt;*.doc`) and common path (`c:\prog\text\source`).

Notes

1. When you search in the file opened in the **Source** window, then only the window buffer will be searched, not the file on disk.

2. Multi-file search is performed in all source files of the project. Upon finishing, the **Multi-File Search Results** dialog remains open.

8.5.8.3 The Confirm Replace Dialog

This dialog asks permission to replace the found string. You can turn the prompt on/off by checking/clearing the **Prompt at Replace** box in the **Replace Text** dialog.

Button	Function
Yes	Replace the found string.
No	Cancel this replacement. If the procedure is started with the **Change All** button for all occurrences in the search area, then the search-and-replace process will continue.
Non-Stop	From this moment, replace all found strings in this file without prompt.
Cancel	Cancel the search-and-replace process.
Skip this File	Stop searching in this file and switch to the next one.
Replace in All Files	Replace all occurrences in all other files without asking for confirmation.
Move cursor to the Yes/No Buttons	If checked, the cursor will be automatically placed on the **Yes** button on each inquiry for confirmation.

8.5.8.4 The Multi-File Search Results Dialog

This dialog displays the multi-file search results. To learn about the multi-file search, see the **Search for Text** dialog.

The **List of Matched Files** shows files in which the search string is found. File name is on the left and its directory is on the right. The line with green text beneath this box displays information about the file selected in the box. "File in memory" means that the file is opened in the **Source** window. General information from FAT means the file is on disk, not loaded. The **Preview** area shows the source line with the found text string.

The **Sort Files by** area includes a radio button with four file sorting options. When the **Consider Directory** box is checked, the files are sorted with respect to their directories.

The **Edit** button opens selected file in a new **Source** window and places the cursor on the line with the found string. The found string background is highlighted. To check for other occurrences of the search string in the file, press **Ctrl+R** or use the **Next Search** command of the **Edit** menu.

The **Close** button closes the dialog but search results are not lost. To reopen the dialog use the **Display Multi-file Search Results** button. You can also use the same command of the **Edit** menu or press **Shift +F5**. The files in the **List of Matched Files** box, which are opened in the **Source** window, will be marked with asterisks on the left.

8.5.8.5   Search for Regular Expressions

Text editor supports "regular expressions." Regular expressions contain control characters in the search string:

?	Means any one character in this place. Example: if you specify *?ell* as the search string, then "bell," "tell," "cell," etc. will be found.
%	Means beginning of line. The characters following '%' must begin from column 1. Example: %Counter - find the word "Counter," which begins at the first column.
$	End of line. The characters preceding the '$' should be at the trailing positions of the line. Example: Counter$ - find the word "Counter" at the line end.
@	Match the next character literally; '@' lets you specify the control characters as usual letters. Example: @? - search for the question mark character.
\xNN	The hexadecimal value of the character. Example: \xA7 - find the character with the hexadecimal code of A7.
+	Indefinite number of repetitions of the previous character. For example, if you specify 1T +2, then the editor will find the lines containing "1" followed by "2", which are separated with any number of the letter T.
[c1-c2]	Match any character in the interval from c1 to c2. Example: [A-Z] means any letter from A to Z.
[~c1-c2]	Match any character whose value is outside the interval from c1 to c2. Example: [~A-Z] means any character except for the uppercase letters.
text1|text2	The "|" character is the logical "OR" and the editor will look for either **text1** or **text2**. Example: LPT|COM|CON means search for "LPT" or "COM" or "CON."

8.5.8.6   The Set/Retrieve Bookmark Dialogs

Bookmarks help you return to a marked cursor position in a source file.

You can set and retrieve up to 10 local bookmarks. Every local bookmark has an individual numbered button assigned to it.

To open the **Set Bookmark** dialog, press **Alt+[**. To open the **Retrieve Bookmark** dialog, press **Alt+]**. To set/retrieve a bookmark, press its numbered button. The number of the bookmarked line, the bookmark position in the line (in brackets) and the text of the line are shown at the right of the button.

Local bookmarks are stored in the configuration file and you can work with them in the next session.

8.5.8.7 The Condensed Mode Setup Dialog

This dialog sets up the parameters for the Condensed mode of the **Source** window.

Display Lines of Text area has radio buttons for switching between two alternative criteria for condensing text in the **Source** window: **Containing String** and **Where First Non-blank Column Is:**

1. If you check the **Containing String** radio button, Source window displays only lines with text that matches the sub-string specified in the text box at the right. Additionally, you can specify case-sensitivity, that whole words only should be used, and that the sub-string is a regular expression.

2. If you check the **Where First Non-blank Column Is** radio button, the Source window will display the lines where text begins from the position specified in the **Column** box. Then you should select one of four options by checking an appropriate radio button:

- **Equal to** - the first non-space character should be exactly in the specified column. For example, if you specify position number 2, the window will display only the lines whose text begins in column 2.
- **Not Equal to** - the first non-space character should be in any column except the position specified here. For example, if you specify position number 2, the window will not display all the lines beginning in this column. All other lines will be displayed.
- **Less than** - display only the lines in which text begins at a position less than the specified one.
- **Greater than** - display only the lines in which text begins at a position greater than the specified one.

Once setup is complete click **OK** to switch the **Source** window into Condensed mode.

8.5.8.8 The Display from Line Number Dialog

Use this dialog to display source file in the active **Source** window starting with specified line. Enter the line number or select any previous number from the **History** list. Line numbers start with 1.

9   Reference

9.1   Error Messages

9.1.1   Error Load/ Save File

5005 "Error reading file"

5004 "CRC mismatch, loading terminated"

5003 "Invalid .HEX file format"

5043 "Address out of range"

5078 "End address should be greater than start address"

5151 "Invalid file format"

5007 "Error writing file"

6899 "Cannot load file '%s': buffer #%u does not exist"

6900 "Cannot load file '%s': sub-level #%u does not exist"

7019 "Unable to open project file: '%s'.\n\nAfter start, the programmer attempts to load the most recent project. This error means that the project file does not exist on disk."

9.1.2 Error Addresses

5189 "Device start address (0x%LX) is too large.\nMax. address is 0x%LX."

5190 "Device end address (0x%LX) is too large.\nMax. address is 0x%LX."

5191 "Buffer start address is too large"

4024 "Address %s is out of range (%s...%s)"

4106 "File format does not allow addresses larger than 0xFFFFFFFF"

4019 "Address in device: 0x%08X, Address in buffer: 0x%08X\n"

6626 "Buffer start address must be even"

6627 "Device start address must be even"

6628 "Buffer end address must be odd"

8002 "Buffer named '%s' already exists. Please choose another name for the buffer."

9.1.3 Error sizes

6372 "Buffer size is too small for selected split data option"

6495 "Requested buffer size (%lu) is too large"

6441 "Size of file is greater than buffer size:\nAddr = %08lX, length = %u"

6431 "Source block does not fit into destination sub-level"

6859 "File size is %u bytes that is less than header size (%u bytes), loading terminated. Probably, you have specified an invalid file format."

4107 "Cannot allocate %Lu MBytes for the buffer, maximal buffer size is %Lu MBytes"

5192 "Invalid number: '%s'"

.

9.1.4 Error command-line option

5329 "/%s command-line option: Device name required"

5330 "/%s command-line option: Missing file name"

5331 "/%s command-line option: Missing file format tag"

5332 "/%s command-line option: Invalid file format tag"

5333 "Command line: unable to determine the file format"

5334 "/%s command-line option: Invalid address value"

4104 "Command-line option /I ignored because /A option is not specified"

9.1.5 Error Programming option

6409 "Invalid programming function or menu name:\n'%s'"

6410 "Invalid programming option name '%s'"

6902 "Invalid '%s' programming option value string: '%s'"

6411 "Programming option '%s' cannot be changed"

6412 "Programming option string is too long.\nMax. length is %u."

6854 "Programming option '%s' has type of '%s'. Use '%s()' script function to get the value of this option."

5188 "Value %.2f is out of range of %.2f...%.2f for programming option '%s'"

6561 "Value %ld is out of range of %ld...%ld for programming option '%s'"

4001 "Not all of the saved auto-programming functions were restored. Check the auto-programming functions list."

9.1.6 Error DLL

6499 "Cannot find bit resource with id 0x%X in DLL:\n'%s'"

6500 "Error handling bit resource with id 0x%X in DLL:\n'%s'"

6502 "Unable to find device '%s' in DLL:\n'%s'"

9.1.7 Error USB

4015 "USB device driver error 0x%04X in '%s'.\n\nCannot recover from this error, exiting.\n\nPlease check if the programmer power is on. If yes, disconnect the USB cable from computer and connect it again, then restart the %s shell."

4016 "All sites reported USB device driver error.\n\nCannot recover from this error, exiting.\n\nPlease check if the programmer(s) power is on. If yes, disconnect the USB cable from computer and connect it again, then restart the %s shell."

4017 "The following site(s):\n\n%s\n\nreported USB device driver error.\n\nThese site(s) will be removed from the gang programming process.\n\nPlease check if the programmer(s) power is on. If yes, disconnect the USB cable from computer and connect it again, then restart the %s shell."

9.1.8   Error programmer hardware

6546   "Source area does not fit into destination address space"

4005   "Attempt to read memory beyond buffer end: Addr = %s, len = %u bytes"

6988   "Unable to establish connection with programmer hardware. Please check if:\n\n"

4006   "Attached programmers have duplicate serial number '%s'"

4010   "This programmer with serial number '%s' has been already assigned the site number = %u"

4011   "This gang programmer with serial number '%s' has been already assigned the site numbers = %u..%u"

4013   "The programmers attached are of different types and cannot be used for gang mode.\n\nExiting."

4014   "ExecFunction() does not work in Gang mode"

4020   "%s reported hardware error 0x%X, error group 0x%X. If problem persists, please contact Phyton."

4000   "The attached programmer with id = %u is not supported"

4102   "Device programming countdown value is zero%s"

9.1.9   Error internal

6527   "Internal error:\nCORE() for %s %s returned NULL.\nPlease contact your %s distributor."

4025   "Internal Error: Unable to allocate %u bytes for the buffer. Please contact Phyton."

9.1.10   Error confiquration

6503   "No programmer configuration files found (prog.ini)"

5325   "The device type '%s %s' stored in configuration "
       "or choosen from script file function 'SetDevice()' is not supported by %s.\n"
       "The device '%s %s' will be selected.\n"
       "Use 'Configure / Select device' to choose the device "
       "you need to operate on."

4002   "The '%s' configuration option has been set to an illegal state due to the data read from file. Setting this option to its default state ('%s')."

9.1.11   Error device

5326   "Device selection error"

4018   "Device '%s' is not supported by the %s. Please choose another device."

9.1.12 Error check box

6852   "Error in check box option specification string: '=' expected"

6853   "Cannot find check box option string '%s'"

9.1.13 Error mix

5195   " Number of repetitions cannot be zero"

5206   "The 'View only' option is on; editing disabled. Click the 'View' button on toolbar to enable editing."

6501   "No power-on tests defined in:\n'%s'"

6903   "'%s' is a sub-menu name, not a function name"

6401   "No more occurences"

6387   "Invalid fill string"

5172   "Checksum = %08lX"

5311   "No more mismatches"

9.1.14 Warning

5338   "Warning: JEDEC file has no file CRC"

5339   "Warning: JEDEC file has invalid CRC"

6933   "Warning: no 'file end' record in file"

6845   "Attention! The %s %s device must be inserted into the programmer's socket shifted by %d row(s) relative to the standard position as shown in the Device Information window."

6846   "Attention! Insert device into socket shifted by %d row(s) as shown on the picture."

9.2   Expressions

Expressions are mathematical constructs for operations on one or more operands.

When a number is required, you may use an expression; ChipProg-02 will accept the value expression. For example, when using the **Modify** command in the **Buffer** window, you can enter the new value in the form of a number or arithmetic expression.

Interpreting the expression result

The expression result is interpreted in accordance with the context in which it is used.

In the dialog box, when an address is required, the program tries to interpret the expression's value as the address. If you enter a variable name, the result of the expression will be the variable's address but not the value of the variable.

If the dialog expects a number to be entered, the expression's value will be interpreted as a number (for example, the **Modify Memory** dialog box of the **Buffer Dump** window). If you enter a variable name there,

then the result will be the value of the variable, but not its address.

Nonetheless, you can follow the default rules:

If you need to use the variable's **value**, where an address is expected, then you can write something like `var + 0`. In this case, the variable's value will be used in the expression.

If you need to use the variable **address**, apply the **&** (address) operation, that is, `&var`.

9.2.1   Operations

The program supports all arithmetic and logical operations valid for the C language, as well as pointer and address operations:

Designation	Description
`( )`	Brackets (higher priority)
`[ ]`	Array component selector
`.`	Structure component or union selector
`->`	Selection of a structure component or a union addressed with a pointer
`!`	Logical negation
`~`	Bitwise inversion
`-`	Bitwise sign change
`&`	Returns address
`*`	Access by address
`(type)`	Explicit type conversion
`(sizeof)`	(returns size of operand, in bytes)
`*`	Multiplication
`/`	Division
`%`	Modulus operator (produces the remainder of an integer division)
`+`	Addition
`-`	Subtraction
`<<`	Left shift
`>>`	Right shift
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than

>=		Greater than or equal to
==		Equal to
!=		Not equal to
&		Bitwise AND
^		Bitwise XOR
|		Bitwise OR
&&		Logical AND
||		Logical OR
=		Assignment

The types of operands are converted in accordance with the ANSI standard.

The results of logical operations are 0 (false) or 1 (true).

Allowed type conversions:
- Operands can be converted to simple types (char, int, ... float).
- Pointers can be converted to simple types (char *, int *, ... float *) and to structures or unions.
- The word "struct" is not necessarily (MyStruct *).

9.2.2   Operands

By default, numbers are treated as decimals. Integers should fit into 32 bits; floating point numbers should fit into the single precision format (32 bits).

The following formats are supported:

1) Decimal integer.

Example: 126889

2) Decimal floating point.

Examples: 365.678; 2.12e-9

3) Hexadecimal.

<%CM%> understands numbers in C format and assembly format.

Examples: 0xF6D7; 0F6D7H; 0xFFFF1111

4) Binary.

Binary numbers must end with 'B'.

Examples: 011101B; 111111111111111000011B

5) Symbol (ASCII).

Examples: 'a'; 'ab'; '$B%8'.'.

9.2.3 Expression Examples

```
#test#i + #test#j << 2
(unsigned char)#test#i + 2
sizeof(##array) > 200
```

```
main

i + j << 2 / :CW0x1200

(unsigned char)i + 2

sizeof(array) > 200

(a == b && a <= 4) || a > '3'

sptr -> Member1 -> a[i]

*p

*((char *)ptr)
```

9.3   Scripting Reference

Description of Script Language

Script Language Built-in Functions

Script Language Built-in Variables

Alphabetical List of Script Language Built-in Functions and Variables

9.3.1   Scripting Language Description

ChipProg-02 scripting language is similar to C programming language. If you are familiar with C, you can proceed to the section describing the differences between the script language and the C language.

Here are the links to the sections of this scripting language manual.

General Syntax of Script Language
Basic Data Types
Data byte order
Operations and Expressions
Operators
Functions
Descriptions
Directives of the Script File Language Preprocessor
Predefined Symbols in the Script File Compilation

9.3.1.1   Difference Between Scripting and C Languages

The script files are written in a C-type language and you should not expect it to meet standards. Many features are not supported because they are not necessary and complication of the language can cause compiler errors (the script file language compiler is not a simple thing).

• Pointers are not directly supported. But arrays are supported, therefore a pointer can always be built

from an array and element number. Note that, for example, string operation functions, such as strcpy, receive a string and a byte number (index) as parameters, which form the pointer. In function declarations, index is equal to zero by default.

- Pointers to functions are not supported. If necessary, a table call can always be replaced with the **switch** operator.
- Multidimensional arrays are not supported. If it is necessary, you can write a couple of functions, such as:

```
int GetElement(int array[], int index1, int index2);
void SetElement(int array[], int index1, int index2, int value);
```

- Structures (and unions) are not supported. In fact, you can always do without structures. Structures may be required for API Windows and user DLLs operations, but as a rule only experienced programmers should do it, such as those who know how to reach structure elements. As a tip, there are functions, such as memcpy, which receive a void "pointer").
- Enumerated types (**enum**) are not supported #define.
- Preprocessor macros, such as **#define half(x) (x / 2)**, are not supported. The same operations can be done with functions.
- Conditional operators such as **x = y == 2? 3 : 4;**, are not supported; the operator "comma" outside variable declaration is not supported. For example,

int i = 0, j = 1; is supported, but
for (i = 0, j = 1; ...) is not supported.

- **User** functions with a variable amount of parameters are not supported. However, there are many system functions, such as printf, with a variable number of parameters.
- Declaration of **user** function parameters such as **void array[]** is not supported. The system functions such as memcpy, have such parameters.
- Logical expressions are always fully computed. It is very important to remember it, as a situation like

```
char array[10];
if (i < 10 && array[i] != 0)
  array[i] = 1;
```

will cause an error at the execution stage, if i is greater than 9, because the expression of array[i] will be computed. In a standard compiler such an expression is not computed, because the condition of **i > 10** would cancel any further processing of the expression.

- Constant expressions are always computed during execution. For example, **int i = 10 * 22** will be computed not during compilation, but during execution.
- The **const** key word is absent.
- Static variables cannot be declared inside functions.

But

- Variables can be declared anywhere, not just in front of the first executed operator. For example:

```
void main()
{
GlobalVar = 0;
int i = 1;        // will be OK as in C++
}
```

- Nested comments are allowed.
- Expressions like **array = "1234"** are allowed.
- Default parameter values in declared functions, as in C++, are allowed. For example, **void func(char array[],int index = 0);**. Expressions can also serve as default values, for example **void func(char array[], int index = func1() + 1);**.
- Expressions in global variable initializers are allowed. For example:

```
float table[] = { sin(0), sin(0.1) };

void main()
{
```

```
    ...
}
```

9.3.1.2 Scripting Language Syntax

[Format](#)
[Comments](#)
[Identifiers](#)
[Reserved words](#)
[Integer Constants](#)
[Long integer constants](#)
[Floating-point Constants](#)
[Character Constants](#)
[String Constants](#)

9.3.1.2.1 Format

Spaces, tabs, line advance and page advance symbols are used as separators. You can use any number of these separator symbols.

9.3.1.2.2 Comments

Comments begin with the pair of the */* symbols and end with the pair of the **/* symbols.

Comments are allowed wherever the spaces are allowed.

The one-line comments (*//*) are supported. The part of the line following the one-line comment symbol is ignored.

Note. Only the one-line comments are allowed in the line that contains the #define directive.

Examples:

```
// The one-line comment

/*    The multi-line comment */
```

9.3.1.2.3 Identifiers

Identifiers are used as the names of variables, functions and data types.

The allowable symbols are: digits from 0 to 9, the Latin lower and upper case letters a - z, A - Z and the underscore symbol (_).

A special case is accessing the names built in <%CM%>, for example, a special function register. Such names are preceded by the dollar mark, for example, **$SP**, and can be used in the program while not being declared. Identifiers shall comply with the following rules:

The first symbol can not be a digit.
The upper and lower case letters are discriminated.
An identifier can consist of up to 48 symbols.

Examples:

```
NAME1  name1  Total_5
```

9.3.1.2.4 Reserved w ords

break	extern	return
case	float	short
char	for	sizeof
continue	goto	switch
default	if	unsigned
do	int	void
else	long	while

9.3.1.2.5 Integer constants

Decimal constants

Numbers from 0 to 9.

Examples:

```
12
111
956
1007
```

Hexadecimal constants

Numbers from 0 to 9; letters a-f or A-F for the values of 10 to 15. The hexadecimal contents shall begin with 0x or 0X.

Examples:

```
0x12 = 18 (decimal);
0x2f = 47 (decimal);
0xA  = 10 (decimal);
```

Binary constants

Numbers 0 and 1. The binary constants shall end in b or B.

Examples:

```
010011101b = 0x9D (hexadecimal) = 157 (decimal);
0101B = 5
```

Note. If the value exceeds 65535, then it will be presented as the long integer.

9.3.1.2.6 Long integer constants

Latin letter l or L following the constant explicitly defines long integer constants. The explicit definition of a long constant is useful, for example, for transforming the type of operand into the long type value.

Examples:

```
Long decimal constant:       12l        12 (decimal)
                             956L      956 (decimal)
Long hexadecimal constant:   0x12l      18 (decimal)
                             0xA3L     163 (decimal)
```

9.3.1.2.7 Floating-point constants

A floating-point constant consists of the following parts:
•	Integer part, which is the sequence of numbers
•	Decimal point
•	Fractional part, which is the sequence of numbers
•	Exponential symbol e or E
•	Exponential in the form of an integer constant (can have sign)

Any of the two parts (but not both at the same time) of the following pairs can be omitted:
•	Integer or fractional part
•	Decimal point or symbol e (E) and the exponential in the form of an integer constant

Examples:

```
 345.     =     345 (decimal);
   3.14159 =       3.14159 (decimal);
   2.1E5   = 210000 (decimal);
    .123E3 =     123 (decimal);
4037e-5    =       .04037 (decimal).
```

9.3.1.2.8 Character constants

A character constant may consist of one ASCII code character enclosed within the apostrophes. Also, you may specify the character by its hexadecimal value of exactly two hexadecimal digits preceded by characters '\x'.

Examples:

'A' 'a' '7' '$' '\x02' '\x88'

Special (control) character constants

```
New line (line feed)        HL (LF)    '\n'
Horizontal tabulation       HT         '\t'
Vertical tabulation         VT         '\v'
Backspacing                 BS         '\b'
Carriage return             CR         '\r'
Form feed                   FF         '\f'
Backslash                   \          '\\'
Apostrophe                  '          '\''
Quotation marks             "          '\"'
Zero character (null)        NUL       '\0'
```

Note. The character constants are considered to be the int-type data.

9.3.1.2.9 String constants

A string constant is the quoted sequence of the ASCII code characters: "...".

A string constant is the quoted character array; its type is char[].

To mark the end of string, the compiler places the null symbol **'\0'** in the end of each string.

If you need to include the quotation mark (") in a string, then enter the backslash (\) before the quotation mark. Any special character constants preceded by the backslash (\) can be included in the string.

A symbol can also be presented by its hexadecimal value (exactly two hexadecimal digits) preceded by the symbols of **'\x'**.

The string constants following in sequence are interpreted as one string constant. This is useful for the advance of the constant part to the next line, for example:

```
printf("Line 1\n"
       "Line 2");
```

Examples:

```
"This is the character string"
"A"
"1234567890\x33"
"_____"
""
```

9.3.1.3   Basic Data Types

The script file compiler supports the following data types:

```
------------------------------------------------------------
signed char      8    1    -128...+127
unsigned char    8    1    0...255
signed short     16   2    -32768...+32767
unsigned short   16   2    0...65535
signed int       16   2    -32768...+32767
unsigned int     16   2    0...65535
signed long      32   4    -2147483648...2147483647
unsigned long    32   4     0...4294967295
float            32   4    +/-1.17549435E-38...+/-3.40282347E+38
```

The "pure" int type coincides with the signed int type.
The long type is equivalent to the signed long.
The short type is equivalent to the signed short.
The char type is equivalent to the signed char.

9.3.1.4   Data byte order

Data byte order

All many-byte data is stored in the memory in the "little engine" format, that is, the low byte is allocated at the low address and the high byte is allocated at the high address in accordance with the 80x86 processor architecture. For experienced programmers, it is useful to know this, if they want to use Windows API and DLL functions access

9.3.1.5   Operations and Expressions

Expressions

An expression consists of one or more operands and operation symbols.

Examples:

a++

b = 10

x = (y * z) / w

Note. Any expression ending with semi is the operator.

[Operand Metadesignation](#)

[Arithmetic Operations](#)

[Assignment Operations](#)

[Relation Operations](#)

[Logical Operations](#)

[Bit Operations](#)

[Array Operations](#)

[Other Operations](#)

[Operation Execution Priorities and Order](#)

[Operand Execution Order](#)

[Arithmetic Conversions in Expressions](#)

9.3.1.5.1 Operand Metadesignation

Some operations require specific operand types. The type of operand is indicated by one of the following letters:

  e - any expression

  v - any expression referring to the variable, to which a value

     can be assigned. Such expressions are called the address ones.

The prefix indicates the type of expression. For example, ie indicates any integer expression. All the possible prefixes are as follows:

  i

  a - the arithmetic expression (the integer number, symbol or

     floating-point number)

  f - the function

Note.

9.3.1.5.2 Arithmetic Operations

+   Usage: ae1 + ae2

Sum of ae1 and ae2.

Example:

i = j + 2;

Sets i equal to j plus 2.

-   Usage: ae1 - ae2

Subtraction of ae1 and ae2.

Example:

i = j - 3;

-   Usage: -ae

Example:

*   Usage: ae1 * ae2

Product of ae1 and ae2.

Example:

z = 3 * x

/

Quotient of ae1 and ae2.

Example:

i = j / 5;

%   Usage: ae1 % ae2

Remainder (modulus division) of the division of ae1 by ae2.

Example:

minutes = time % 60;

Note. Execution of the ++ and -- operations produces side effects; the value of variable used as an operand changes.

++ Usage: iv++

Increasing iv by 1. The value of this expression is the value of ie

before increasing.

Example:

j = i++;

++ Usage: ++iv

Increasing iv by 1. The value of this expression is the value of ie

after increasing.

Example:

i = ++j;

-- Usage: iv--

Decreasing iv by 1. The value of this expression is the value of ie

before decreasing.

Example:

j = i--;

-- Usage: --iv

Decreasing iv by 1. The value of this expression is the value of ie

after decreasing.

Example:

i = --j;

9.3.1.5.3 Assignment Operations

Note. The value of expression containing the assignment operation is the value of the left operand after the assignment.

= Usage: v = e

The value of e is assigned to variable v.

Example:

x = y;

Note. The following operations combine arithmetic or bit-by-bit operations with the assignment operation.

+= Usage: av += ae

Increasing av by ae.

Example:

y += 2;

-= Usage: av -= ae

Decreasing av by ae.

Example:

     x -= 3;

*=   Usage: av *= ae

    Multiplication of av by ae.

    Example:

     timesx *= x;

/=   Usage: av /= ae

    Division of av by ae.

    Example:

     x /= 2;

%=   Usage: iv %= ie

    The value of iv in modulus ie.

    Example:

     x %= 10;

>>= Usage: iv >>= ie

    The right ie bit shift of the iv binary form.

    Example:

     x >>= 4;

Usage: iv <<= ie

    The left ie bit shift of the iv binary form.

    Example:

     x <<= 1;

&=   Usage: iv &= ie

    The bit-by-bit AND operation of the iv and ie binary forms.

    Example:

     remitems &= mask;

^=   Usage: iv = ie

    The bit-by-bit exclusive OR operation of the iv and ie binary forms.

    Example:

     control ^= seton;

|=   Usage: iv |= ie

    The bit-by-bit OR operation of the iv and ie binary forms.

    Example:

     additems |= mask;

9.3.1.5.4 Relation Operations

Note. Logical False is presented by integral zero, and logical True is presented by any integer other than zero.

The expressions that contain the relation operations or logical operations have the values of 0 (False) or 1 (True).

== Usage: ie1 == ie2

True, if ie1 is equal to ie2; False otherwise.

Example:

```
if (i == 0) break;
```

!= Usage: ie1 != ie2

True, if ie1 is not equal to ie2.

Example:

```
while (i != 0)
   i = func;
```

< Usage: ae1 < ae2

True, if ae1 is less than ae2.

Example:

```
if (x < 0)
   printf ("negative");
```

<= Usage: ae1 <= ae2

True, if ae1 is less than or equal to ae2.

 Usage: ae1 > ae2

True, if ae1 is larger than ae2.

Example:

```
if (x > 0)
   printf ("positive");
```

>= Usage: ae1 >= ae2

True, if ae1 is larger than or equal to ae2.

9.3.1.5.5 Logical Operations

! Usage: !ae

True, if ae or pe is false.

Example:

```
 if (!good)
   printf ("not good");
```

|| Usage: e1 || e2

checked. The value of e2 will be checked only, if e1 is False. The expression   will be

True, if e1 or e2 is True.

Example:

```
 if(x < A || x > B) printf
   ("out of range");
```

&&

Logical AND operation of e1 and e2. At first, the value of e1

is checked. The value of e2 will be checked only, if e1 is True.

The expression will be True, if e1 and e2 are True.

Example:

```
 if (a ! = 0 && b > 7)
   n++;
```

9.3.1.5.6 Array Operations

[] Usage: name[ie]

The expression value is the number equal to the value of the element

number ie of the name array. The array elements are numbered beginning from   0.

Example:

```
 arname[i] = 3;
```

To assign 3 to the array element i.

Note the first element as described by the expression of

arname[0].

9.3.1.5.7 Bit Operations

~ Usage: ~ie

One's complement of the value ie. The expression value contains ones in

all those bits, in which ie contains 0, and contains 0 in all

those bits, in which ie contains ones.

Example:

opposite = ~mask;

>> Usage: ie1 >> ie2

The right ie2 shift of the ie1 binary form.

The shift may be arithmetic (that is, the bits cleared from the left

assume the value of the sign bit) for the signed numbers and

logical for the unsigned numbers (the bits cleared from the left are

filled with zeroes).

Example:

x = x >> 3;

<< Usage: ie1 << ie2

The left ie2 bit left of the ie2 binary form.

The bits cleared from the right are filled with zeroes.

Example:

fourx = x << 2;

&   Usage: ie1 & ie2

The bit-wise AND operation of the ie1 and ie2 binary forms. The expression

value assumes 1 in all those bits, in which both ie1 and ie2 contain

1, and assumes 0 in all other bits.

flag = ((x & mask) != 0);

| Usage: ie1 | ie2

The bit-wise OR operation of the ie1 and ie2 binary forms. The expression

value assumes 1 in all those bits, in which either ie1 or ie2 contain

1, and assumes 0 in all other bits.

Example:

attrsum = attr1 | attr2;

^   Usage: ie1 ^ ie2

The bit-wise exclusive OR operation of the ie1 and ie2 binary forms.

The expression value contains 1 in all those bits, in which ie1 and

ie2 contain different binary values, and the expression value

contains 0 in all other bits.

Example:

diffbits = x ^ y;

9.3.1.5.8 Other Operations

sizeof Usage: sizeof(e)

> The number of bytes required for allocation of e-type data. If e

> describes the array, then e means the whole array, and not only the

> address of the first element, as in other operations.

(type) Usage: (type)e

> The value of e is converted into the data type.

> Example:

> x = (float)n / 3;

> The integer value of the variable n is transformed into

> the floating-point number before dividing by 3.

()   Usage: fe(e1, e2,..., eN)

> The fe function is called with the arguments e1, e2,..., eN.

> order from

> Example:

> x = sqrt(y);

9.3.1.5.9 Operation Execution Priorities and Order

> Priorities are the same for each group of operations listed in the table below. The higher the priority of operation, the higher is its place in the table.

> If there are no brackets and the operations are related to the same group, then the order of execution determines the operation and operand grouping (from left to right or from right to left).

> Examples

> The expression of a * b / c is equivalent to the expression of (a * b) / c,

> as the operations are executed from left to right.

> The expression of a = b = c is equivalent to the expression of a = (b = c),

> as the operation is executed from right to left.

[] Array element selection

! Logical negation   From right to left (RL)

~ Bit-by-bit negation

- Sign change

++ Increasing by one

-- Decreasing by one

(type) Type conversion

sizeof Determining of size in bytes

* Multiplication   LR

/ Division

% Modulus division

+ Addition   LR

- Subtraction

<< Left shift   LR

  Right shift

< Less than   LR

<= Less than or equal to

> Larger than

>= Larger than or equal to

== Equal to   LR

!= Not equal to

& Bit-by-bit AND operation   LR

^ Bit-by-bit exclusive OR operation   LR

| Bit-by-bit OR operation   LR

&& Logical AND operation   LR

|| Logical OR operation   LR

=

*= /= %= += -=

<<= >>= &= ^= |=

9.3.1.5.10 Operand Execution Order

The operands are normally executed from left to right.

. If you assign a value to a variable in any expression (including the function call), do not use this variable again in the same expression.

Example:

y = (x = 5) + (++x);

9.3.1.5.11 Arithmetic Conversions in Expressions

First, every char-type operand is converted into the int-type value, and the unsigned char-type operand is converted into the unsigned int-type value.

Then, if one of the operands is of the float type, then the other will be converted into the float-type value and the result will be of the float type.

Otherwise, if one of the operands is of the unsigned long type, then the other will be converted into the unsigned long-type value and the result will be of the same type.

Otherwise, if one of the operands is of the long type, then the other will be converted into the long-type value and the result will be of the same type.

Otherwise, if one of the operands is of the long type, and the other is of the unsigned int type, then both operands will be converted into the unsigned long-type value and the result will be of the same type.

Otherwise, if one of the operands is of the unsigned type, then the other will be converted into the unsigned-type value and the result will be of the same type.

Otherwise, both operands should be of int type and the result will be of the same type.

9.3.1.6   Operators

Format and nesting

Operator label

Composite operator

Operator-expression

Operator Break

Operator Continue

Operator Return

Operator Goto

Conditional Operator If-Else

Operator Switch

Cycle Operator While

Cycle Operator Do-While

Cycle Operator For

9.3.1.6.1 Format and nesting

Format and nesting

Format. One operand can occupy one or more lines. Two or more operands can be located in one line.

Nesting. The execution control operators (if, if-else, switch, while, do-while and for) can be nested in each other.

9.3.1.6.2 Operator label

Operator label

The label can be placed before any operator, which makes it possible to go to this operator with the help of the "goto" operator.

A label consists of an identifier followed by the colon (:). The definition domain of the label is the specified function.

Example:

```
next: x = 3;
```

9.3.1.6.3 Composite operator

Composite operator

The composite operator (block) consists of one or more operators of any type enclosed in the brackets ({ }).

There shall be no semicolon (;) behind the closing bracket.

Example:

```
{
  x = 1;
  y = 2;
  z = 3;
}
```

9.3.1.6.4 Operator-expression

Any expression, which ends with the semicolon (;), is the operator. Refer to the following examples of operators-expressions.

Assignment operator

```
    Identifier = expression;
```

Example:

```
    x = 3;
```

Function call operator

```
    Function_name (argument1,..., argumentN);
```

Example:

```
    fclose(file);
```

Empty operator

```
    Consists only of semicolon (;).
    It is used to identify the empty body of the control operator.
```

9.3.1.6.5 Operator Break

Syntax:

break;

Stops execution of the nearest nested external operator switch, while, do, or for. Control is transferred to the operator following the operator being completed. One purpose of this operator is to complete the cycle, when specific value is assigned to the variable.

Example:

```
for (i = 0; i < n; i++)

  if (a[i] == 0)

      break;
```

9.3.1.6.6 Operator Continue

Syntax:

continue;

Transfers control to the beginning of the nearest external operator of the cycle while, do, or for, which starts the next iteration. This effects produced by this operator are opposite to those of the break operator.

Example:

```
for (i = 0; i < n; i++)

{

  if (a[i] == 0) continue;

  a[i] = b[i];

}
```

9.3.1.6.7 Operator Return

Syntax:

return;

Stops execution of the current function and returns control to the function that called it.

expression return;

Stops execution of the current function and returns control to the program that called it, together with the expression value.

Example:

```
 return x + y;
```

9.3.1.6.8 Operator Goto

Syntax:

```
goto label;
```

Control is unconditionally transferred to the operator with the label "label". It is used to exit from the nested control operators. The scope of the label is limited within the current function.

Example:

```
    goto next;
```

9.3.1.6.9 Conditional Operator If-Else

Syntax:

if (expression)

  operator

If the expression is True, then the operator will be executed. If the expression is False, then nothing will happen.

Example:

```
 if (a == x) temp = 3;
```

if (expression)

  operator1

else

  operator2

If the expression is True, then operator1 will be executed and control will be transferred to the operator following operator2 (which means that operator2 will not be executed).

If the expression is False, then operator2 will be executed.

The "else" part of the operator can be omitted. That is why ambiguity may arise in the nested operators with omitted "else" part. In this case, else is related to the nearest preceding operator in the same block that does not have the "else" part.

Examples:

1) The "else" part relates to the second if operator:

```
if(x > 1)
    if (y == 2)
        z = 5;
    else
        z = 6;
```

2) The "else" relates to the first if operator:

```
if (x > 1)
{
  if (y == 2) z = 5;
}
else z = 6;
```

3) The nested if operators:

```
if (x == 'a') y = 1;
else
  if (x == 'b')
  {
    y = 2;
    z = 3;
  }
  else
    if (x == 'c') y = 4;
else
    printf("ERROR");
```

9.3.1.6.10 Cycle Operator While

Syntax:

while (expression)

 operator

If the expression is True, then the operator will be executed until the expression becomes False.

If the expression is False, then control is passed to the next operator.

Note. The value of the expression is determined before executing the operator. Therefore, if the expression is False from the very beginning, then the operator will not be executed at all.

Example:

```
while (k < n)
{
  y *= x;
  k++;
}
```

9.3.1.6.11 Cycle Operator Do-While

Syntax:
```
do
    operator
while (expression);
```

If the expression is True, then the operator will be executed and the expression value will be calculated. This will be repeated until the expression becomes False.

If the expression is False, then control is passed to the next operator.

Note. The expression value is determined after the operator is executed. Therefore, the operator is executed at least once.

The do-while operator checks the condition in the end of the cycle.

The while operator checks the condition in the beginning of the cycle.

Example:
```
x = 1;
do
    printf('%d\n", pow(x, 2));
while (++x <= 7);
```

9.3.1.6.12 Cycle Operator For

Syntax:
```
for (expression1; expression2; expression3)
  operator
```

Expression1 describes the cycle initialization. Expression2 is checking the condition of the cycle completion. If it is True, then:

the "for" operator of the cycle body will be executed;

expression3 will be executed.

Everything will be repeated until expression2 becomes False.

If it is False, then the cycle will be finished and control will be passed to the next operator.

Expression3 is calculated after each iteration.

The "for" operator is equivalent to the following operator sequence:

expression1;

while (expression2)

{

  operator

  expression3;

Example:

for(x = 1; x <= 7; x++)

    printf("%d\n", pow(x, 2));

In any of the three expressions, or in all three expressions of the operator, "for" may be absent, but the semicolons (;) separating them cannot be omitted.

If expression2 is omitted, then it will be considered True. The "for" operator (;;) is the endless cycle equivalent to the While(1) operator.

9.3.1.7 Functions

[Function Definition](#)

[Function Call](#)

[Function Main](#)

9.3.1.7.1 Function Definition

Functions are defined by description of the type of result, formal parameters and composite operator (block) that describe the actions carried out by the function.

Example:

```
int                     the type of result
   func(                 function name
        long a, char str[]   list of parameters, which describes the names and
types
        )
{                        composite operator
   // ...
   return 0;             returned value
}
```

The **return** operator can not return any value or return the value of the expression included in this operator.

The function, which does not return a value, shall be described as having type void.

One or several last parameters on the list can assume the default values. Examples:

```
int func(int x, int y = 0);
int f1(char s[], char s1[] = "null", int x = func(0));

void errmesg(char s[])
```

```
{
    printf{"***Error: %s", s);
    // the Return operator (explicit) is not required
}
```

9.3.1.7.2 Function Call

Syntax of a function call is as follows:

`function_name(e1, e2, ..., eN)`

Arguments that are not arrays (actual parameters) are transferred by value, that is, each expression e1, ..., eN is calculated and the parameter is transferred to the function. Arrays are transferred "by pointer", as shown in the example:

```
void func(char s[])
{
    s[0] = 2;
}
void main()
{
    char array[3];
    func(array);
}
```

The func function modifies the value of element0 of the "array" array declared in the main function, and not of its duplicate.

Pointers to functions (like all other pointers) are not supported.

9.3.1.7.3 The main Function

The script file operation commonly starts with the **main** function. The **main** function shall be declared as follows:

```
void main()
{
    ...
}
```

Note. The **main** function should not necessarily be included in a script file. If there is no **main** function, then the script file will be loaded into the memory and stay there with its global functions and data available to other script files.

9.3.1.8   Descriptions

Descriptions are used for variable definitions and to declare types of variable and functions defined

elsewhere. Descriptions are also used for defining new data types on the basis of existing data types.

A description can be an operator, if an initialized variable or array are described.

[Basic Types](#)

[Arrays](#)

[Local Variable Definition](#)

Global Variable Definition

9.3.1.8.1 Basic Types

Examples:

char c;

int x = 0;

The basic types are:

char   - character (one byte);

short   - short integer (word, 16 bit);

int   - integer (word, 16 bit);

unsigned - non-negative integer (of the same size as integer);

long   - long integer (word or double word);

float   - floating-point number (single precision);

void   - no value (used to neutralize the value

        returned by function)

The Short type is equivalent to the Int type and was introduced for generality.

Also, see Basic Data

9.3.1.8.2 Arrays

Only one-dimensional arrays are supported.

Example:

int a[50];

Variable a is the array consisting of 50 integer numbers.

9.3.1.8.3 Local Variable Definition

The automatic variable is temporary, because it loses its value upon the exit from the block. The domain of the variable is the block, where it is defined. Variables defined inside the block take precedence over the variables defined in the enclosing blocks. Example:

void func(char c)

{

 int i = 0;

 if (c == '0')

 {

   char i = 8;

```
    i++;
  }
  i++;
}
```

The local variable can be described everywhere within the function, as in C++.

Values of non-initialized local variables are undefined.

The function formal parameters are processed the same way as local variables.

Static variables inside the function are not implemented.

9.3.1.8.4 Global Variable Definition

Global variables

Example:

```
        int Global_flag;
```

Global variables are defined on the same level as functions, that is, they are not local in any block. They are initialized with 0, unless other initial value is explicitly defined. The scope is all script files currently being executed. Global variables should be described in all the script files that can access them.

Static variables

Example:

```
        static int File_flag;
```

Constant. The scope is the script file, in which the variable is defined. The static variables shall be described before they are used in the file for the first time.

[Variable Initialization](#)

[External Object Description](#)

9.3.1.8.4.1 Variable Initialization

Any variable, except for formal parameters, can be initialized upon definition.

Any permanent variable is initialized with 0, unless other initial value is explicitly defined.

Any expression can be used as the initial value.

Basic types

Examples:

```
  int i = 1 + j;
  float x = sin(_PI / 2);
```

Arrays

Examples:

  int a[] = {1,4,9,16,25,36};

  char s[20] = { 'a', 'b', 8 };

The values of array elements are listed in curly brackets.

If an array size is defined, then the values, which are not explicitly defined, will be equal to 0.

If an array size is omitted, then it will be determined by the amount of initial values.

Strings

Example:

  char s[] = "hello";

This description is equivalent to the description of

  char s[] = {'h','e','l','l','o','\0'};

9.3.1.8.4.2 External Object Description

Any type of external objects (for example, variables or functions) not defined explicitly in another script file, should be described explicitly.

Use the keyword Extern hen describing an external object.

Examples:

  extern int Global_var;

  extern char *Name;

  extern int func();

The length of external one-dimensional array can be omitted.

Example:

  extern float Num_array[];

Because all functions are defined on the external level, the adjective extern is not needed to describe a function inside the block and you can omit it.

9.3.1.9 Directives of the Script Language Preprocessor

If you use the # symbol as the first symbol in the program line, this line is the preprocessor (microprocessor) command line.

The preprocessor command line ends with the line advance symbol.

Identifier Change (#define)

Inclusion of Files (#include)

Conditional

9.3.1.9.1 Identifier Change (#define)

Syntax:

#define identifier line

Example:

#define Count 100

Changes each occurrence of the Count identifier in the program text to 100.

#undef identifier

Example:

#undef Count

Cancels any previous definition for the Count identifier.

9.3.1.9.2 Inclusion of Files (#include)

Note. You can put the #include command line everywhere in the program, but normally, all inclusions are located in the beginning of the source file text.

Syntax:

#include <file_name>

Example:

#include <system.h>

The preprocessor changes this line to the contents of the system.h file. The angle brackets indicate that the system.h file will be taken from the standard catalog. The directory, where CPI2-B1 is installed, and the list of directories specified in the "include-file directory" field in the Script Files dialog, are automatically used as the standard directory. If the file is not found in any of the standard directories, then the current directory will be checked.

#include "file_name"

Example:

#include "defs.h"

This structure operates the same way as the #include <system.h>, except that the compiler searches the current directory first.

9.3.1.9.3 Conditional Compilation

Preprocessor command lines are used for conditional compilation of various parts of the source text depending on external conditions.

Syntax:

#ifdef identifier

Example:

> #ifdef Debug

True, if the Debug identifier was defined earlier by the #define directive. Identifiers can also be defined in the Defines text box in the [Script Files dialog.](#)

Syntax:

#ifndef identifier

Example:

> #ifndef Debug

Syntax:

```
#else
#endif
```

If all previous checks of #if, #ifdef, or #ifndef show the True value, then the lines from #else to #endif will be ignored during compilation.

If those checks show the False value, then the lines from the check to #else (and if #else is missing, then from the check to #endif) will be ignored.

The #endif command ends the conditional compilation.

Example:

> #ifdef DEBUG   printf("Location: x = %d", x); #endif

9.3.1.10 Predefined Symbols in the Script File Compilation

The compiler automatically defines these symbols, as if they were defined by the #define directive.

Symbols that define the microcontroller family

One of the following symbols is defined:

__ARM   - for the ARM debuggers

- for the MCS-51 debuggers;

__MCS_96 - for the MCS-96 debuggers;

__PIC   - for the Microchip PIC debuggers.

9.3.2   Built-in Functions by Group

The script file system provides you with a large set of built-in functions intended for work with lines, files, for mathematical calculations, and access to the processor resources. The **system.h** file contains descriptions of these built-in functions. You should include the **system.h** file in the script file source text with the #include directive:

> #include <system.h>

You can use these built-in functions in the same way you use any function that you have defined.

[Buffer access functions](#)

Note. To get help on a function or variable, while editing the script source with the <%CM%> built-in editor, point that function/variable name with the cursor and hit **Alt+F1**.

9.3.2.1 Buffer access functions

9.3.2.1.1 CheckSum

unsigned long CheckSum(unsigned long start_addr, unsigned long end_addr, int addr_space);

Description

Calculates checksum for data in the addr_space memory{addr_sp} starting at start_addr and ending at end_addr. Checksum is calculated by simple addition of byte values.

Return Value

32-bit checksum.

Example

```
printf("%08lX", CheckSum(0, 0x1FFF, SubLevel(1, 0)));
```

9.3.2.1.2 GetByte

```
unsigned int GetByte(unsigned long addr, int addr_space);
```

Description

To read a byte from a specified address space{addr_sp} (parameter addr_space) at a specified address.

Returned value

Read byte or word.

Example

```
printf("%02X", GetByte(SubLevel(0, 0), 0x1F));
```

9.3.2.1.3 GetDword

```
unsigned long GetDword(unsigned long addr, int addr_space);
```

Description

To read a double word (32 bits) from a specified memory area{addr_sp} (parameter addr_space) at a specified address.

Returned value

Read double word.

Example

```
printf("%08lX", GetDword(0, 0x1F));
```

9.3.2.1.4 GetMemory

```
void GetMemory(void dest[], int n, unsigned long addr, int addr_space);
```

Description

To read n-byte memory block from a specified memory area{addr_sp} (parameter addr_space) at a specified address to the array dest.

Example

char array[20]; GetMemory(array, sizeof(array), 0x20, SubLevel(0, 0));

9.3.2.1.5 GetWord

unsigned int GetWord(unsigned long addr, int addr_space);

Description

To read a word (16 bits) from a specified memory area{addr_sp} (parameter addr_space) at a specified address.

Returned address

Read word.

Example

printf("%04X", GetWord(0, 0x1F));

9.3.2.1.6 LoadProgram

void LoadProgram(unsigned char file_name[], int format, int addr_space=AS_CODE, unsigned long start_addr=0);

Description

To download a program in the buffer{buffer} memory.

Parameters:

```
file_name - Name of the loaded file.
format    - Format of the loaded file. Character constants with the
            prefix F_ declared  in the mprog.h header file
            are provided for this parameter. To understand this
            better, open the Load Programm Dialog.
            and go through format names.
addr_space - address space{addr_sp} where the data is downloaded
            (0 by default).
start_addr - Load address. This parameter is used only for loading
            a file that is a binary memory image.
```

Example

LoadProgram("C:\\PROG\\TEST.HEX", F_HEX, SubLevel(1, 0));

9.3.2.1.7 MaxAddr

unsigned long MaxAddr(int addr_space);

Description

Returns the address of the address space{addr_sp} upper boundary.

9.3.2.1.8 MinAddr

unsigned long MinAddr(int addr_space);

Description

Returns the address of the address space{addr_sp} lower boundary.

9.3.2.1.9 ReloadProgram

void ReloadProgram();

Description

To reload the file that was the last to be loaded to the buffer. This is equivalent to "Re-Load" in the File menu.

9.3.2.1.10 SaveData

void SaveData(unsigned char file_name[], int format, int addr_space, unsigned long start_addr, unsigned long end_addr);

Description

To save memory area from buffer{buffer} in the file.

Parameters:

```
  file_name - Name of unloaded file.
  format    - Format of unloaded file. Character constants with
              the  prefix  F_ declared in the mprog.h header file
              are provided for this parameter.  To understand  this  better,
              open the Save Program Dialog and go through
              format names.
  addr_space - address space{addr_sp} where data is unloaded from.
  start_addr - Initial address of unloaded area.
  end_addr   - Final address of unloaded area (inclusive).
```

Example

SaveData("C:\\PROG\\TEST.HEX", F_HEX, SubLevel(0, 0), 0, 0x3FFF);

9.3.2.1.11 SetByte

void SetByte(unsigned long addr, int addr_space, unsigned int value);

 Description

To write value (byte) to a specified memory area{addr_sp} (parameter addr_space) at a specified address.

 Description

SetByte(0x2000, SubLevel(0, 1), 0xFF);

9.3.2.1.12 SetDevice

int SetDevice(char manufacturer[], char name[]);

 Description

Set device type. The manufacturer parameter is the device manufacturer name, name is the device name.

 Returned value

TRUE if the device is successfully selected, FALSE if it is not found.

 Example

SetDevice("Altera", "EP910");

9.3.2.1.13 SetDword

void SetDword(unsigned long addr, int addr_space, unsigned long value);

 Description

To write a double word (32 bits) to a specified memory area{addr_sp} (parameter addr_space) at a specified address.

 Example

SetDword(0x2000, 0, 0x12345678);

9.3.2.1.14 SetMemory

void SetMemory(void src[], int n, unsigned long addr, int addr_space);

  Description

To write n-byte memory block to a specified memory area{addr_sp} (parameter addr_space) at a specified address from the array src.

  Example

SetMemory("12345678", 8, 0x20, 0);

9.3.2.1.15 SetWord

void SetWord(unsigned long addr, int addr_space, unsigned int value);

  Description

To write a word (16 bits) to a specified memory area{addr_sp} (parameter addr_space) at a specified address.

  Example

SetWord(0x2000, 0, 0xFFFF);

9.3.2.2 Device programming control functions and variables

Here is the list of the functions that control programming scripts (alphabetic order):

AllProgOptionsDefault
ExecFunction
GetProgOptionBits
GetProgOptionFloat
GetProgOptionList
GetProgOptionLong
GetProgOptionString
mprintf
ProgOptionDefault
SetProgOption

Here is the list of the variables that controls programming operations in scripts (alphabetic order):

BlankCheck
BufferStartAddr
ChipEndAddr
ChipStartAddr

9.3.2.2.1 Function AllProgOptionsDefault

void AllProgOptionsDefault();

Description:

Set all the programming options to their default values.

9.3.2.2.2 Function ExecFunction

int ExecFunction(char func_name[], int buffer=0, int repetitions=1);

Description:

Perform the specified action (function) on device - programming, blank check, etc. The list of available functions is displayed in the upper left corner of the Program window.

Parameters:

func_name - function name, for example **"Blank Check"**. If you need to execute a function located in the pop-up menu, you should precede the function name with the menu name and separate them with '^' sign, e.g. "Data Memory^Program".

buffer - the buffer number.

repetitions - number of repetitions of the function.

Returned value:

For the value returned by **ExecFunction**, the header file **mprog.h** contains two constants:

EF_OK - function was completed successfully

EF_ERROR - there was an error while executing function. In this case, the error description is copied into the built-in variable LastErrorMessage.

Example:

```
if (ExecFunction("Blank Check") != EF_OK)
  printf("Error in blank check: %s", LastErrorMessage);
```

See also DialogOnError.

9.3.2.2.3 Function GangExecute

int GangExecute(int site, int buffer=0);

Description:

In the gang mode, launch the **Auto Programming** command on the socket, the number of which is specified by the **site** parameter (the first socket in the gang programmer has the number 0)**.** The buffer's number is specified by the parameter **buffer**. The default buffer number is 0.

A successful launch of the **GangExecute()** function returns 1; if the function fails it returns 0. Regardless of the **Auto Programming** result, immediately after launching the **GangExecute()** function, full control returns to the active script. In order to check the **Auto Programming** command completion, use the script functions **GangStatus()** or **GangWaitComplete().**

9.3.2.2.4 Function GangGetError

int GangGetError(int site, char s[]);

Description:

In the gang mode get an error message about the failure of the socket, the number of which is specified by the parameter **site** (the first socket in the gang programmer has the number 0). The error message (a string) dumps to the array with the pointer **s**. If no single error has occurred during the programming session the first byte in the error string will be 0 (zero).

9.3.2.2.5 Function GangStatus

int GangStatus(int site);

Description:

In the gang mode get the status of the operation on the socket, the number of which is specified by the **site** parameter (the first socket in the gang programmer has the number 0). The function call returns the status string, two bits of which define the operation statuses:

If the bit **GS_EXECUTING =1** this indicates that **Auto Programming** is still in process;
If the bit **GS_FAILED =1** this indicates an **Auto Programming** failure.

9.3.2.2.6 Function GangWaitComplete

void GangWaitComplete(int site);

Description:

In the gang mode, wait for completion of the **Auto Programming** operation on the socket, the number of which is specified by the **site** parameter (the first socket in the gang programmer has the number 0). Regardless of the operation result, a call of this function returns control to the script only upon completion of the **Auto Programming** operation.

9.3.2.2.7 Function GetBadDeviceCount

unsigned long GetBadDeviceCount(int site=0);

Description:

In the gang mode get the current number of devices that could not be successfully programmed or did not pass verification in the socket, the number of which is specified in the **site** parameter (the first socket in the gang programmer has the number 0). Each socket in the gang programmer has a virtual counter **"Bad"** that increments the variable after each programming cycle failure. The **"Bad"** counter display is accessible in the **Statistics** tab in the **Program Manager** window.

9.3.2.2.8 Function GetGoodDeviceCount

unsigned long GetGoodDeviceCount(int site=0);

Description:

In the gang mode, get the current number of the devices successfully programmed in the socket, the number of which is specified by the **site** parameter (the first socket in the gang programmer has the number 0). Each socket in the gang programmer has a virtual counter **"Good"** that increments the variable after each successful device programming cycle. The **"Good"** counter display is accessible in the **Statistics** tab in the **Program Manager** window.

9.3.2.2.9 Function GetProgOptionBits

unsigned long GetProgOptionBits(char option_name[]);

Description:

Returns current value of the **option_name** programming option. The option must be of type 'Bits' - a list of options; each of them can be checked or unchecked. Example: "Sectors" option of the Fujitsu MBM29LV008BA device.

9.3.2.2.10 Function GetProgOptionFloat

float GetProgOptionFloat(char option_name[]);

Description:

Returns current value of the **option_name** programming option. The option must be of type 'Long' - a floating-point number. Example: "Vcc" option of the Microchip PIC16F628A device.

9.3.2.2.11 Function GetProgOptionList

unsigned int GetProgOptionList(char option_name[]);

Description:

Returns current value of the **option_name** programming option. The option must be of type '**List**' - a menu-like list of strings. Example: "WDT" option of the Microchip PIC16F628A device.

9.3.2.2.12   Function GetProgOptionLong

long GetProgOptionLong(char option_name[]);

Description:

Returns current value of the **option_name** programming option. The option must be of type '**Long**' - a 32-bit integer. Example: "Tpgm" option of the Atmel ATF2500C device.

9.3.2.2.13   Function GetProgOptionString

void GetProgOptionString(char option_name[], char str[]);

Description:

Copies the current value of the **option_name** programming option to the **str** string. The option must be of type '**String**' - a text string. Example: "Copyright" option of the National Semiconductor COP87SER7 device.

9.3.2.2.14   Function mprintf

void mprintf(char format[], ...);

  Description:

The **mprintf** function is used just like **printf** but the message is displayed not in the **Console** window but in the "**Operation Progress**" window of the **Program Manager** window.

9.3.2.2.15   Function OpenProject

void OpenProject(char file_name[]);

Description:

Load the project with the name specified as the **file_name.** Call of this function is equivalent of loading the project via the menu **Project > Open.** Use of projects is very convenient, especially for mass production.

9.3.2.2.16   Function ProgOptionDefault

void ProgOptionDefault(char option_name[]);

Description:

Set the default value of the **option_name** programming option.

9.3.2.2.17   Function ReadShadowArea

void ReadShadowArea(int sub_level, unsigned long addr, unsigned int len, void data[], int site=0);

Description:

Read data from a specified shadow memory to the array "data". First, you have to create a shadow area through the menu **Configure** > **The Serialization, Checksum and Log File** dialog > **Custom shadow**

---

memory tab. The start address of the data to be written into the **addr** may differ from the start address of the custom shadow area but it is necessary the end address should not exceed the end address of the shadow area.

The **int site=0** means the socket# 0; i.e., the only socket for a single-site programmer or the first socket in a gang programmer. In the gang mode it is necessary to specify the socket number (the first one has the number 0).

9.3.2.2.18 Function SetProgOption

void SetProgOption(char option_name[], char option_string[]);

Description:

Set value for the programming option. The programming options are listed in the lower right corner of the **Device and Algorithm Parameters' Editor** window.

Parameters:

option_name - option name, e.g. "Vpp".

option_string - option value as character string. Options can be of several types (certain option type can be determined by hitting the "Edit" button in the **Device and Algorithm Parameters' Editor** window).

• floating point numbers, for example, programming voltage. For such options, the option_string parameter should represent a floating point number, for example, "12.3".

• integer numbers. The option_string parameter should represent an integer value, for example, "215".

• "menu" type options. In these cases, the option_string parameter should be a menu item string, for example, "Disabled". Menu can be observed by hitting the "Edit" button in the **Device and Algorithm Parameters' Editor** window).

• character strings, for example, "Copyright".

• check boxes option. Check boxes option is a list of options; each of them can be checked or unchecked. To specify a value for a check box option, append an '=' sign to the option name followed with 0 or 1. For example, to set up the CPD memory protection bit of PIC18F8720 chip, write

SetProgOption("Memory protection", "CPD=1");

Examples

SetProgOption("Vpp", "12.5");
SetProgOption("PWRT", "Disabled");

See also examples that come with the ChipProg package.

9.3.2.2.19 Function WriteShadow Area

void WriteShadowArea(int sub_level, unsigned long addr, unsigned int len, void data[], int site=0);

Description:

Write data from the array **data** to a specified shadow memory. First, you have to create a shadow area through the menu **Configure** > **The Serialization, Checksum and Log File** dialog > **Custom shadow memory** tab. The start address of the data, to be written into the **addr**, may differ from the start address of the custom shadow area but it is necessary that the end address should not exceed the end address of the shadow area.

The **int site=0** means the socket# 0; i.e., the only socket for a single-site programmer or the first socket in a gang programmer. In the gang mode it is necessary to specify the socket number (the first one has the number 0).

9.3.2.2.20 Variable BlankCheck

extern int BlankCheck;

The value of the "Blank check before program" option in the **Program Manager** window (tab **Options**). Assigning value to **BlankCheck** automatically changes the option in the window and vice versa.

9.3.2.2.21 Variable BufferStartAddr

extern unsigned long BufferStartAddr;

The value of the start address in the buffer used for operation. Assigning value to **BufferStartAddr** automatically changes the buffer start address field in the window and vice versa.

9.3.2.2.22 Variable Checksum

extern unsigned long Checksum;

A checksum of the data to be written into the device being currently programmed. This checksum can be specified by the script that defines an algorithm for the checksum computation. This parameter is usually set in the **Checksum** tab of the **Serialization, Checksum and Log File** dialog of the **Configure** menu.

9.3.2.2.23 Variable ChipEndAddr

extern unsigned long ChipEndAddr;

The value of the start address in the device used for operation. Assigning value to ChipEndAddr automatically changes the end address field in the window and vice versa.

9.3.2.2.24 Variable ChipStartAddr

extern unsigned long ChipStartAddr;

The value of the start address in the device used for operation. Assigning value to ChipStartAddr automatically changes the start address field in the window and vice versa.

9.3.2.2.25 Variable DeviceBatchSize

extern unsigned long DeviceBatchSize;

Number of devices in the lot to be programmed. This variable is used for counting down the devices from

the **DeviceBatchSize** value to zero. A check box for enabling the device count-down and other controls is accessible in the **Statistic** tab of the **Program Manager** window.

Example: if you need to program 10000 devices of the same type with the same data and then the programming should be stopped, the **DeviceBatchSize**=10000.

9.3.2.2.26   Variable DialogOnError

extern int DialogOnError;

If the value of this variable is set to nonzero (default), then if there is an error occurred during a programming function execution (see ExecFunction), the dialog with error description is displayed. Otherwise no dialog is displayed and ExecFunction() immediately returns with code EF_ERROR.

9.3.2.2.27   Variable GangMode

extern int GangMode;

The variable's value will be 1 if the ChipProgUSB software has been launched in the gang mode; for example, if it has been launched in the command line mode with the key **/GANG**, otherwise it will be 0. The **GangMode** variable is accessible for reading only.

9.3.2.2.28   Variable InsertTest

extern int InsertTest;

The value of the "Insert test" option in The Program Manager Window (tab Options). Assigning value to InsertTest automatically changes the option in the window and vice versa.

9.3.2.2.29   Variable LastErrorMessage[]

extern char LastErrorMessage[];

String that contains the last error message about operation on device. See also ExecFunction.

9.3.2.2.30   Variable NumSites

extern int NumSites;

The number of the gang programmer's operable sockets (for example, for a ChipProg-G41 device programmer, **NumSites** is four. The **NumSites** variable is accessible for reading only.

9.3.2.2.31   Variable ReverseBytesOrder

extern int ReverseBytesOrder;

The value of the "Reverse bytes order" option in The Program Manager Window (tab Options). Assigning value to ReverseBytesOrder automatically changes the option in the window and vice versa.

9.3.2.2.32   Variable SerialNumber

extern unsigned long SerialNumber;

The serial number of the device currently being programmed. This number can be specified by the script

that defines a start serial number and an algorithm for the serial number incrementation. These parameters are usually set in the **Serial Number** tab of the **Serialization, Checksum and Log File** dialog of the **Configure** menu.

9.3.2.2.33 Variable Signature

extern char Signature[];

A string of characters to be written in the device being currently programmed as a unique signature. This signature can be specified by the script. Usually it is set in the **Signature String** tab of the **Serialization, Checksum and Log File** dialog of the **Configure** menu.

9.3.2.2.34 Variable VerifyAfterProgram

extern int VerifyAfterProgram;

The value of the "Verify after program" option in The Program Manager Window (tab Options). Assigning value to VerifyAfterProgram automatically changes the option in the window and vice versa.

9.3.2.2.35 Variable VerifyAfterRead

extern int VerifyAfterRead;

The value of the "Verify after read" option in The Program Manager Window (tab Options). Assigning value to VerifyAfterRead automatically changes the option in the window and vice versa.

9.3.2.3 Mathematical functions

sin

asin

cos

acos

tan

tanh

atan

log

log10

sqrt

ceil

floor

exp

fabs

fmod

frexp

abs

pow

pow10

9.3.2.4   String operation functions

Functions for string operation receive arrays as parameters. Functions of the memxxxx type can use arrays of any type; other functions use the char arrays.

The script file language does not support pointers, that is why all string operation functions include the index, desr_index, and scr_index parameters to specify the initial shift in the array. The default value of these parameters is 0. These parameters are not considered in the following line function descriptions.

Note once again that arrays are transferred "by pointer", that is, the array itself is transferred and not its copy.

memccpy

memcpy

memmove

movmem

memchr

memset

setmem

memcmp

memicmp

stpcpy

strcat

strchr

strcmp

stricmp

strcmpi

[****]

strcpy

[****]

strlwr

[****]

strncat

strncmp

strncmpi

strnicmp

strncpy

strnset

strpbrk

strspn

[****]

strrchr

strrev

[****]

9.3.2.5 Character operation functions

isalnum

isalpha

isascii

iscntrl

isdigit

isgraph

islower

isprint

ispunct

isspace

isupper

isxdigit

toascii

tolower

toupper

9.3.2.6 Functions for file and directory operation

chdir

getcurdir

findfirst

findnext

 ff_attrib

 ff_time

 ff_date

 ff_size

 ff_name

fnsplit

fnmerge

 fullpath

getcwd

getdisk()

setdisk

mkdir

rmdir

searchpath

getdfree

unlink

chsize

close

creat

creatnew

creattemp

dup

dup2

eof

filelength

getftime

setftime

isatty

lock

unlock

locking

lseek

open

read

write

rename

setmode

[****]

9.3.2.7 Stream file functions

clearerr

fclose

fdopen

feof

ferror

fflush
fgetc

fgets

fileno

fopen

fprintf

fputc

fputs

fread

freopen

fscanf

fseek

ftell

fwrite

getc

getw

putc

[****]

rewind

9.3.2.8   Formatted input-output functions

Formatted input-output functions perform data conversion in accordance with the format line. You can find description of the format line in any book on the C language.

Note that the arguments for input functions should be arrays, and not simple variables. This is because the pointers are not supported in the script file language, and it is impossible to transfer an address with the simple variable.

Attention! Your arguments passed to the formatted input-output functions shall match the format line. Otherwise, the CPI2-B1 program may fail.

fprintf

fscanf

scanf

pscanf

sscanf

printf

_printf

sprintf

MessageBox

MessageBoxEx

9.3.2.9   Script File Manipulation Functions

ExecScript

GetScriptFileName

TerminateScript

TerminateAllScripts

exit

9.3.2.10 Text editor functions

The text editor functions manipulate with text in the [Source](#) You can start the script files with the custom hot keys (for more about this, see).

All text editor functions assume that the text editor window is active, when function is called, so they do not receive the window handle as a parameter unlike other functions that manipulate windows in CPI2-B1.

The CPI2-B1 package includes several examples of script files performing useful commands. The sources are located in the KEYCMD sub-directory.

Note that line and column numbers begin from 1.

[GotoXY](#)

[Up](#)

[Down](#)

[Left](#)

[Right](#)

[Tof](#)

[Eof](#)

[Eol](#)

[BackSpace](#)

[Cr](#)

[DelLine](#)

[DelChar](#)

[CurChar](#)

[GetLine](#)

[ForwardTill](#)

[ForwardTillNot](#)

[GetWord](#)

[WordLeft](#)

[WordRight](#)

[FirstWord](#)

[SetMark](#)

[GetMark](#)

[Text](#)

[BlockBegin](#)

BlockEnd

BlockOff

BlockCopy

BlockFastCopy

BlockDelete

BlockMove

BlockPaste

Search

SearchReplace

SetFileName

GetFileName

SaveFile

[****]

OpenEditorWindow

Text editor built-in variables

InsertMode

CaseSensitive

WholeWords

[****]

BlockCol1

BlockCol2

BlockLine1

BlockLine2

BlockStatus

CurLine

CurCol

LastFoundString

9.3.2.11 Debug shell control functions

These functions control CPI2-B1.

RedrawScreen

9.3.2.12 Windows operation functions and other system functions

Attention! Only the experienced programmers should use the Windows operation functions. These functions provide advanced capabilities, but when used incorrectly, they may hang the operating system.

API

LoadLibrary

FreeLibrary

WaitSendMessage

WaitGetMessage

inport

inportb

outport

outportb

peek

peekb

poke

pokeb

exec

getenv

putenv

9.3.2.13 Graphical output functions

Graphical output functions draw various graphical objects and text in special User window. To draw in a user window, first open it with the OpenUserWindow function that returns the window identifier (handle). Then use the identifier to reference the window (multiple user windows can be open at the same time). For more information, see User window.

In all graphical output functions, the first parameter () is the window identifier.

OpenUserWindow

ClearWindow

SetCaption

SetToolbar

SetUpdateMode

UpdateWindow

SelectPen

SelectBrush

SelectFont

SetTextColor

SetBkColor

SetBkMode

DisplayText

DisplayTextF

MoveTo

LineTo

FillRect

Rectangle

[****]

InvertRect

Curcuit

Ellipse

Polyline

SetPixel

AddButton

RemoveButtons

WaitWindowEvent

[****]

LastEventInt{1...4}

9.3.2.14 I/O Stream window operation functions

Stream window control functions allow you to display text in the special I/O Stream window.

In all Stream window control functions, the first parameter (handle) is the window identifier.

OpenStreamWindow

SetTextColor

wprintf

wgetchar

LastChar

wgethex

wgetstring

LastString

9.3.2.15 Event Wait Functions

These extremely useful functions serve to simulate external environment. Also, you can use them in simulators to develop various tests.

Wait

WaitMemoryAccess

WaitExprTrue

WaitExprChange

WaitStop

WaitWindowEvent

9.3.2.16 Other Various Functions

delay
gettime
getdate
difftime
atof
atoi
itoa
ltoa
ultoa
rand
random
randomize
srand
strtol
strtoul

9.3.3   Built-in Variables by Group

You can access script language built-in variables in the same way as regular global variables. However, some built-in variables are accessible only for reading, and in case of attempt to write to such variable.

The built-in variables are declared in the **system.h** header file.

Programming variables:
InsertTest
ReverseBytesOrder
BlankCheck
VerifyAfterProgram
VerifyAfterRead
ChipStartAddr
ChipEndAddr
BufferStartAddr
LastErrorMessage
DialogOnError

Text editor built-in variables:
InsertMode
CaseSensitive
WholeWords
RegularExpressions
BlockCol1
BlockCol2
BlockLine1
BlockLine2
BlockStatus
CurLine
CurCol
LastFoundString

Miscellaneous variables:
WorkFieldWidth
WorkFieldHeight

9.3.4 List of Built-in Functions and Variables

Below is the alphabetical list of all built-in functions and variables of scripting language.

asin
atan
atof
atoi
ceil
chdir
chsize
clearerr
close
cos
creat
creatnew
creattemp
delay
difftime
dup
dup2
eof
errno
exec
exit
exp
fabs
fclose
fdopen
feof
ferror
fflush
fgetc
fgets
filelength
fileno
findfirst
findnext
floor
fmod
fnmerge
fnsplit
fopen
fprintf
fputc
fputs
fread
freopen
frexp
fscanf
fseek

ftell
fwrite
getc
getcurdir
getcwd
getdate
getdfree
getdisk()
getenv
getftime
gettime
getw
inport
inportb
isalnum
isalpha
isascii
isatty
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
itoa
lock
locking
log
log10
lseek
ltoa
memccpy
memchr
memcmp
memcpy
memicmp
memmove
memset
mkdir
movmem
mprintf
open
outport
outportb

peek
peekb
poke
pokeb
pow
pow10
printf
pscanf
putc
putenv
putw
rand
random
randomize
read
rename
rewind
rmdir
scanf
searchpath
setdisk
setftime
setmem
setmode
sin
sprintf
sqrt
srand
sscanf
stpcpy
strcat
strchr
strcmp
strcmpi
strcpy
strcspn
stricmp
strlen
strlwr
strncat
strncmp
strncmpi
strncpy
strnicmp
strnset
strpbrk
strrchr

strrev
strset
strspn
strstr
strtol
strtoul
strupr
tan
tanh
tell
toascii
tolower
toupper
ultoa
unlink
unlock
wgetchar
wgethex
wgetstring
wprintf
write

9.3.5 Scripting Functions

Enter topic text here.

9.3.5.1 fnmerge

Declaration:

void fnmerge(char path[], char drive[], char dir[], char name[], char ext[]);

Builds a path from component parts.

fnmerge makes the path name from its components. The new path name is

X:\DIR\SUBDIR\NAME.EXT

where:

drive =   X

dir   =   \DIR\SUBDIR\

name =   NAME

ext =   .EXT

fnmerge assumes there is enough space in path for the constructed path name. The maximum constructed length, MAXPATH, is defined in system.h.

fnmerge and fnsplit are invertible: if you split the given path with fnsplit, then merge the resultant components with fnmerge and you end up with this path.

9.3.5.2 Function _ff_attrib

Declaration:

`char _ff_attrib(char ffblk[]);`

Description

Returns the attribute byte of the file found upon the function findfirst or findnext access. The ffblk parameter is the buffer filled with information on the file after findfirst or findnext access.

Example

See function findfirst

9.3.5.3 Function _ff_date

Declaration:

`int _ff_date(char ffblk[]);`

Description

Returns the word with the file (creation or modification) date for the file found upon the function findfirst or findnext access. The ffblk parameter is the buffer filled with information on the file after the findfirst or findnext access.

Example

See function findfirst

9.3.5.4 Function _ff_name

Declaration:

`void _ff_name(char ffblk[], char fname[]);`

Description

Copies the name of the file found upon the function findfirst or findnext access to the fmane array. The ffblk parameter is the buffer filled with information on the file after the findfirst or findnext access. The file name does not contain the disk name or path.

Example

See function findfirst

9.3.5.5 Function _ff_size

Declaration:

`long _ff_size(char ffblk[]);`

Description

Returns the size of the file found upon the function findfirst or findnext access. The ffblk parameter is the buffer filled with information on the file after the findfirst or findnext access.

Example

See function findfirst

9.3.5.6   Function _ff_time

Declaration:

`int _ff_time(char ffblk[]);`

Description

Returns the word with the file creation (or modification) time for the file found upon the function findfirst or findnext access. The ffblk parameter is the buffer filled with information on the file after the findfirst or findnext access.

Example

See function findfirst.

9.3.5.7   Function _fullpath

Declaration:

int _fullpath(char buf[], char path[]);

Description

Converts a relative path name to the absolute one.

_fullpath converts the relative path name in a path to the absolute path name that is stored in the array of characters pointed to by buf. The function returns FALSE the path contains an invalid drive letter.

Returned value

If successful, the _fullpath function will return TRUE. On error, it returns FALSE.

9.3.5.8   Function _GetWord

Declaration:

`void _GetWord(char dest[]);`

Description

Copies the word under the cursor to the **dest** array. If there is no word under the cursor, then the first element of **dest** will be 0.

9.3.5.9   Function _printfv

Declaration:

`void _printf(char format[], ... );`

Description

Acts like printf, but does not append the newline character to the line.

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the <% CM%> program may crash, because it cannot check the correspondence between the format string and parameters passed.

9.3.5.10 Function abs

Declaration:

`long abs(long x);`

Description

The abs function calculates the absolute value of the integer argument val.

Returned value

The abs function returns the absolute value of the integer argument val.

9.3.5.11 Function acos

Declaration:

`float acos(float x);`

Description

The acos function calculates the arc cosine of the floating-point number x. Argument x should range from -1 to 1, otherwise the result will be equal to 0 (for x > 1) or to PI (for x < -1). The function returns value in the range from 0 to PI.

Returned value

The acos function returns the arc cosine of argument x.

9.3.5.12 Function ActivateWindow

Declaration:

`void ActivateWindow(unsigned long handle);`

Description

Activates the specified window. The window becomes 'active' and is placed over all other windows of <% CM%>.

9.3.5.13 Function AddButton

Declaration:

`int AddButton(unsigned long handle, char button_text[], int x, int y, int width, int height);`

Description

Adds a button to the window. The button is a usual button of the standard Windows dialog boxes. When you click the button, the event is generated that can be captured with the WaitWindowEvent function, and the corresponding operation is carried out.

If the specified button already exists in the window (already added by AddButton with the same parameters), the new button will not be added and the existing button will be used.

Parameters:

button_text - the text written on the button

```
        x, y          - the coordinates of the upper left corner within the window
        width         - the button width
        height        - the button height
```

Returned value

The button identifier. It is used by the WaitWindowEvent function to determine, which button was clicked (there multiple buttons in the window).

Example

```
        AddButton(handle, "Start", 50, 50, 70, 24);
```

9.3.5.14 Function AddrExpr

Declaration:

```
unsigned long AddrExpr(char str[]);
```

Description

Calculates the expression and returns the result (the **str** parameter) as an address in microcontroller memory.

Example

```
        int addr_port0 = AddrExpr("PORT0");
        WaitMemoryAccess(addr_port0, AS_DATA, 1, MA_WRITE);
```

Note that 'AddrExpr("PORT0")' is the same as 'Expr("&PORT0")'.

Also, see Expr, FloatExpr, Operations and Expressions.

9.3.5.15 Function AddWatch

Declaration:

```
void AddWatch(char name[], int format=DF_HEX);
```

Description

Adds the specified name (the name parameter) to the **Watches** window in the specified format. If the **Watches** window is not already opened, it will be opened automatically.

Examples

```
        AddWatch("Duration", DF_DEC);
        AddWatch("Address");   // the default format is hexadecimal
```

9.3.5.16 Function API

Declaration:

```
unsigned long API(char func_name[], ... );
```

Description

Calls a 16-bit Windows API function with the name specified in func_name and transfers the parameters

specified in API to this function.

Make sure you use the correct parameter number and size, because <%CM%> knows nothing about them. When necessary, use the explicit type conversions and put character 'L' in the end of long-type constants.

To reduce problems, when an array is transferred as the parameter, a long (32-byte) pointer is transferred.

Returned value

What was returned by the called API function is in registers DX:AX. If it is a pointer, then data can be accessed using the peek, poke, peekb, or pokeb functions.

Example

```
int ScreenHeight = API("GetSystemMetrics", SM_CYFULLSCREEN);
```

9.3.5.17 Function asin

Declaration:

```
float asin(float x);
```

Description

The asin function calculates the arc sine of the floating-point number x. The argument x should range from -1 to 1, otherwise the result will be equal to PI/2 (for x > 1) or to -PI/2 (for x < -1). The function returns value in the range from -PI/2 to PI/2.

Returned value

The asin function returns the arc sine of argument x.

9.3.5.18 Function atan

Declaration:

```
float atan(float x);
```

Description

The atan function calculates the arc tangent of the floating-point number x. The function returns value in the range from -PI/2 to PI/2.

Returned value

The atan function returns the arc tangent of argument x.

9.3.5.19 Function atof

Declaration:

```
float atof(char s[]);
```

Description

Converts an ASCII-string (parameter s) into the floating-point number.

9.3.5.20 Function atoi

Declaration:

```
int atoi(char s[]);
```

Description

Converts an ASCII-string (parameter s) into the integer number.

9.3.5.21 Function BackSpace

Declaration:

```
void BackSpace();
```

Description

Works like the **BackSpace** key.

9.3.5.22 Function BlockBegin

Declaration:

```
void BlockBegin(int block_type);
```

Description

Begins marking of block (see Block Operations). The **block_type** parameter indicates the type of block. For convenience, the **system.h** header file defines constants for the block functions:

```
EB_NONE    - no block (not used in this function)
EB_LINE    - line block
EB_VERT    - vertical block
EB_STREAM  - stream block
```

9.3.5.23 Function BlockCopy

Declaration:

```
void BlockCopy();
```

Description

Copies the block to the clipboard.

9.3.5.24 Function BlockDelete

Declaration:

```
void BlockDelete();
```

Description

Deletes the block. The block is copied to the clipboard

9.3.5.25 Function BlockEnd

Declaration:

```
void BlockEnd();
```

Description

Finishes marking of block. It is supposed that before calling BlockEnd(), the BlockBegin function is called

and then the cursor is moved to the end of the block.

9.3.5.26 Function BlockFastCopy

Declaration:

`void BlockFastCopy();`

Description

Copies the block from the cursor position.

9.3.5.27 Function BlockMove

Declaration:

`void BlockMove();`

Description

Moves the block to the cursor position.

9.3.5.28 Function BlockOff

Declaration:

`void BlockOff();`

Description

Turns the block off..

9.3.5.29 Function BlockPaste

Declaration:

`void BlockPaste();`

Description

Pastes the block from the clipboard to the cursor position

9.3.5.30 Function CallLibraryFunction

Declaration:

`unsigned long CallLibraryFunction(unsigned long inst, char func_name[], ... );`

Description

Calls the func_name function from DLL and its HINSTANCE is transferred to inst. Otherwise, this function is similar to the function API call.

Example

```
unsigned long instance = LoadLibrary("EXTEND.DLL");
long result = CallLibraryFunction(instance, "Initialize", 0, 1L);
```

9.3.5.31 Function ceil

Declaration:

```
float ceil(float x);
```

Description

The ceil function calculates the least integer value that is greater than or equal to x.

Returned value

The ceil function returns the double-type number equal to the least integer that is no greater than x.

9.3.5.32 Function chdir

Declaration:

```
int chdir(char path[]);
```

Description

Sets up the new default directory specified in parameter path. The latter might also contain a disk name, but the disk does not change: only the default directory changes on this disk.

Returned value

If the directory change is successful, 0 will be returned, and -1 otherwise.

9.3.5.33 Function CheckSum

Declaration:

```
unsigned long CheckSum(unsigned long start_addr, unsigned long end_addr, int
addr_space);
```

Description

Calculates the checksum for data in the **addr_space** memory that starts from start_addr and ends at end_addr. The checksum is calculated by simple addition of byte values.

Returned value

The 32-bit checksum.

Example

```
printf("%08lX", CheckSum(0, 0x1FFF, AS_DATA));
```

9.3.5.34 Function chsize

Declaration:

int chsize(long handle, long size);

Description

Changes the file size.

chsize changes the size of the file associated with handle. It can truncate or extend the file, depending on the value of size compared to the file's original size.

The mode, in which you open the file, must allow writing.

If chsize extends the file, it will append the null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator will be lost.

Returned value

On success, chsize returns 0. On failure, it returns -1 and sets the errno global variable to one of the following values:

```
EACCESS        Permission denied
EBADF          Bad file number
ENOSPC         No space left o
```

9.3.5.35 Function ClearAllBreaks

Declaration:

```
void ClearAllBreaks();
```

Description

Clears all breakpoints of all types.

9.3.5.36 Function ClearBreak

Declaration:

```
void ClearBreak(unsigned long addr);
```

Description

Clears the code breakpoint at the specified address.

9.3.5.37 Function ClearBreaksRange

Declaration:

```
void ClearBreaksRange(unsigned long start_addr, unsigned long end_addr);
```

Description

Clears the code breakpoints in the range from start_addr to end_addr inclusive.

9.3.5.38 Function clearerr

Declaration:

void clearerr(unsigned long stream);

Description

Resets error indication.

clearerr resets the specified stream's error and end-of-file indicators to 0. Once the error indicator is set up, the stream operations continue to return the error status until a call is made to clearerr or rewind. The end-of-file indicator is reset with each input operation.

9.3.5.39 Function ClearWindow

Declaration:

```
void ClearWindow(unsigned long handle);
```

Description

Clears the specified window, which can be a **User** window or an **I/O Stream** window.

9.3.5.40 Function close

Declaration:

int close(long handle);

Description

Closes a file.

The close function closes the file associated with handle (the file handle obtained from a call to creat, creatnew, creattemp, dup, dup2,).

It does not write the Ctrl-Z character to the end of the file. If you want to terminate the file with Ctrl-Z, you must explicitly output it.

Returned value

Upon successful completion, close returns 0. On error (if it fails because handle is not the handle of a valid, open file), close returns -1 and the errno global variable is set to

EBADF   Bad file number

9.3.5.41 Function CloseProject

Function CloseProject

Declaration:

```
void CloseProject();
```

Description

Closes the project. If no project is loaded, nothing will happen.

Calling this function is useful, if you want to prepare the shell for loading a program without a project.

9.3.5.42 Function CloseWindow

Declaration:

```
void CloseWindow(unsigned long handle);
```

Description

Closes the specified window. The handle parameter is the window identifier produced by the calls of the OpenWindow, and FindWindow functions.

9.3.5.43 Function cos

Declaration:

```
float cos(float x);
```

Description

The cos function calculates the cosine of the floating-point number x.

Returned value

The cos function returns the cosine of argument **0x**.

9.3.5.44 Function Cr

Declaration:

void Cr();

Description

Works like the **Enter** key.

9.3.5.45 Function creat

Declaration:

int creat(char path[], int amode);

Description

Creates a new file or overwrites an existing one.

Note. Remember that the backslash in a path requires '\\'.

creat creates a new file or prepares to rewrite an existing file given by path. amode applies only to newly created files. A file created with creat is always created in the translation mode specified by the _fmode global variable (O_TEXT or O_BINARY). If the file exists and the write attribute is set, then creat will truncate the file to the length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, then the creat call will fail and the file will remain unchanged. The creat call examines only the S_IWRITE bit of the access-mode word amode. If this bit is 1, then the file can be written to. If the bit is 0, then the file is marked as read-only. All other operating system attributes are set to 0. amode can be one of the following (defined in system.h):

```
Value of amode      Access permission
   S_IWRITE               Permission to write
   S_IREAD                Permission to read
   S_IREAD | S_IWRITE     Permission to read and write (write permission
                          implies read permission))
```

Returned value

Upon successful completion, creat returns the new file handle (a nonnegative integer); otherwise, it returns -1. In the event of error, the errno global variable is set to one of the following:

```
   EACCES        Permission denied
   ENOENT        Path or file name not found
   EMFILE        Too many open files
```

9.3.5.46 Function creatnew

Declaration:

int creatnew(char path[], int amode);

Description

Creates a new file.

creatnew is identical to creat with the only exception: if the file exists, then creatnew will return error and leave the file untouched. The amode

FA_HIDDEN   Hidden file

FA_RDONLY   Read-only attribute

FA_SYSTEM   System file

Returned value

Upon successful completion, creatnew returns the handle of new file (a non-negative integer); otherwise, it returns -1. In the event of error, the errno global variable is set to one of the following values:

EACCES   Permission denied

EEXIST   File already exists

EMFILE   Too many open files

ENOENT   Path or file name not found

9.3.5.47 Function creattemp

Declaration:

int creattemp(char path[], int attrib);

Description

Creates a unique file in the directory associated with the path name. A file created with creattemp is always created in the translation mode specified by the _fmode global variable (O_TEXT or O_BINARY).

path is the path name ending with backslash (\). The unique file name is selected in the directory given by path. The newly created file name is stored in the path string supplied. path should be long enough to hold the resulting file name. The file is not automatically deleted, when the program terminates.

creattemp accepts attrib, the DOS attribute word. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

The attrib argument to creattemp can be either zero or an OR-combination of any of the following constants (defined in system.h):

```
    FA_HIDDEN      Hidden file
    FA_RDONLY      Read-only attribute
    FA_SYSTEM      System file
```

Returned value

Upon successful completion, the new file handle (a non-negative integer) is returned; otherwise, -1 is returned. In the event of error, the errno global variable is set to one of the following values:

```
EACCES        Permission denied
EMFILE        Too many open files
ENOENT        Path or file name not found
```

9.3.5.48 Function CurChar

Declaration:

```
char CurChar();
```

Description

Returns the character under the cursor. If the cursor is beyond the line end, then CurChar() will return 0.

9.3.5.49 Function Curcuit

Declaration:

```
void Curcuit(unsigned long handle, int x1, int y1, int x2, int y2);
```

Description

Draws an unpainted ellipse using the pen selected with the SelectPen function; (x1, y1) are the coordinates of the upper left corner of the rectangle, in which the ellipse will be drawn, (x2, y2) are the coordinates of its lower right corner.

9.3.5.50 Function delay

Declaration:

```
void delay(unsigned int milliseconds);
```

Description

Suspends the program for the specified time interval.

Example

```
while (1)
{
  Step();         // to execute a step
  RedrawScreen(); // To update the screen. Step() does not do it.
  delay(1000);    // wait for one second. During this time step
}               // results can be observed
```

9.3.5.51 Function DelChar

Declaration:

```
void DelChar(int count=1);
```

Description

Deletes **count** characters beginning from the cursor position

9.3.5.52 Function DelLine

Declaration:

```
void DelLine(int count=1);
```

Description

Deletes the current line.

9.3.5.53 Function difftime

Declaration:

```
unsigned long difftime(int time1[], int time2[]);
```

Description

Obtains the time difference between the two counts transferred in the time1 and time2 arrays. The counts should be obtained with the gettime function; time1 is the earlier count.

Because the gettime function uses the system timer, computation error for the interval can be as long as 104 milliseconds.

Returned value

The time difference between two counts in milliseconds.

Example

```
int time1[4];
int time2[4];
gettime(time1);
while (1)
{
   gettime(time2);
   printf("Difference: %lu", difftime(time1, time2));
}
```

9.3.5.54 Function DisplayText

Declaration:

```
void DisplayText(unsigned long handle, char text[], int x, int y);
```

Description

Displays text in the window using a monospaced font and text coordinates, that is, **x** is the column number, and **y** is the line number.

To display text with any font and in any place, use the DisplayTextF function.

9.3.5.55 Function DisplayTextF

Declaration:

```
void DisplayTextF(unsigned long handle, char text[], int x, int y);
```

Description

Displays text in the window using a proportional font (see the SelectFont function) and graphical coordinates (in pixels).

9.3.5.56 Function Down

Declaration:

```
void Down(int count=1);
```

Description

Move the cursor **count** lines down. The same result can be achieved by incrementing the CurLine built-in variable.

9.3.5.57 Function dup

Declaration:

int dup(long handle);

Description

Duplicates a file handle.

dup creates a new file handle that has the following common features with the original file handle:

Same open file or device
Same file pointer (that is, changing the file pointer of one changes the other)
Same access mode (read, write, read/write))

handlecreatopen, dup, or dup2.

Returned value

Upon successful completion, dup returns the new file handle, a nonnegative integer; otherwise, dup returns -1. In the event of error, the errno global variable is set to one of the following values:

```
EBADF        Bad file number
EMFILE       Too many open files
```

9.3.5.58 Function dup2

Declaration:

int dup2(long oldhandle, long newhandle);

Description

Duplicates a file handle (oldhandle) onto an existing file handle (newhandle).

dup2 creates a new file handle that has the following common features with the original file handle:

Same open file or device

Same file pointer (that is, changing the file pointer of one changes the other)

Same access mode (read, write, read/write)

dup2 creates a new handle with the value of newhandle. If the file associated with newhandle is open, when dup2 is called, then the file will be closed.

newhandle and oldhandle are the file handles obtained from the creat, open, dup, or dup2 call.

Returned value

dup2 returns 0 on successful completion, and -1 otherwise. In the event of error, the errno global variable is set to one of the following values:

EBADF   Bad file number

EMFILE   Too many open files

9.3.5.59 Function Ellipse

Declaration:

```
void Ellipse(unsigned long handle, int x1, int y1, int x2, int y2);
```

Description

Draws an ellipse using the pen selected with the SelectPen function and paints it with the brush selected by the SelectBrush function; (x1, y1) are the coordinates of the upper left corner of the rectangle, in which the ellipse will be drawn; (x2, y2) are the coordinates of its lower right corner.

9.3.5.60 Function eof

Declaration:

```
int eof(long handle);
```

Description

Checks for end-of-file.

eof determines whether the file associated with **handle** has reached the end-of-file.

Returned Value

If the current position is the end-of-file, then eof will return 1; otherwise, it will return 0. The return value of -1 indicates an error; the errno global variable is set to

EBADF   Bad file number

9.3.5.61 Function Eof

Declaration:

```
void Eof();
```

Description

Move the cursor to the file end.

9.3.5.62 Function Eol

Declaration:

`void Eol();`

Description

Move the cursor to the end of the current line.

9.3.5.63 Function exec

Declaration:

`int exec(char program[], char params[], char work_dir[], int show=SW_SHOW);`

Description

Starts a Windows application or DOS.

Parameters:

```
program  - the name of the file under execution
params   - the command line parameters
work_dir - the working directory for the application to be started
show     - the constant to define the application window display mode.
           Constants with the SW_ prefix are given in system.h.
```

Note that the script file will not wait for the started application to stop operation, if special measures are not taken.

Returned value

What was returned by the function API ShellExecute, that is, HINSTANCE of the application or error message.

Example

```
exec("pifedit.exe", "command.pif");
```

9.3.5.64 Function ExecMenu

Declaration:

`int ExecMenu(char title[], char items[], int start_sel=0);`

Description

Displays the dialog menu on the screen.

Parameters:

```
title     - the dialog box title;
items     - the line describing the menu items. Every item ends with the zero
            byte; the last item ends with two zero bytes.
start_sel - the number of the menu item that will be selected by default,
            when the window opens.
```

Returned value

The number of the menu item selected by the user or -1, if the **Cancel** button or Esc key is pressed. The selected menu line is copied to the SelectedString[] built-in variable. If the user cancels the selection, then the null string will be copied to the Selected String.

Example

```
int choice =
  ExecMenu("Choose program to load",   // the title
           " Load Example #1 \0"
           " Load Example #2 \0"
           " Load Example #3 \0"        // the items
           "\0");                       // the second zero at the end
switch (choice)
{
  case 0:  LoadProgram("EXAMPLE1.OMF", LF_UBROF); break;
  case 1:  LoadProgram("EXAMPLE2.OMF", LF_UBROF); break;
  case 2:  LoadProgram("EXAMPLE3.OMF", LF_UBROF); break;
  default: printf("No example will be loaded");
}
```

9.3.5.65 Function ExecScript

Declaration:

```
void ExecScript(char file_name[], char include_dir[]="", char defines[]="", int
debug=0);
```

Description

The ExecScript function starts the script file, whose name is indicated in the file_name parameter.

Parameters:

file_name[]	The name of the script file to be started. It can contain a partial or full path. If extension is not specified, the CMD extension will be automatically substituted. If the file is not found, the <%CM%> system directory will be automatically scanned.
include_dir[]	The listing of directories, where the compiler will search for the #include-files. You can specify multiple directory names separated by semicolon.
char defines[]	The string with the definitions of preprocessor variables. Also, see the **Script Files** dialog.
debug	If not equal to 0, then the **Script Source window** will be opened for the loaded script file. After loading the script file, switches to the debug mode.

Note that only the first parameter is required, other parameters have the default values.

If the specified script file is already under executing, then another script file cannot be loaded.

Also, see Inclusion of Files (#include).

9.3.5.66 Function exit

Declaration:

```
void exit();
```

Description

Stops execution of the script file that called this function. The file is unloaded from the memory, if

possible.

9.3.5.67 Function ExitProgram

Declaration:

```
void ExitProgram();
```

Description

Exits the work session of <%CM%> in the same way as by closing its window.

9.3.5.68 Function exp

Declaration:

```
float exp(float x);
```

Description

The exp function raises number e to the power x. The argument shall range from -88.72280 to 88.72280.

Returned value

The exp function returns the value of e raised to the power x.

9.3.5.69 Function Expr

Declaration:

unsigned long Expr(char str[]);

Description

Calculates the expression and returns the result as a 32-bit integer. The expression string is passed in the str parameter.

Example

printf("Result=%08lX", Expr("array[i] -> StartValue"));

Also, see AddrExpr, FloatExpr, Expressions.

9.3.5.70 Function fabs

Declaration:

```
float fabs(float x);
```

Description

The fabs function determines the absolute value of the floating-point number val.

Returned function

The fabs function returns the absolute value of val.

9.3.5.71 Function fclose

Declaration:

int fclose(unsigned long stream);

Description

Closes a stream.

fclose closes the specified stream. All buffers associated with the stream are flushed before closing. The system-allocated buffers are freed upon closing.

Returned value

fclose returns 0 on success. It will return EOF, if any errors are detected.

9.3.5.72 Function fdopen

Declaration:

unsigned long fdopen(long handle, char type[]);

Description

Associates a stream with a file handle.

obtained from creatdup, dup2, or open. The type of stream must match the mode of the opened handle. The type string used in a call to fdopen is one of the following values:

Value   Description

r Open for reading only.

Create for writing.

a Append; open for writing at the end-of-file or create for writing, if the file does not exist.

r+ Open an existing file for update (reading and writing).

w+ Create a new file for update.

a+

To specify that the given file is being opened or created in the text mode, append t to the value of the type string (for example, rt or w+t).

Similarly, to specify the binary mode, append brb or w+b). If t or b is not given in the type string, the mode is controlled by the _fmode global variable. If _fmode is set to O_BINARY, then files will be opened in the binary mode. If _fmode is set to O_TEXT, then files will be opened in the text mode.

Note. The O_* constants are defined in file system.h.

* output cannot be directly followed by input without intervening fseekor rewind;
* input cannot be directly followed by output without intervening fseek, rewind, or an input that encounters the end-of-file.

Returned value

On successful completion, fdopen returns the unsigned long identifying the stream. In the event of error, it returns 0.

9.3.5.73 Function feof

Declaration:

int feof(unsigned long stream);

Description

Detects the end-of-file on a stream.

feof tests the given stream for the end-of-file indicator. Once the indicator is set, the read operations on the file return the indicator until rewind is called or the file is closed. The end-of-file indicator is reset with each input operation.

Returned value

feof will return nonzero, if the end-of-file indicator is detected on the last input operation on the specified stream, and 0, if the end-of-file has not been reached.

9.3.5.74 Function ferror

Declaration:

int ferror(unsigned long stream);

Description

Detects errors on stream.

ferror tests the given stream for a read or write error. If the stream's error indicator is set, it will remain set until clearerr or rewind is called or until the stream is closed.

Returned value

ferror will return nonzero, if an error is detected on the specified stream.

9.3.5.75 Function fflush

Declaration:

int fflush(unsigned long stream);

Description

Flushes a stream.

If the given stream has buffered output fflush writes the output for stream to the associated file. The stream remains opened after fflush is executed. fflush produces no effect on the unbuffered stream.

Returned Value

fflush returns 0 on success. It will return EOF, if any errors are detected.

9.3.5.76 Function fgetc

Declaration:

int fgetc(unsigned long stream);

Description

Gets character from stream.

fgetc returns the next character on the specified input stream.

Returned Value

On success fgetc returns the character read after converting it to an int without the sign extension. On the end-of-file or error, it returns EOF.

9.3.5.77 Function fgets

Declaration:

```
int fgets(char dest[], int n, unsigned long stream);
```

Description

Gets a string from a stream.

fgets reads characters from **stream** into the **dest** string. The function stops reading, when it reads either n-1 characters or the newline character, which event comes first. fgets retains the newline character at the end of **dest**. The null byte is appended to s to mark the end of the string.

Returned Value

TRUE is returned on success; and FALSE on the end-of-file or error.

9.3.5.78 Function FileChanged

Declaration:

int FileChanged();

Description

If the file is changed since the last save, it will return 1; 0 otherwise.

9.3.5.79 Function filelength

Declaration:

long filelength(long handle);

Description

Gets file size in bytes.

filelength returns the length (in bytes) of the file associated with handle.

Returned Value

On success, filelength returns the long value of the file length in bytes. On error, it returns -1 and the errno global variable is set to

EBADF   Bad file number

9.3.5.80 Function fileno

Declaration:

```
int fileno(unsigned long stream);
```

Description

Gets file handle.

fileno returns the file handle for the given **stream**. If stream has more than one handle, then fileno will return the handle assigned to the stream, when it was first opened.

Returned Value

fileno returns the integer file handle associated with the stream.

9.3.5.81 Function FillRect

Declaration:

```
void FillRect(unsigned long handle, int x1, int y1, int x2, int y2);
```

Description

Draws a painted rectangle using the brush selected with the SelectBrush function; (x1, y1) are the coordinates of the upper left corner; (x2, y2) are the coordinates of the lower right corner.

9.3.5.82 Function findfirst

Declaration:

```
int findfirst(char path[], char ffblk[], int attrib);
```

Description

Starts search for files with the attributes specified in parameter attrib by the mask specified in path. The search can be continued with the findnext function.

The ffblk parameter specifies an internal data storage buffer for the function. Its size should be 48 bytes.

After findfirst access, the ffblk buffer contains information about the found file. The _ff_attrib, _ff_time, _ff_date, _ff_size and _ff_name functions receive ffblk as the parameter and return information on the file.

Returned value

If the specified file is found, it will return 0, and -1 otherwise.

Example

```
char ffblk[48];
int done = findfirst("c:\\data.*", ffblk, 0);
long total_size = 0;
while (@!done)
{
  total_size += _ff_size(ffblk);
  done = findnext(ffblk);
}
printf("Total size of the files @%lu", total_size);
```

9.3.5.83 Function findnext

Declaration:

```
int findnext(char ffblk[]);
```

Description

Continues the search for files started by the findfirst function.

Parameter ffblk is the buffer filled upon the findfirst access.

After the findnext access, the ffblk buffer contains information on the found file. The _ff_attrib, _ff_time, _ff_date, _ff_size and _ff_name functions receive ffblk as the parameter and return information on the file.

Returned value

If the specified file is found, it will return 0, and -1 otherwise.

Example

See function findfirst.

9.3.5.84 Function FindWindow

Declaration:

```
unsigned long FindWindow(int type);
```

Description

Finds the window of specified type (disassembler, dump, etc.) among the opened windows.

Constants describing window types are declared in the **system.h** header file (see description of the OpenWindow function).

If the window of specified type is opened but minimized, it will not be found.

Returned

The identifier of the opened window, if the latter is found; otherwise it returns 0.

9.3.5.85 Function FirstWord

Declaration:

```
void FirstWord();
```

Description

Moves the cursor to the first non-empty character in the line.

9.3.5.86 Function FloatExpr

Declaration:

```
float FloatExpr(char str[]);
```

Description

The same as Expr, but the result is a floating-point number.

Also, see AddrExpr, Expr.

9.3.5.87 Function floor

 Declaration:

```
float floor(float x);
```

Description

The floor function calculates the greatest integer number that is no greater than x.

Returned value

The floor function returns the greatest floating-point number that is no greater than argument x, with the fractional part equal to 0.

9.3.5.88 Function fmod

Declaration:

```
float fmod(float x, float y);
```

Description

The fmod function calculates the remainder of dividing x by y.

Returned function

The fmod function returns the value equal to x - i * y, for integer i, and the absolute value of x - i * y is less than the absolute value of y. The returned value has the same sign as x. If y is equal to 0, then 0 will be returned.

9.3.5.89 Function fnsplit

Declaration:

```
int fnsplit(char path[], char drive[], char dir[], char name[], char ext[]);
```

Description

Selects components of the path to the file. Receives the file name with the path, for example, **C:\PROGRAM\TEST.C**, as the parameter path, and copies the components of the path to appropriate lines. The useful constants for describing the array sizes (MAXPATH, MAXDRIVE, MAXDIR, MAXFILE, MAXEXT) are defined in the system.h file.

If any of the path components is missing, then 0 will be the first character in the corresponding line.

Returned value

Returns the flag word describing the result. Constants corresponding to the flag word bits (WILDCARDS, EXTENSION, ...) are defined in system.h.

9.3.5.90 Function fopen

Declaration:

```
unsigned long fopen(char file_name[], char mode[]);
```

Description

Opens a stream.

fopen opens the file specified by **file_name** and associates a stream with it. fopen returns an unsigned long value to be used as the stream identificator in subsequent operations. The **mode** string used in calls to fopen is one of the following values:

```
   Value          Description
```

```
   --------------------------
   r   Open for reading only.
   w   Create for writing.
   a   Append; open for writing at the end-of-file or create for writing, if the
file does not exist.
   r+  Open an existing file for update (reading and writing).
   w+  Create a new file for update.
   a+  Open for append; open (or create, if the file does not exist) for update
at the end of file.
```

To specify that the given file is being opened or created in the text mode, append t to the value of the type string (for example, rt or w+t).

Similarly, to specify the binary mode, append b to the type string (for example, rb or w+b). If t or b is not given in the type string, then the mode is controlled by the _fmode global variable. If _fmode is set to O_BINARY, then files will be opened in the binary mode. If _fmode is set to O_TEXT, then files will be opened in the text mode.

Note. The O_* constants are defined in file system.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,
*   output cannot be directly followed by input without intervening fseekor rewind;
*   input cannot be directly followed by output without intervening fseek, rewind, or an input that encounters the end-of-file.

Returned Value

On successful completion fdopen returns the unsigned long identifying the stream. In the event of error, it returns 0.

9.3.5.91 Function ForwardTill

Declaration:

```
void ForwardTill(char delimits[]);
```

Description

Moves the cursor right until any character from **delimits** or the end-of-line is reached.

Example:

```
        ForwardTill(" ({[<");
```

9.3.5.92 Function ForwardTillNot

Declaration:

```
void ForwardTillNot(char delimits[]);
```

Description

Moves the cursor right until any character not contained in **delimits** or the end-of-line is reached.

9.3.5.93 Function fprintf

Declaration:

```
int fprintf(unsigned long stream, char format[], ... );
```

Description

Writes formatted output to a **stream**.

fprintf accepts a series of arguments, applies to each of them a format specifier contained in the format string pointed to by **format** and outputs the formatted data to a **stream**. There must be the same number of format specifiers as the arguments.

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the <% CM%> program may crash, because it cannot check the correspondence between the format string and parameters passed. For more, see description of format specifiers for printf.

Returned Value

fprintf returns the number of bytes that were output. In the event of error, it returns EOF.

9.3.5.94 Function fputc

Declaration:

```
int fputc(char c, unsigned long stream);
```

Description

Puts a character on a stream.

fputc outputs character c to the specified stream.

Returned Value

On success, fputc returns character c. On error, it returns EOF.

9.3.5.95 Function fputs

Declaration:

```
int fputs(char s[], unsigned long stream);
```

Description

Outputs a string on a stream.

fputs copies the **s** null-terminated string to the given output **stream**; it does not append the newline character and the terminating null character is not copied.

Returned Value

On success fputs returns a non-negative value. On error it returns the value of EOF.

9.3.5.96 Function FrameRect

Declaration:

```
void FrameRect(unsigned long handle, int x1, int y1, int x2, int y2);
```

Description

Draws an unpainted rectangle using the brush selected with the SelectBrush function. The drawing line

width is always of 1 pixel; (x1, y1) are the coordinates of the upper left corner, (x2, y2) are the coordinates of the lower right corner.

9.3.5.97 Function fread

Declaration:

```
int fread(void s[], int size, int n, unsigned long stream);
```

Description

Reads data from a stream.

fread reads **n** items of data of **size** bytes long each from the given input **stream** into the block pointed to by **s**. The total amount of bytes read is (**n** * **size**).

Returned Value

On success fread returns the number of items (not bytes) actually read. On end-of-file or error it returns a short count (possibly 0).

9.3.5.98 Function FreeLibrary

Declaration:

```
void FreeLibrary(unsigned long inst);
```

Description

De-allocates the specified DLL. HINSTANCE obtained by the LoadLibrary call is transferred as the parameter.

9.3.5.99 Function freopen

Declaration:

unsigned long freopen(char file_name[], char mode[], unsigned long stream);

Description

Associates a new file with an opened stream.

freopen substitutes the specified file in place of the open stream. It closes the stream regardless of whether the open succeeds. freopen is useful for changing the file attached to stdin, stdout, or stderr. The mode string used in calls to fopen is one of the following values:

Value   Description

----

r Open for reading only.

w Create for writing.

a

r+ Open an existing file for update (reading and writing).

w+ Create a new file for update.

a+ Open for append; open (or create, if the file does not exist) for update at the end of file.

To specify that the given file is being opened or created in the text mode, append t to the value of the type string (for example, rt or w+t).

Similarly, to specify the binary mode, append brb or w+b). If t or b is not given in the type string, the mode is controlled by the _fmode global variable. If _fmode is set to O_BINARY, then files will be opened in the binary mode. If _fmode is set to O_TEXT, then files will be opened in the text mode.

Note. The O_* constants are defined in file system.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,
*   output cannot be directly followed by input without intervening fseekor rewind;
*   input cannot be directly followed by output without intervening fseek, rewind, or an input that encounters end-of-file.

On successful completion freopen returns the argument stream. On error it returns NULL.

9.3.5.100 Function frexp

Declaration:

```
float frexp(float x, int exponent[]);
```

Description

The frexp function breaks up the floating-point number f into the normalized mantissa and exponent (the integer power of number two), which is stored in the memory cell indicated by exp.

Returned value

The frexp returns the value of x such that x is the floating-point number in double format ranging from 0.5 to 1 or equal to 0, and the first argument of this function is equal to x multiplied by 2 raised to the power exp.

9.3.5.101 Function fscanf

Declaration:

int fscanf(unsigned long stream, char format[], ...);

Description

Scans and formats input from a .

format. Finally, fscanf stores the formatted input at the address passed to it as the argument following the format. The number of format specifiers and addresses must be the same as the number of input fields.

1.   Your arguments passed to this function shall match the format line. In case of mismatch, the CPI2-B1 program may crash, because it cannot check the correspondence between the format string and parameters passed. For details on format specifiers, see the scanf Format Specifiers.
2.   All arguments for this function shall be arrays, because only the array parameters are passed by

address to functions. Also, see example for scanf.

fscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion on possible causes.

Returned Value

fscanf returns the number of input fields successfully scanned, converted and stored. The return value does not include the scanned fields that were not stored. If fscanf attempts to read at the end-of-file, then EOF will be returned. If no fields are stored, then 0 will be returned.

9.3.5.102 Function fseek

Declaration:

int fseek(unsigned long stream, long offset, int fromwhere);

Description

Repositions a file pointer on a stream.

fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by fromwhereoffset should be 0 or the value returned by ftellfromwhere must be one of the values 0. 1, or 2, which represent three symbolic constants (defined in system.h) as follows:

Constant   fromwhere File location

SEEK_SET   0   Beginning of the file

SEEK_CUR   1   Current file pointer position

SEEK_END   2   End-of-file

fseek discards any character pushed back. fseek is used with stream I/O; for file handle I/O, use lseek. The next operation on the update file after fseek can be either input or output.

Returned Value

fseek will return 0, if the pointer is successfully moved, and nonzero on failure. fseek may return 0 indicating that the pointer has been moved successfully, when in fact it has not been. This is because DOS, which actually resets the pointer, does not verify the setting. fseek returns an error code only on an unopened file or device. In the event of an error return, the errno global variable is set to one of the following values:

EBADF   Bad file pointer

EINVAL   Invalid argument

ESPIPE   Illegal seek on device

9.3.5.103 Function ftell

Declaration:

```
long ftell(unsigned long stream);
```

Description

Returns the current file pointer.

ftell returns the current file pointer for **stream**. The offset is measured in bytes from the beginning of the file (for the binary file). The value returned by ftell can be used in the subsequent call to fseek.

Returned Value

on success ftell returns the current file pointer position. It returns -1L on error and sets the errno global variable to a positive value. In the event of error return, the errno global variable is set to one of the following values:

```
  EBADF       Bad file pointer
  ESPIPE      Illegal seek on device
```

9.3.5.104 Function fwrite

Declaration:

```
int fwrite(void buf[], int size, int n, unsigned long stream);
```

Description

Writes to a stream.

fwrite appends **n** items of data of **size** bytes long each to the given output file. The data written begins at **buf**. The total number of bytes written is (**n** * **size**). In the declarations, **buf** is an array object.

Returned Value

On successful completion fwrite returns the number of items (not bytes) actually written. On error it returns a short count.

9.3.5.105 Function GetByte

Declaration:

```
unsigned int GetByte(unsigned long addr, int addr_space);
```

Description

Reads a byte from the specified address in the specified address space (the addr_space parameter). Constants with the **AS_** prefix for microcontroller memory areas (address spaces) are defined in the **system.h** header file.

Returned value

The read byte.

Example

```
    printf("%02X", GetByte(AS_DATA, 0x1F);
```

9.3.5.106 Function getc

Declaration:

```
int getc(unsigned long stream);
```

Description

Gets character from stream.

getc returns the next character on the given input **stream** and increments the stream's file pointer to point to the next character.

Returned Value

On success, getc returns the character read, after converting it to an int without the sign extension. On the end-of-file or error, it returns EOF.

9.3.5.107 Function getcurdir

Declaration:

```
int getcurdir(int drive, char directory[]);
```

Description

Writes the name of the current directory for the device specified in parameter drive (0 - current disk; 1 - A; 2 - B; ...) to parameter directory.

The received name does not contain the disk name and does not start with symbol ****.

Returned value

0 will be returned, if the name is received successfully, and -1 otherwise

9.3.5.108 Function getcwd

Declaration:

```
void getcwd(char path[]);
```

Description

Gets the current working directory.

getcwd gets the full path name (including the drive) of the current working directory and stores it in **buf**.

9.3.5.109 Function getdate

Declaration:

```
void getdate(int date[]);
```

Description

Obtains the current computer date. The time information is stored in the **date** array:

> **date[0]** - day (1...31)
> **date[1]** - month (1...12)
> **date[2]** - year

Example

```
int date[3];
```

```
getdate(date);
printf("Date: %d/%d/%d", date[0], date[1], date[2]);
```

9.3.5.110 Function getdfree

Declaration:

unsigned long getdfree(int drive);

Description

Gets disk free space.

getdfree accepts a drive specifier in drive (0 for default, 1 for A, and so on) and returns the disk free space in bytes.

9.3.5.111 Function getdisk()

Declaration:

int getdisk();

Description

Gets the current drive number. getdisk gets the current drive number and returns an integer: 0 for A, 1 for B, 2 for C, and so on.

9.3.5.112 Function getenv

Declaration:

int getenv(char name[], char dest[]);

Description

Obtains the value of the name environment variable. The name should be in the upper case and should not end with the equal sign (=). The variable value is copied to dest.

Returned value

1, if the specified variable is found; and 0 otherwise.

Example

```
char value[MAXPATH];
getenv("COMSPEC", value);
```

9.3.5.113 Function GetFileName

Declaration:

void GetFileName(char dest[]);

Description

Copies the name of the current **Edit** window to the **dest** array.

9.3.5.114 Function getftime

Declaration:

unsigned long getftime(long handle);

Description

Gets the file date and time.

getftime retrieves the file time and date for the disk file associated with the open handle. The return value has the following format:

Bits   Value

0...4   two seconds

5...10   minutes

11...15 hours

16...20 days

21...24 months

25...31 year - 1980

Returned Value

getftime returns the file date and time on success. In the event of an error, 0xFFFFFFFF is returned and the errno global variable is set to one of the following values:

EACCES   Permission denied

EBADF   Bad file number

EINVFNC   Invalid function number

9.3.5.115 Function GetLine

Declaration:

void GetLine(char dest[]);

Description

Copies the whole current line to the **dest** array.

9.3.5.116 Function GetMark

Declaration:

void GetMark(int number);

Description

Retrieves the bookmark with the **number** number (1...10).

9.3.5.117 Function GetMemory

Declaration:

```
void GetMemory(void dest[], int n, unsigned long addr, int addr_space);
```

Description

Reads n-byte memory block from the specified address in the specified memory area (the addr_space parameter) to the dest array. Constants with the **AS_** prefix for microcontroller memory areas (address spaces) are defined in the **system.h** header file.

Example

```
char array[20]; GetMemory(array, sizeof(array), 0x20, AS_DATA);
```

9.3.5.118 Function GetScriptFileName

Declaration:

```
void GetScriptFileName(char script_name[], char file_name[]);
```

Description

GetScriptFileName copies to **file_name** the fully qualified path of the script file passed in **script_name**.

Each script has name containing 8 characters: the name of the script source file without path and extension. The GetScriptFileName function retrieves the path to the source file.

Example:

```
char path[MAXPATH];
GetScriptFileName("test", path);
```

9.3.5.119 Function gettime

Declaration:

```
void gettime(int time[]);
```

Description

Obtains the current computer time. The time information is stored in the **time** array:

> **time[0]** - hundredths of a second (0...99)
> **time[1]** - seconds (0...59)
> **time[2]** - minutes (0...59)
> **time[3]** - hours (0...23)

Because the gettime function uses the system timer, you may expect a time error of about 52 milliseconds.

Example

```
int time[4];
while (1)
{
  gettime(time);
  printf("Time: %d:%d:%d.%d", time[3], time[2], time[1], time[0]);
```

```
    }
```

9.3.5.120 Function getw

Declaration:

`int getw(unsigned long stream);`

Description

Gets integer from stream.

getw returns the next integer in the specified input **stream**. It assumes no special alignment in the file. getw should not be used, when the stream is opened in the text mode.

Returned Value

getw returns the next integer on the input stream. On the end-of-file or error, getw returns EOF.

Note. Because EOF is the allowed value for getw to return, feof or ferror should be used to detect the end-of-file or error.

9.3.5.121 Function GetWindowHeight

Declaration:

`int GetWindowHeight(unsigned long handle);`

Description

Obtains the height of the specified window user area.

The handle parameter is the window identifier produced by the call of the OpenWindow, and FindWindow functions.

This function is useful, when it is necessary to draw in the **User** window regardless of its size.

Returned value

The height of the specified window user area in pixels.

9.3.5.122 Function GetWindowWidth

Declaration:

`int GetWindowWidth(unsigned long handle);`

Description

Obtains the width of the specified window user area.

The handle parameter is the window identifier produced by the call of the OpenWindow, and FindWindow functions.

This function is useful, when it is necessary to draw in the **User** window regardless of its size.

Returned value

The height of the specified window user area in pixels.

9.3.5.123 Function GetWord

Declaration:

```
unsigned int GetWord(unsigned long addr, int addr_space);
```

Description

Reads a word (16 bits) from the specified address in the specified memory area (the addr_space parameter). Constants with the **AS_** prefix for microcontroller memory areas (address spaces) are defined in the **system.h** header file.

Returned value

The read word.

Example

```
printf("%04X", GetWord(AS_DATA, 0x1F);
```

9.3.5.124 Function GetWord

Function GetWord

Declaration:

```
unsigned int GetWord(unsigned long addr, int addr_space);
```

Description

Reads a word (16 bits) from the specified address in the specified memory area (the addr_space parameter). Constants with the **AS_** prefix for microcontroller memory areas (address spaces) are defined in the **system.h** header file.

Returned value

The read word.

Example

```
printf("%04X", GetWord(AS_DATA, 0x1F);
```

9.3.5.125 Function GotoXY

Declaration:

```
void GotoXY(int col, int line);
```

Description

Set the cursor position. The cursor is moved to line number 'line' and column number 'col'.

Alternatively, to position the cursor, just assign values to the CurCol and CurLine built-in variables.

9.3.5.126 Function HStep

Declaration:

```
void HStep();
```

Description

Executes one high-level step. Calling this function makes sense only if a program containing the character information is loaded. If no such program is loaded, then calling HStep will be equivalent to

calling the Step function.

Note. The screen is not updated automatically after the HStep call. To organize automatic updates, use the RedrawScreen function at the appropriate moment.

9.3.5.127 Function inport

Declaration:

```
unsigned int inport(unsigned int port_num);
```

Description

Reads a value (word) from the specified parallel port.

Returned value

The read word.

Example

```
unsigned int val = inport(0x300);
```

9.3.5.128 Function inportb

Declaration:

```
unsigned char inportb(unsigned int port_num);
```

Description

Reads a value (byte) from the specified parallel port.

Returned value

The read byte.

Example

```
unsigned char val = inportb(0x3F8);
```

9.3.5.129 Function Inspect

Declaration:

```
unsigned int Inspect(char name[]);
```

Description

Opens the **Inspector window** for the specified name (the name parameter).

9.3.5.130 Function InvertRect

Declaration:

```
void InvertRect(unsigned long handle, int x1, int y1, int x2, int y2);
```

Description

Inverts colors within a rectangular area; (x1, y1) are the coordinates of the upper left corner, (x2, y2) are the coordinates of the lower right corner.

9.3.5.131 Function isalnum

Declaration:

```
int isalnum(unsigned char c);
```

Description

The isalnum function checks, whether parameter c is a Latin alphabet letter or a digit ('A'-'Z', 'a'-'z', or '0'-'9').

Returned value

The isalnum function will return a non-zero value, if c is an alphabetic character or a digit, and will return 0 otherwise.

9.3.5.132 Function isalpha

Declaration:

```
int isalpha(unsigned char c);
```

Description

The isalpha function checks, if parameter c is a Latin alphabet character ('A'-'Z', or 'a'-'z').

Returned value

The isalpha function will return a non-zero value, if c is an alphabetic character, otherwise it will return 0.

9.3.5.133 Function isascii

Declaration:

```
int isascii(unsigned char c);
```

Description

The isascii function checks, if parameter c is an ASCII character.

Returned value

The isascii function will return a non-zero value, if the value of c is greater than or equal to 0 but less than 128.

9.3.5.134 Function isatty

Declaration:

int isatty(long handle);

Description

Checks for device type.

isatty determines, whether handle is associated with any one of the following character devices:

```
a terminal
a console
a printer
a serial port
```

Returned value

If the device is one of the four character devices listed above, then isatty returns a nonzero integer. Otherwise, isatty returns 0.

9.3.5.135 Function iscntrl

Declaration:

```
int iscntrl(unsigned char c);
```

Description

The iscntrl function checks, if parameter c is a control character (from 0x00 to 0x1F, or 0x7F).

Returned character

The iscntrl function will return a non-zero value, if c is a control character or digit, otherwise it will return 0.

9.3.5.136 Function isdigit

Declaration:

```
int isdigit(unsigned char c);
```

Description

The isdigit function checks, if parameter c is a decimal number ('0'-'9').

Returned value

The isdigit function will return a non-zero value, if parameter c is a decimal number, otherwise it will return 0.

9.3.5.137 Function isgraph

Declaration:

```
int isgraph(unsigned char c);
```

Description

The isgraph function checks, if parameter c is a printed character excluding spaces (0x21 - 0x7E).

Returned value

The isgraph function will return a non-zero value, if c is a printed character, otherwise it will return 0.

9.3.5.138 Function islower

Declaration:

```
int islower(unsigned char c);
```

Description

The islower function checks, if parameter c is a lower case letter ('a'-'z').

Returned value

The islower function will return non-zero value, if c is a lower case character, otherwise it will return 0.

9.3.5.139 Function isprint

Declaration:

```
int isprint(unsigned char c);
```

Description

The isprint function checks, if parameter c is a printed character (0x20 - 0x7E).

Returned value

The isprint function will return a non-zero value, if c is an alphabetic character or a digit, otherwise it will return 0.

9.3.5.140 Function ispunct

Declaration:

```
int ispunct(unsigned char c);
```

Description

The ispunct function checks, if parameter c is a punctuation symbol of the following set:

```
!    "    #    $    %    &    '    (
)    *    +    ,    -    .    /    :
;    <    =    >    ?         [    \
]    ^    _    `    {    |    }    ~
```

Returned value

The ispunct function will return a non-zero value, if c is a punctuation symbol, otherwise it will return 0.

9.3.5.141 Function isspace

Declaration:

```
int isspace(unsigned char c);
```

Description

The isspace function checks, if parameter c is a space character (0x09 - 0x0D or 0x20).

Returned function

The isspace function will return a non-zero value, if c is a space character, otherwise it will return 0.

9.3.5.142 Function isupper

Declaration:

```
int isupper(unsigned char c);
```

Description

The isupper function checks, if parameter c is an upper case letter ('A'-'Z').

Returned value

The isupper function will return a non-zero value, if c is an upper case letter, otherwise it will return 0.

9.3.5.143 Function isxdigit

Declaration:

```
int isxdigit(unsigned char c);
```

Description

The isxdigit function checks, if parameter c is a hexadecimal number ('A'-'F', 'a'-'f', '0'-'9').

Returned value

The isxdigit function will return a non-zero value, if parameter c is a hexadecimal number, otherwise it will return 0.

9.3.5.144 Function itoa

Declaration:

```
void itoa(int value, char string[], int radix);
```

Description

Converts an integer number (**value**) into the character string (**string**). The **radix** parameter is the radix of notation (2...36), in which the conversion is carried out.

9.3.5.145 Function LastChar

Declaration:

```
int LastChar(unsigned long handle);
```

Description

Returns the code of the button pressed at the last call of wgetchar or the hexadecimal number entered at the last call of wgethex.

9.3.5.146 Function LastEvent

Declaration:

```
int LastEvent(unsigned long handle);
```

Description

Returns the type of the latest event that occurred to the window and is accessed by the WaitWindowEvent function.

Returned value

The type of event (constants are defined in system.h):

WE_REDRAW is the window data update request, an image display request. This event is generated in all those cases, when it is necessary to update the window, for example, at the Windows task switch. This event informs you that the window wishes to redraw itself, and your script file, generally speaking, does not have to respond to this event. If the script file does not update the window data, the old picture

will be drawn.

WE_MOUSEBUTTON (only the **User** window) - You clicked a mouse button, when the mouse cursor was in the window. Information on the click can be obtained by calling the LastEventIntx function:

- LastEventInt1() and LastEventInt2() return the coordinates in pixels (x, y) for the point, where the cursor was located, when the button was clicked.
- LastEventInt3() and LastEventInt4() return the text coordinates (x, y) for the point, where the cursor was located, when the button was clicked; x is the column number; y is the line number.

WE_USERBUTTON (only the **User** window) You clicked one of the buttons added to the window by the AddButton function. The LastEventInt1() function returns identifier of the clicked button. It equivalent to the button identifier returned by the AddButton function.

WE_TOOLBARBUTTON (only the **User** window) You clicked one of the **0**...**F** buttons on the window toolbar. These buttons are particularly intended for simple interactions with the window. Using the customer buttons (see AddButton) is more complicated, although it is more flexible.

WE_CHAR - (only the **I/O Stream** window) You pressed an alphanumeric key on the keyboard. LastEventInt1() returns its code.

WE_CLOSE - You closed the window. After that, further window operation is useless and should be stopped.

9.3.5.147 Function LastEventInt{1...4}

Declaration:

```
int LastEventInt{1...4}(unsigned long handle);
```

Description

Four functions - LastEventInt1(), LastEventInt2(), LastEventInt3(), and LastEventInt4() - return parameters that are generated upon event occurrence in a **user** window. See LastEvent, WaitWindowEvent.

9.3.5.148 Function LastString

Declaration:

```
int LastString(unsigned long handle, char s[]);
```

Description

Copies the string entered at the last call of wgetstring to the string (the **s** parameter).

9.3.5.149 Function LineTo

Declaration:

```
void LineTo(unsigned long handle, int x, int y);
```

Description

Draws a line from the point set up by the MoveTo or LineTo function to the point with coordinates (x, y). The line is drawn with the pen selected with the SelectPen function (or a standard pen, when SelectPen was not called). After the LineTo call, the benchmark is moved to the destination point.

Example

```
// To draw triangle ABC
MoveTo(handle, 10, 10);  // point A
LineTo(handle, 50, 50);  // A --> B
```

```
LineTo(handle, 20, 40);  // B --> C
LineTo(handle, 10, 10);  // C --> A
```

9.3.5.150 Function LoadDesktop

Declaration:

```
void LoadDesktop(char file_name[]);
```

Description

Downloads the specified screen configuration file (see Configuration Files).

9.3.5.151 Function Left

Declaration:

```
void Left(int count=1);
```

Description

Move the cursor **count** positions left. The same result can be achieved by decrementing the CurCol built-in variable.

9.3.5.152 Function LoadLibrary

Declaration:

```
unsigned long LoadLibrary(char lib_name[]);
```

Description

Loads the specified DLL by calling the LoadLibrary function of Windows API. After the loading, the functions from this DLL can be called with the CallLibraryFunction.

Returned value

What is returned by the LoadLibrary function of Windows API, that is, HINSTANCE of the loaded DLL or error code.

Example

```
unsigned long instance = LoadLibrary("EXTEND.DLL");
```

9.3.5.153 Function LoadOptions

Declaration:

```
void LoadOptions(char file_name[]);
```

Description

Downloads the specified option file (see Configuration Files).

9.3.5.154 Function LoadProgram

Declaration:

```
void LoadProgram(unsigned char file_name[], int format, int addr_space=AS_CODE,
unsigned long start_addr=0);
```

Description

Downloads a program into the microcontroller memory.

Parameters:

```
file_name - the name of the loaded file.
format    - the format of the loaded file. Character constants with the
             prefix LF_ declared in the system.h header file
             are provided for this parameter. To understand this
             better, open the Load Program dialog
             and see the list of formats.
addr_space - the microprocessor address space, where the program is downloaded
             (the code memory by default).
start_addr - the load address. This parameter is used only for loading
             a file that is the binary memory image.
```

Not only programs can be loaded: you can also load data memory images that were saved, for example, with the [SaveData](#) function.

Example

```
LoadProgram("C:\\PROG\\TEST.D32", LF_UBROF);
```

9.3.5.155 Function LoadProject

Declaration:

```
void LoadProject(char file_name[]);
```

Description

Loads the project with the specified name. If no extension is specified, then '.IDE' will be assumed.

<%CM%> will perform the same actions as if the project were loaded via menu.

9.3.5.156 Function locking

Declaration:

```
int locking(long handle, int cmd, long length);
```

Description

Sets or resets file-sharing locks.

locking provides interface to the operating system file-sharing mechanism. **handle** specifies the opened file to be locked or unlocked. The region to be locked or unlocked starts at the current file position, and is **length** bytes long. Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into the locked region will retry the operation three times. If all three retries fail, the call fails with an error. **cmd** specifies the action to be taken:

```
0    Unlock the region, which must have been previously locked.
1    Lock the region. If the lock is unsuccessful, try once a second for 10
seconds before giving up.
2    Lock the region. If the lock if unsuccessful, give up immediately.
```

Returned Value

On successful operations, locking returns 0. Otherwise, it returns -1 and the [errno](#) global variable is set to one of the following values:

```
EACCES        File already locked or unlocked
EBADF         Bad file number
EDEADLOCK     File cannot be locked after 10 retries (cmd is LK_LOCK or LK_RLCK)
EINVAL        Invalid cmd, or SHARE.EXE not loaded
```

9.3.5.157 Function log

Declaration:

```
float log(float x);
```

Description

The log function calculates the natural logarithm of the floating-point number val.

Returned function

The log function returns the natural logarithm of val. If val is negative or equal to 0, then the function will return _MINUS_INF.

9.3.5.158 Function log10

Declaration:

float log(float x);

Description

The log function calculates the natural logarithm of the floating-point number val.

Returned Value

The log function returns the natural logarithm of val. If val is negative or equal to 0, then the function will return _MINUS_INF.

9.3.5.159 Function lseek

Declaration:

long lseek(long handle, long offset, int fromwhere);

Description

Moves file pointer.

lseek sets the file pointer associated with handle to a new position, which is offset bytes beyond the file location specified by fromwhere. fromwhere must be one of the following symbolic constants (defined in system.h):

SEEK_CUR   Current file pointer position

SEEK_END   End-of-file

SEEK_SET   File beginning

Returned Value

lseek returns the offset of the pointer new position measured in bytes from the file beginning. lseek

returns -1L on error, and the errno global variable is set to one of the following values:

EBADF   Bad file handle

EINVAL   Invalid argument

ESPIPE   Illegal seek on device

For the devices incapable of seeking (such as terminals or printers), the return value is undefined.

9.3.5.160 Function ltoa

Declaration:

```
void ltoa(long value, char string[], int radix);
```

Description

Converts a long integer number (**value**) into the character string (**string**).

The **radix** parameter is the radix used for conversion (2...36).

9.3.5.161 Function MaxAddr

Declaration:

```
unsigned long MaxAddr(int addr_space);
```

Description

Returns the upper boundary address of the processor address space. Constants with the **AS_** prefix for the **addr_space** parameter are defined in the system.h header file.

Example

See MinAddr

9.3.5.162 Function memccpy

Declaration:

```
int memccpy(void dest[], void src[], int c, int n, int dest_index=0, int
src_index=0);
```

Description

The memccpy function copies the contents of the scr memory block to the dest memory block. Copying stops, when either byte with the value of c is encountered and copied or when c bytes are copied.

Returned value

The memccpy function returns the number of copied bytes.

9.3.5.163 Function memchr

Declaration:

```
int memchr(void s[], int c, int n, int index=0);
```

Description

The memchr function searches for the first entry of character c (which was earlier converted into the unsigned char) among the first n characters (interpreted as the unsigned char) of the object specified by s.

Returned value

The memchr function returns the number of the found byte counting from the beginning of the array, or **-1**, if the byte is not found

9.3.5.164 Function memcmp

Declaration:

```
int memcmp(void s1[], void s2[], int n, int s1_index=0, int s2_index=0);
```

Description

The memcmp function compares the first n bytes of objects s1 and s2 and returns the comparison result. The bytes are interpreted as the unsigned char.

```
Result              Meaning
-----------------------------------------
< 0                 s1 is less than s2
= 0                 but1 is equal to s2
> 0                 s1 is greater than s2
```

Returned value

The memcmp function returns the positive, negative, or zero value depending on the result of comparing the first n bytes of objects s1 and s2.

9.3.5.165 Function memcpy

Declaration:

```
void memcpy(void dest[], void src[], int n, int dest_index=0, int src_index=0);
```

Description

The memcpy function copies n bytes from the buffer specified by scr to the buffer specified by dest. If these buffers have common memory cells (that is, they overlap), then the memcpy function does not ensure that byte copying is executed correctly. If overlapping is possible, then use the memmove function instead.

Returned value

None.

9.3.5.166 Function memicmp

Declaration:

```
int memicmp(void s1[], void s2[], int n, int s1_index=0, int s2_index=0);
```

Description

The memicmp function compares the first n bytes of objects s1 and s2 regardless of the character case, and returns the comparison result. The bytes are interpreted as the unsigned char.

```
Result              Meaning
----------------------------------------
```

```
< 0                s1 is less than s2
= 0                but1 is equal to s2
> 0                s1 is greater than s2
```

Returned value

The memicmp function returns the positive, negative or zero value, depending on the result of comparing the first n bytes of objects s1 and s2.

9.3.5.167 Function memmove

Declaration:

```
void memmove(void dest[], void src[], int n, int dest_index=0, int src_index=0);
```

Description

The memmove function copies n bytes from the buffer specified by scr to the buffer specified by dest. When these buffers have common memory cells (that is, they overlap), the memmove function ensures that bytes are copied correctly.

Returned value

None.

9.3.5.168 Function memset

Declaration:

```
void memset(void s[], int c, int n, int index=0);
```

Description

The memset function sets the first n bytes of the object, specified by s, equal to the value transferred to c (and converted into the unsigned char).

Returned value

None.

9.3.5.169 Function MessageBox

Declaration:

```
int MessageBox(char format[], ... );
```

Description

The MessageBox function displays data in accordance with the format line in the form of a dialog message.

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the <% CM%> program may crash, because it cannot check the correspondence between the format string and parameters passed.

Returned value

1, if the **Close** button is pressed;

0, if the **Esc** key is pressed.

Also, see:

9.3.5.170 Function MessageBoxEx

Declaration:

```
int MessageBoxEx(int flags, char title[], char format[], ... );
```

Description

This function displays data in accordance with the format line in the form of a dialog message. The dialog has title, buttons and icon, which are specified by **flags** and **title**.

The **flags** parameter may contain one or several flags that determine the dialog buttons and icon. For these flags, file **system.h** defines constants with the MB_ prefix.

The **title** parameter is the text in the dialog title bar.

The **format** parameter is the format string, it may be followed by data (see printf).

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the <% CM%> program may crash, because it cannot check the correspondence between the format string and parameters passed.

Returned value

The function returns one of constants with the ID prefix determined in **system.h**, which corresponds the dialog button pressed.

Example:

```
if (MessageBoxEx(MB_YESNO | MB_ICONQUESTION, "Confirm exit", "Do you want to
exit?") == IDYES)
  ExitProgram();
```

Also, see:

9.3.5.171 Function MinAddr

Declaration:

```
unsigned long MinAddr(int addr_space);
```

Description

Returns the lower boundary address of the processor address space. Constants with the **AS_** prefix for the **addr_space** parameter are defined in the system.h header file.

Example

```
// To set the whole data memory to zero
int i;
for (i = MinAddr(AS_DATA), i <= MaxAddr(AS_DATA); i++)
  SetByte(i, AS_DATA, 0);
```

9.3.5.172 Function mkdir

Declaration:

```
int mkdir(char path[]);
```

Description

Creates a directory. mkdir creates a new directory from the given path name path.

Returned Value

mkdir will return 0, if the new directory is created.

The returned value of -1 indicates an error and the errno global variable contains one of the following values:

EACCES   Permission denied

ENOENT   No such file or directory

9.3.5.173 Function MoveTo

Declaration:

```
void MoveTo(unsigned long handle, int x, int y);
```

Description

Sets up the coordinates of the start point of the line to be drawn with the LineTo function.

Examples

```
   // To draw a line from the point with coordinates (10, 10) to the point (50,
50).
   MoveTo(handle, 10, 10);
   LineTo(handle, 50, 50);
```

9.3.5.174 Function MoveWindow

Declaration:

```
void MoveWindow(unsigned long handle, int x, int y);
```

Description

Moves the specified window. The handle parameter is the window identifier produced by the call of the OpenWindow, and FindWindow functions. **x** and **y** are the new coordinates (in pixels) of the window upper left corner in the user area of the <%CM%> window. Coordinates 0, 0 correspond to the window upper left corner.

The window size does not change.

9.3.5.175 Function movmem

Declaration:

```
void movmem(void dest[], void src[], unsigned int length, int dest_index=0, int
src_index=0);
```

Description

The movmem function copies length bytes from the buffer specified by scr to the buffer specified by dest. When these buffers have common memory cells (that is, they overlap), the movmem function ensures that byte are copied correctly.

Returned value

None.

9.3.5.176 Function open

Declaration:

int open(char path[], int access);

Description

Opens a file for reading or writing.

open opens the file specified by path and prepares it for reading and/or writing as determined by the value of access. To create a file in a particular mode, you can either assign to the _fmode global variable or call open with the O_CREAT options ORed with the translation mode desired. For example, the call:

open("XMP", O_CREAT | O_BINARY);

creates a binary-mode file named XMP, truncating its length to 0 bytes, if it already exists. For open, access is constructed by performing the bitwise OR with the flags from the following list. Only one flag from the first list can be used (and one must be used); the remaining flags can be used in any logical combination. These symbolic constants are defined in system.h.

Read/Write Flags:
```
O_RDONLY      Open for reading only.
O_WRONLY      Open for writing only.
O_RDWR        Open for reading and writing
```

Returned Value

On success, open returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file. On error, open returns -1 and the errno global variable is set to one of the following values:
```
EACCES        Permission denied
EINVACC       Invalid access code
EMFILE        Too many open files
ENOENT        No such file or directory
```

9.3.5.177 Function OpenEditorWindow

Declaration:

unsigned long OpenEditorWindow(char file_name[]);

Description

Opens the **Source** window and loads the specified file into it.

If the window with the specified file is already opened, it will become active and the new window will not be opened.

9.3.5.178 Function OpenStreamWindow

Declaration:

`unsigned long OpenStreamWindow(char title[]);`

Description

Opens the **I/O Stream window** window and sets up its title (the title parameter).

You can do the same with the OpenWindow function, by transferring the WIN_STREAM constant to it as a parameter, however in this case, you cannot set up the title.

If there is an "unowned" stream window on the screen, the new window will not be opened and the already opened window will be used.

The new window opens in a random place on the screen and has certain preset size. To resize the window, use the SetWindowSize function, or do it manually.

Returned value

The identifier of opened window. It can be transferred to other window operation functions as a parameter.

Example

`unsigned long handle = OpenStreamWindow("Serial port I/O");`

9.3.5.179 Function OpenUserWindow

Declaration:

`unsigned long OpenUserWindow(char title[]);`

Description

Opens the **User window** and specifies its title (parameter title).

This can also be done with the OpenWindow function, by transferring the WIN_USER constant to it as a parameter, however in this case, you cannot specify the title.

If there is an unowned user window opened on the screen, a new window will not be opened and the current window will be used.

A new window is opened in a random screen location and has the preset size. To resize the window, use the SetWindowSize function or do it manually.

Returned value

The identifier of the opened window. It can be transferred as a parameter to other window operation functions.

Example

`unsigned long handle = OpenUserWindow("A/D conversion");`

9.3.5.180 Function OpenWindow

Declaration:

`unsigned long OpenWindow(int type);`

Description

Opens the specified window type (disassembler, dump, etc.). The constants to describe the window types are declared in the **system.h** header file:

```
WIN_CONSOLE        - Console
WIN_DUMP           - Memory Dump
WIN_AUTO_WATCHES   - AutoWatches
WIN_INSPECT        - Inspector
WIN_SF_SOURCE      - Script source
WIN_STREAM         - I/O stream
WIN_USER           - User window
```

The **View** menu gives access to the available windows.

The window will be opened, if an instruction of the **View** menu is executed. If you need to move a window and/or change its size, use the SetWindowSize, SetWindowSizeT, or MoveWindow functions.

Returned value

The identifier of the opened window. It can be transferred as a parameter to other window operation functions.

For Windows programmers: identifier is a window HWND.

9.3.5.181 Function outport

Declaration:

```
void outport(unsigned int port_num, unsigned int value);
```

Description

Writes a value (word) to the specified parallel port.

9.3.5.182 Function outportb

Declaration:

```
void outportb(unsigned int port_num, unsigned char value);
```

Description

Write a value (byte) to the specified parallel port.

9.3.5.183 Function peek

Declaration:

```
int peek(unsigned int segment, unsigned int offset);
```

Description

Reads a word from computer memory by a specified segment: offset. The segment is a **selector**.

Returned value

The read word.

9.3.5.184 Function peekb

Declaration:

```
unsigned char peekb(unsigned int segment, unsigned int offset);
```

Description

Reads a byte from the computer memory by a specified segment:offset. The segment is a **selector**.

Returned value

The read byte.

9.3.5.185 Function poke

Declaration:

```
void poke(unsigned int segment, unsigned int offset, int value);
```

Description

Writes a word to the computer memory by a specified segment: offset. The segment is a **selector**.

9.3.5.186 Function pokeb

Declaration:

```
void pokeb(unsigned int segment, unsigned int offset, unsigned char value);
```

Description

Writes a byte to the computer memory by specified segment: offset. segment is a **selector**.

9.3.5.187 Function Polyline

Declaration:

```
void Polyline(unsigned long handle, unsigned int points[], int n);
```

Description

Connects the points, whose coordinate pairs are transferred in the **points** array, with a line. The **n** parameter is the amount of points. Each subsequent horizontal coordinate should be greater than the previous one.

Example

```
Polyline(handle, { 0, 0,
                   10, 20,
                   12, 30,
                   78, 10 }, 4);
```

9.3.5.188 Function pow

Declaration:

```
float pow(float x, float y);
```

Description

The pow function raises x to the power y.

Returned function

The pow function returns the result of raising x to the power y. If y is equal to 0, then the function will return 1.0. If x == 0 and y < 0, then the error will occur (falling outside the range) and the function will return 0. If x < 0 and y is not an integer, then the error of falling outside the range will also occur and the pow function

will return 0.

9.3.5.189 Function pow10

Declaration:

```
float pow10(int x);
```

Description

The pow10 function raises number 10 to the power x.

Returned value

The pow10 function returns the result of raising 10 to the power x. If x is 0, then the function will return 1.0.

9.3.5.190 Function printf

Declaration:

void printf(char format[], ...);

Description

The printf function displays the values of transferred parameters in the Console in accordance with the format line.

Upon every printf access, data is displayed in the new window line, that is, "\n" is automatically added to the displayed string.

If the **Console** window is already opened, it will be automatically opened.

The wprintf function provides more capabilities for the formatted output, but it requires certain preparatory operations.

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the CPI2-B1 program may crash, because it cannot check the correspondence between the format string and parameters passed.

For more info, see:

Format String

Format Specifiers

Flag Characters

Width Specifiers

Precision Specifiers

Input-size Modifiers

Type Characters

Format Specifier Conventions

Returned Value

.

Example

printf("Counter = %d\n"

"Value = %08lX",

Counter, Value);

9.3.5.190.1 printf Conversion Type Characters

The information in this table is based on the assumption that no flag characters, width specifiers, precision specifiers, or input-size modifiers were included in the format specifier.

Note. Certain accompany some of these format specifiers.

Type Char	Expected Input	Format of output
Numerics		
d	Integer	signedinteger
i	Integer	signed decimal integer
o	Integer	unsigned octal integer
u	Integer	unsigned decimal integer
x	Integer	unsigned hexadecimal int (with a, b, c, d, e, f).
X	Integer	unsigned hexadecimal int (with A, B, C, D, E, F).
f	Floating-point	signed value of the form [-]dddd.dddd.
e	Floating-point	signed value of the form [-]d.dddd or [+/-]ddd
g	Floating-point	signed value in either ef form, based on given value and precision. Trailing zeros and the decimal point are printed if necessary.
E	Floating-point	Same as e; with E for exponent.
G	Floating-point	Same as g; with E for exponent if e format used.
Characters		
c	Character	Single character.
s	String pointer	
%	None	Prints the % character.

Infinite floating-point numbers are printed as +INF and -INF.

An IEEE Not-A-Number is printed as +NAN or -NAN.

9.3.5.190.2 printf Flag Characters

The Flag characters can appear in any order and combination.

Flag   Description

-              Left-justifies the result, pads on the right with blanks. If not given, it right-justifies the
               result, pads on the left with zeros or blanks.
+   Signed conversion results always begin with a plus (+) or minus (-) sign.
blank   If value is nonnegative, the output begins with a blank instead of a plus; negative
               values still begin with a minus.
Specifies that arg is to be converted using an alternate form.

Note. Plus (+) takes precedence over blank () if both are given

9.3.5.190.3 printf Format Specifier Conventions

Certain conventions accompany some of the printf format specifiers for the following conversions:

- %e or %E

- %f

- %g or %G

- %x or %X

Note. Infinite floating-point numbers are printed as +INF and -INF. An IEEE Not-a-Number is printed
as +NAN or -NAN.

9.3.5.190.3.1 %e or %E Conversions

The argument is converted to match the style

[-] d.ddd...e[+/-]ddd

where:
•   one digit precedes the decimal point
•   the number of digits after the decimal point is equal to the precision;
•   the exponent always contains at least two digits.

9.3.5.190.3.2 %f Conversions

The argument is converted to decimal notation in the style

[-] ddd.ddd...

where the number of digits after the decimal point is equal to the precision (if a non-zero precision
was given).

9.3.5.190.3.3 %g or %G Conversions

The argument is printed in style e, E or f, with the precision specifying the number of significant digits.

Trailing zeros are removed from the result, and a decimal point appears only if necessary.

The argument is printed in style e or f (with some restraints) if g is the conversion character. Style e is used only if the exponent that results from the conversion is either greater than the precision or less than –4.

The argument is printed in style if G is the conversion character.

9.3.5.190.3.4 %x or %X Conversions

For x conversions, the letters a, b, c, d, e, and f appear in the output.

For X conversions, the letters A, B, C, D, E, and F appear in the output.

9.3.5.190.3.5 Alternate Forms for printf Conversion

If you use the # flag conversion character, it has the following effect on the argument (arg) being converted:

Conversion character	How # affects the argument
c s d iu	No effect.
0	0 is prepended to a nonzero arg.
x X	0x (or 0X) is prepended to arg.
e E f	The result always contains a decimal point even if no digits follow the point. Normally, a decimal point appears in these results only if a digit follows it.
g G	Same as e and E, except that trailing zeros are not removed.

9.3.5.190.4 printf Format Specifiers

The printf format specifiers have the following form:

% [flags] [width] [.prec] [F|N|h|l|L] type_char

Each format specifier begins with the percent character (%). After the % come the following optional specifiers, in this order:

Optional Format String Components

These are the general aspects of output formatting controlled by the optional characters, specifiers, and modifiers in the format string:

Component	Optional/Required	
[flags]	(Optional)	Flag character(s) Output justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes.
[width]	(Optional)	Width specifier Minimum number of characters to print, padding with blanks or zeros.
	(Optional)	Precision specifier Maximum number of characters to print; for integers, minimum number of digits to print.
[F|N|h|l|L]	(Optional)	Input size modifier Override default size of next input argument:H = short int
		L = long
		L = long double
type_char	(Required)	Conversion-type character.

9.3.5.190.5 printf Format String

The format string shall be present in each of the printf function calls. It controls how each function will convert, format, and print its arguments. The format string is a character string that contains two types of objects:
• Plain characters are copied verbatim to the output stream.
• Conversion specifications fetch arguments from the argument list and apply formatting to them.

Plain characters are just copied verbatim to the output stream. Conversion specifications fetch arguments from the argument list and apply formatting to them.

Note. There must be enough arguments for the format; if not, the results will be unpredictable and possibly disastrous. Excess arguments (more than required by the format) are ignored.

9.3.5.190.6 printf Input-size Modifiers

These modifiers determine how printf functions interpret the next input argument, arg[f].

Modifier	Type of arg	arg is interpreted as ...
F	p, s,	A far pointer
N	and n)	A near pointer (Note. N cannot be used with any conversion in the huge model.)
h	d i o u x X	A short int
l	d i o u x X	A long int

	e E f g G	A double
L	e E f g G	A long double

arg.

Both F and N reinterpret the input variable arg. Normally, the arg for a p, %s, or n conversion is a pointer of the default size for the memory model.

h, l, and L override the default size of the numeric data input arguments. Neither h nor l affects character (c,s) or pointer () types.

9.3.5.190.7 printf Precision Specifiers

The printf precision specifiers set the maximum number of characters (or minimum number of integer digits) to print. A printf precision specification always begins with a period (".") to separate it from any preceding width specifier.

Then, like the width specifier, precision is specified in one of two ways:
• directly, through a decimal digit string;
• indirectly, through an asterisk (*).

If you use an * for the precision specifier, the next argument in the call (treated as an int) specifies the precision.

If you use asterisks for the width or the precision, or for both, the width argument must immediately follow the specifiers, followed by the precision argument, then the argument for the data to be converted.

[.prec]	How Output Precision Is Affected
(none)	Precision set to default:
	= 1 for d,i,,u,x,X types;
	= 6 for e,E,f types;
	= All significant digits for g,G types;
	= Print to first null character for s types;
	= No effect on types.
.0	For d,i,o,u,x types, precision set to default.
	for e,E,f types, no decimal point is printed.
.n	n characters or n decimal places are printed.
	If the output value has more than n characters, the output might be truncated or rounded. (Whether this happens depends on the type character.)
.	The argument list supplies the precision specifier, which must precede the actual argument being formatted.

No numeric characters will be output for a field (i.e., the field will be blank) if the following conditions are all met:
• you specify an explicit precision of 0;
• the format specifier for the field is one of the integer formats (d, i, o, u, or x);
• the value to be printed is 0

How [.prec] Affects Conversion

Char Type	Effect of [.prec] (.n) on Conversion

d	Specifies that at least n digits are printed.
i	n digits,
o	output value is left-padded x with zeros.
u	If input argument has more than n digits,
x	the output value is not truncated.

e	Specifies that n characters are
E	printed after the decimal point, and
f	the last digit printed is rounded.

g	Specifies that at most n significant
G	digits are printed.

c	Has no effect on the output.
s	Specifies that no more than n characters are printed.

9.3.5.190.8 printf Width Specifiers

The width specifier sets the minimum field width for an output value. Width is specified in one of two ways:
- directly, through a decimal digit string;
- indirectly, through an asterisk (*).

If you use an asterisk for the width specifier, the next argument in the call (which must be an int) specifies the minimum output field width.

Nonexistent or small field widths do cause truncation of a field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Width specifier	How output width is affected
n	At least n characters are printed. If the output value has less than n characters, the output is padded with blanks (right-padded if - flag given, left-padded otherwise).
0n	At least n characters are printed. If the output value has less than n characters, it is filled on the left with zeros.
*	The argument list supplies the width specifier, which must precede the actual argument being formatted.

9.3.5.191 Function pscanf

Declaration:

int pscanf(char title[], char format[], ...);

Description

performs the same as scanf; however, it receives an additional parameter, the header of the prompt dialog box.

pscanf scans a series of input fields one character at a time reading from a stream. After that, each field is formatted in accordance with a format specifier passed to pscanf in the format string pointed to by format. Finally, pscanf stores the formatted input at the address passed to it as the argument following the format. The number of format specifiers and addresses must be the same as the number of input fields.

Notes
1. scanf Format Specifiers.
2. All arguments for this function shall be arrays, because only the array parameters are passed by address to functions. Also, see example for scanf.

pscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion on possible causes.

Returned Value

pscanf returns the number of input fields successfully scanned, converted and stored. The return value does not include the scanned fields that were not stored. If no fields are stored, then 0 will be returned.

9.3.5.192 Function putc

Declaration:

int putc(int c, unsigned long stream);

Description

Outputs a character to a stream.

putc outputs character c to the stream specified by stream.

Returned Value

On success, putc returns the character printed, c. On error, putc returns EOF.

fprintf

fputc

fputs

fwrite

getc

printf

putw

9.3.5.193 Function putenv

Declaration:

int putenv(char name[]);

Description

Sets up the value of the environment variable. Here, is a string like:

"COMSPEC=C:\\COMMAND.COM"

Returned value

1, if the value of specified variable is set up; otherwise it returns 0.

9.3.5.194 Function putw

Declaration:

int putw(int c, unsigned long stream);

Description

Puts an integer on a stream.

putw outputs integer c to the given . putw neither expects nor causes special alignment in the file.

Returned Value

On success, putw returns integer c. On error, putw returns EOF. Because EOF is the allowed integer, use ferror to detect errors with putw.

9.3.5.195 Function rand

Declaration:

int rand();

Returns a pseudorandom number in the range from 0 to 32767.

9.3.5.196 Function random

Declaration:

int random(int num);

Description

-1.

9.3.5.197 Function randomize

Declaration:

void randomize();

Description

Initializes a random number generator by a random number.

9.3.5.198 Function read

Declaration:

int read(long handle, void buf[], int len);

Description

Reads from file.

read attempts to read len bytes from the file associated with handle into the buffer pointed to by buf. For a file opened in text mode, then read removes the carriage returns and reports the end-of-file, when it reaches Ctrl-Z. The handle file handle is obtained from the creat, open, dup, or dup2 call. On disk files, read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

Returned Value

On successful completion, read returns an integer indicating the number of bytes placed in the buffer. If the file is opened in the text mode, then read does not count the carriage returns or Ctrl-Z characters in the number of bytes read. On the end-of-file, read returns 0. On error, read returns -1 and sets the errno global variable to one of the following two values:

EACCES	Permission denied
EBADF	Bad file number

9.3.5.199 Function Rectangle

Declaration:

void Rectangle(unsigned long handle, int x1, int y1, int x2, int y2);

Description

Draws an unpainted rectangle using the pen selected with the SelectPen function and paints it using the brush selected with the SelectBrush function; (x1, y1) are the coordinates of the upper left corner; (x2, y2) are the coordinates of the lower right corner.

9.3.5.200 Function RedrawScreen

Declaration:

void RedrawScreen();

Description

Updates all open windows of the name. Use this function, when the script file changes the microcontroller resources and you want to view the result of the change. A script file cannot update the screen on its own, because it takes significant time (as compared with the script file execution speed).

Example:

 SetByte(addr, AS_DATA, 0x11);

 RedrawScreen();

9.3.5.201 Function ReloadProgram

Declaration:

void ReloadProgram();

Description

Reloads a program that was the last loaded into the microcontroller memory. It is equivalent to the Re-Load program in the File menu.

9.3.5.202 Function RemoveButtons

Declaration:

void RemoveButtons(unsigned long handle);

Description

Removes all buttons from the window that were added by the AddButton function. This function is useful, when a script file is restarted and the user window used by this script file contains buttons generated by the script file during the previous run.

9.3.5.203 Function rename

Declaration:

int rename(char oldname[], char newname[]);

Description

Renames a file.

rename changes the name of a file from oldname to newname

Directories in oldname and newname need not be the same, so rename can be used to move a file from one directory to another. Wildcards are not allowed.

This function will fail (EACCES), if either file is currently open in any process.

Returned Value

On success, rename returns 0. On error (if the file cannot be renamed), it returns -1 and the global variable is set to one of the following values:

EACCES   Permission denied: filename already exists or the path is invalid

ENOENT   No such file or directory

ENOTSAM   Not same device

9.3.5.204 Function rewind

Declaration:

void rewind(unsigned long stream);

Description

Repositions the file pointer to the beginning of the stream.

rewind(stream) is equivalent to fseek (stream, 0L, SEEK_SET), except that rewind clears the end-

of-file and error indicators, while fseek clears the end-of-file indicator only. After rewind, the next operation on the update file can be either input or output.

9.3.5.205 Function Right

Declaration:

void Right(int count=1);

Description

Move the cursor positions right. The same result can be achieved by incrementing the CurCol built-in variable.

9.3.5.206 Function rmdir

Declaration:

int rmdir(char path[]);

Description

Removes a directory.

rmdir deletes the directory, whose path is given by path. The directory named by path:

> must be empty

> must not be the current working directory

> must not be the root directory

Returned Value

rmdir will return 0, if the directory is successfully deleted. The returned value of -1 indicates an error and the errno global variable contains one of the following values:

EACCES   Permission denied

ENOENT   Path or file function not found

9.3.5.207 Function SaveData

Declaration:

void SaveData(unsigned char file_name[], int format, int addr_space, unsigned long start_addr, unsigned long end_addr);

Description

Saves the microcontroller memory area in the file.

Parameters:

file_name - the name of unloaded file.

format   - the format of unloaded file. Character constants with

the prefix SF_ declared in the system.h header file

are provided for this parameter. To understand this better,

open the Save file dialog and go through the

format names.

addr_space - the microcontroller memory space, from where data is unloaded.

start_addr - the initial address of unloaded area.

end_addr - the final address of unloaded area (inclusive).

Example

SaveData("C:\\PROG\\TEST.HEX", SF_HEX, AS_CODE, 0, 0x3FFF);

9.3.5.208 Function SaveDesktop

Declaration:

void SaveDesktop(char file_name[]);

Description

Saves the screen configuration in the specified file (see Configuration Files)

9.3.5.209 Function SaveFile

Declaration:

int SaveFile();

Description

Saves the file from the current window.

9.3.5.210 Function SaveOptions

Declaration:

void SaveOptions(char file_name[]);

Description

Saves the options in the specified file (see Configuration Files

9.3.5.211 Function scanf

Declaration:

int scanf(char format[], ...);

Description

The scanf function displays prompt to enter a character string. The string you enter is parsed in

accordance with the format line.

scanf scans a series of input fields one character at a time reading from a stream. After that, each field is formatted in accordance with a format specifier passed to scanf in the format string pointed to by format. Finally, scanf stores the formatted input at the address passed to it as the argument following the format. The number of format specifiers and addresses must be the same as the number of input fields.

Notes
1.	Your arguments passed to this function shall match the format line. In case of mismatch, the CPI2-B1 program may crash, because it cannot check the correspondence between the format string and parameters passed. For details on format specifiers, see the scanf Format Specifiers.
2.	All arguments for this function shall be arrays, because only the array parameters are passed by address to functions. Also, see example below.

scanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion on possible causes.

Returned Value

scanf returns the number of input fields successfully scanned, converted and stored. The return value does not include the scanned fields that were not stored. If no fields are stored, then 0 will be returned.

Example

	int i[1];

	float f[1];

	char name[64];

	scanf("%d %f %s", i, f, name);

	// If "123 4.56 String" is entered in the prompt, then:

	// i[0] will assume value 123,

	// name will be equal to the string "String".

9.3.5.212 Function Search

Declaration:

int Search(char text[], int in_block=0);

Description

Searches for text text. The search area is defined by the in_block parameter: if it is 0, the search will be performed in the whole text, otherwise, in the marked block only.

The search is always performed from the cursor position.

The search options are defined by the CaseSensitive, WholeWords and RegularExpressions built-in variables.

If text is found, then Search will return 1, otherwise it will return 0. The string that was found is

copied to the LastFoundString variable. This is because the found string may not be the same as the search argument

9.3.5.213 Function searchpath

Declaration:

int searchpath(char file_name[], char path[]);

Description

Searches the operating system path for a file.

searchpath attempts to locate a file by searching along the operating system path specified by the PATH=... directive in the environment. The complete path-name string is stored in path. First, searchpath searches for the file in the current directory of the current drive. If the file is not found there, the PATH environment variable will be fetched and each directory in the path will be searched in turn until the file is found or the path is exhausted. If the file is located, the string with the full path name will be copied to path. This string can be used in a call to access the file (for example, with fopen).

searchpath returns TRUE on success, otherwise it returns FALSE.

9.3.5.214 Function SearchReplace

Declaration:

unsigned long SearchReplace(char text[], char new_text[], int in_block=0, int replace_all=0);

Description

Searches for text and replaces. The replace_all parameter specifies, whether the search is continued after the first occurence of text is replaced. If replace_all is 0, then only the first occurence will be replaced, otherwise, all occurences.

SearchReplace returns the number of replaces

9.3.5.215 Function SelectBrush

 Declaration:

void SelectBrush(unsigned long handle, unsigned long color);

Description

Selects a brush for drawing with the specified color. By default, a brush with the standard color is selected, when the window opens. Brushes are used for drawing painted figures such as circles, rectangles, etc.

9.3.5.216 Function SelectFont

Declaration:

void SelectFont(unsigned long handle, char name[], int height);

Description

Selects a font for text output. As opposed to the SetWindowFont function, this font can be proportional. It is used for displaying text with the DisplayTextF function anywhere in the window.

name is the line with the font name; height specifies the font height.

9.3.5.217 Function SelectPen

Declaration:

void SelectPen(unsigned long handle, unsigned long color, int width=1, int style=PS_SOLID);

Description

Selects a pen for drawing with the specified parameters. The standard pen (a solid line with the width of 1) and the standard color are selected by default, when the window opens. Pens are used for drawing lines, circumferences, etc.

Parameters:

color

width - the pen width; certain videoadapters face problems while drawing lines

    with a width greater than 1;

style - the line type:

    PS_SOLID   - solid

    PS_DOT   - dotted

    PS_DASHDOT   - dash-and-dot

    PS_DASHDOTDOT - dash-and-dot-and-dot

9.3.5.218 Function SetBkColor

Declaration:

void SetBkColor(unsigned long handle, unsigned long color);

Description

Sets up the window background color.

9.3.5.219 Function SetBkMode

Declaration:

void SetBkMode(unsigned long handle, int mode);

Description

Sets the text display mode for the window. For the mode parameter, the system.h system header file contains two constants: OPAQUE and TRANSPARENT. When text is displayed (see DisplayText, DisplayTextF) and the display mode is set to OPAQUE, then the rectangle with text will be first filled with the background color. In the TRANSPARENT mode, the text overlaps the

previous output.

9.3.5.220 Function SetBreak

Declaration:

void SetBreak(unsigned long addr);

Description

Sets up the code breakpoint at the specified address

9.3.5.221 Function SetBreaksRange

Declaration:

void SetBreaksRange(unsigned long start_addr, unsigned long end_addr);

Description

Sets up the code breakpoints in the range from start_addr to end_addr inclusive.

9.3.5.222 Function SetByte

Declaration:

void SetByte(unsigned long addr, int addr_space, unsigned int value);

Description

Writes value (byte) to the specified address in the specified memory area (the parameter). Constants with the AS_ prefix for microcontroller memory areas (address spaces) are defined in the system.h header file.

Example

    SetByte(0x2000, AS_CODE, 0xFF);

9.3.5.223 Function SetCaption

Declaration:

void SetCaption(unsigned long handle, int set);

Description

Removes or restores the window's caption bar in accordance with the value of set.

9.3.5.224 Function setdisk

Declaration:

int setdisk(int drive);

Description

Sets the current drive number.

setdisk sets the current drive to the one associated with drive: 0 for A, 1 for B, 2 for C, and so on.

9.3.5.225 Function SetDword

Declaration:

void SetDword(unsigned long addr, int addr_space, unsigned long value);

Description

Writes a double word (32 bits) to the specified address in the specified memory area (the addr_space parameter). Constants with the AS_ prefix for microcontroller memory areas (address spaces) are defined in the system.h header file.

Example

SetDword(0x2000, AS_CODE, 0x12345678);

9.3.5.226 Function SetFileName

Declaration:

void SetFileName(char name[]);

Description

Sets the file name for the current Source

9.3.5.227 Function setftime

Declaration:

int setftime(long handle, unsigned long time);

Description

Sets the file date and time.

setftime sets the file date and time of the disk file associated with the open handle to the date and time provided in the time parameter. The file must be open for writing; the EACCES error will occur if the file is open for read-only access. The file must not be written to after the setftime call or the changed information will be lost. setftime requires the file to be open for writing; an EACCES error will occur if the file is open for read-only access. The time parameter has the following layout:

Bits   Value

0...4   two seconds

5...10   minutes

11...15   hours

16...20   days

21...24   months

25...31   year - 1980

Returned Value

setftime returns 0 on success. In the event of an error, -1 is returned and the errno global variable is set to one of the following values:

EACCES   Permission denied

EBADF   Bad file number

EINVFNC   Invalid function number

9.3.5.228 Function SetMark

Declaration:

void SetMark(int number);

Description

Sets the bookmark with the numberNumber shall be within 1...10.

9.3.5.229 Function setmem

Declaration:

void setmem(void s[], unsigned int length, char value, int index=0);

Description

In the object specified by ssetmemvalue (and converted into the unsigned char).

Returned value

None.

9.3.5.230 Function SetMemory

Declaration:

void SetMemory(void src[], int n, unsigned long addr, int addr_space);

Description

Writes n-byte memory block to the specified address in the specified memory area (the addr_space parameter) from the src array. Constants with the prefix for microcontroller memory areas (address spaces) are defined in the system.h header file.

Example

        SetMemory("12345678", 8, 0x20, AS_DATA);

9.3.5.231 Function setmode

Declaration:

int setmode(long handle, int amode);

Sets mode of an open file.

setmode sets the mode of the opened file associated with handle to either binary or text. The amode argument must have the value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in system.h).

Returned Value

setmode returns the previous translation mode, if successful. On error, it returns -1 and sets the errno global variable to

EINVAL   Invalid argument

9.3.5.232 Function SetPixel

Declaration:

void SetPixel(unsigned long handle, int x, int y, unsigned long color);

Draws one point of the specified color in the specified place.

9.3.5.233 Function SetTextColor

Declaration:

void SetTextColor(unsigned long handle, unsigned long color);

Description

Sets up color of the text printed out by the wprintf function, or displayed by the DisplayText and DisplayTextF functions. The color you set remains unchanged until SetTextColor is called for the next time. The standard color is used by default.

Example

 unsigned long handle = OpenStreamWindow("Serial port");

 SetTextColor(handle, 0xFF);

 wprintf(handle, "Will be written in red color\n");

 SetTextColor(handle, 0xFF00);

 wprintf(handle, "Will be written in green color");

9.3.5.234 Function SetToolbar

Declaration:

void SetToolbar(unsigned long handle, int set);

Description

Removes or restores the window's toolbar in accordance with the value of set.

9.3.5.235 Function SetUpdateMode

Declaration:

void SetUpdateMode(unsigned long handle, int update);

Description

Sets up the window update mode. By default, all graphical output is immediately displayed in the

window. The SetUpdateMode function sets up a different update mode, when graphical output is cached in the memory and drawing is carried out by calling the UpdateWindow function. Using this, the drawing is performed faster. The update parameter can assume two values:

UM_IMMEDIATE   - immediate drawing;

UM_ONREQUEST   - drawing by calling the UpdateWindow function.

Example

ulong handle = OpenUserWindow("Test");

MoveTo(handle, 20, 20);

LineTo(handle, 40, 40);

LineTo(handle, 45, 45);

UpdateWindow(handle);

9.3.5.236 Function SetWindowFont

Declaration:

void SetWindowFont(unsigned long handle, char font_name[], int height);

Description

Sets up the font for the specified window.

The handle parameter is the window identifier produced by the call of the , and FindWindow functions.

font_name is the string with the font name; is the font height.

Only monospaced fonts, such as Courier or Fixedsys, shall be used.

You can draw with any font, in the User window. To select the font, use the SelectFont function.

Example

unsigned long handle = OpenWindow(WIN_DUMP);

SetWindowFont(handle, "Courier New", 12);

9.3.5.237 Function SetWindowSize

Declaration:

void SetWindowSize(unsigned long handle, int w, int h);

Description

Sets up the new size for the specified window. The handle parameter is the window identifier produced by the call of the OpenWindow, and FindWindow functions. w and are the new width and height of the window (in pixels). The size also includes the non-user area of the window (the frame and title).

The position of the window upper left corner does not change.

9.3.5.238 Function SetWindowSizeT

Declaration:

void SetWindowSizeT(unsigned long handle, int w, int h);

Description

Sets up the new size for the specified window in text units. Since almost all windows of CPI2-B1 use the pseudotext mode, it can be useful to specify the window size only in terms of text.

The handle parameter is the window identifier produced by the call of the OpenWindow, and FindWindow functions. w is the number of text characters in the line; h is the number of lines in the window.

9.3.5.239 Function SetWord

Declaration:

void SetWord(unsigned long addr, int addr_space, unsigned int value);

Description

Writes a word (16 bits) to the specified address in the specified memory area (the addr_space parameter). Constants with the AS_ prefix for microcontroller memory areas (address spaces) are defined in the system.h header file.

Example

SetWord(0x2000, AS_CODE, 0xFFFF);

9.3.5.240 Function sin

Declaration:

```
float sin(float x);
```

Description

The sin function calculates the sine of the floating-point number x.

Returned value

The sin function returns the sine of x.

9.3.5.241 Function sprintf

Declaration:

void sprintf(char dest[], unsigned char format[], ...);

Description

The sprintf function displays the values of transferred parameters in the dest line in accordance with the format line.

Note. Your arguments passed to this function shall match the format line. In case of mismatch, the CPI2-B1 program may crash, because it cannot check the correspondence between the format string and parameters passed.

Returned value

None.

9.3.5.242 Function sqrt

Declaration:

`float sqrt(float x);`

Description

The sqrt function calculates the square root of number x.

Returned value

The sqrt function returns the square root of x. The returned value for negative arguments is 0.

9.3.5.243 Function srand

Declaration:

void srand(unsigned int seed);

Description

Initializes a random number generator by a specified number.

9.3.5.244 Function sscanf

Declaration:

int sscanf(char buf[], char format[], ...);

Description

The sscanf function parses the buf string in accordance with the format line.

sscanf scans a series of input fields one character at a time reading from a stream. After that, each field is formatted in accordance with a format specifier passed to sscanf in the format string pointed to by format. Finally, sscanf stores the formatted input at the address passed to it as the argument following the format. The number of format specifiers and addresses must be the same as the number of input fields.

Notes
1.   Your arguments passed to this function shall match the format line. In case of mismatch, the CPI2-B1 program may crash, because it cannot check the correspondence between the format string and parameters passed. For details on format specifiers, see the scanf Format Specifiers.
2. All arguments for this function shall be arrays, because only the array parameters are passed by address to functions. Also, see example for scanf.

sscanf can stop scanning a particular field before it reaches the normal end-of-field character

(whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion on possible causes.

Returned Value

9.3.5.245 Function Step

Declaration:

```
void Step();
```

Description

Executes one machine instruction (the low-level step mode).

Note. The screen is not updated automatically after this function is called. To organize the automatic update, use the RedrawScreen function at the appropriate moment.

9.3.5.246 Function Stop

Declaration:

void Stop();

Description

Stops the program under execution.

9.3.5.247 Function stpcpy

Declaration:

int stpcpy(char dest[], char src[], int dest_index=0, int src_index=0);

Description

The stpcpysrc line to the dest line and attaches the zero character.

Returned value

The stpcpy function returns the number of the last byte copied to dest

9.3.5.248 Function strcat

Declaration:

void strcat(char dest[], char src[], int dest_index=0, int src_index=0);

Description

The strcat function joins the line to the dest line and ends the dest line with zero.

Returned value

None.

9.3.5.249 Function strchr

Declaration:

int strchr(char s[], int c, int index=0);

Description

The strchr function searches the first entry of character cs. The zero characters also participate in the search.

Returned function

The strchr function returns the number of the found character to s and returns -1, if there is no such character there.

9.3.5.250 Function strcmp

Declaration:

int strcmp(char s1[], char s2[], int s1_index=0, int s2_index=0);

Description

The strcmps1 and s2 letter-by-letter and returns the result of the search.

Returned value

The function returns the following values of comparison result:

Value	Meaning
< 0	s1 is less than s2
= 0	s1 is equal to s2
> 0	s1 is greater than s2

9.3.5.251 Function strcmpi

Declaration:

int strcmpi(char s1[], char s2[], int s1_index=0, int s2_index=0);

The same as stricmp

9.3.5.252 Function strcpy

Declaration:

void strcpy(char dest[], char src[], int dest_index=0, int src_index=0);

Description

The strcpy function copies the contents of line src to line dest and attaches the zero character.

Returned value

None.

9.3.5.253 Function strcspn

Declaration:

int strcspn(char s1[], char s2[], int s1_index=0, int s2_index=0);

Description

The function searches any character from line s2 to line s1.

Returned value

The strcspn function returns the number of the first character in line s1 equal to any character from line s2. Zero will be returned, if the first character in line s1 is equal to any character from line s2. If there are no such characters there, then the length of line s1 will be returned (the zero character is not taken into account).

9.3.5.254 Function stricmp

Declaration:

int stricmp(char s1[], char s2[], int s1_index=0, int s2_index=0);

Description

The stricmp function compares lines s1 and s2 letter-by-letter regardless of the character case and returns the result of the search.

Returned value

The stricmp function returns the following comparison results:

Value   Meaning

< 0   s1 is less than s2

= 0   s1 is equal to s2

> 0   s1 is greater than s2

9.3.5.255 Function strlen

Declaration:

int strlen(char s[], int index=0);

Description

The strlen function calculates the length of line src in bytes. The last zero character is not counted.

Returned value

The strlen function returns the length of line src.

9.3.5.256 Function strlwr

Declaration:

void strlwr(char s[], int index=0);

Description

The strlwr function converts line s to the lower case.

Returned value

None.

9.3.5.257 Function strncat

Declaration:

void strncat(char dest[], char src[], int n, int dest_index=0, int src_index=0);

Description

The strncat function attaches the maximum of n characters from line scr to line dest and ends dest with the zero character. If there are less than n characters in line , then the whole line src together with the zero character will be copied.

Returned value

None.

9.3.5.258 Function strncmp

Declaration:

```
int strcmp(char s1[], char s2[], int s1_index=0, int s2_index=0);
```

Description

The strcmp function compares lines s1 and s2 letter-by-letter and returns the result of the search.

Returned value

The strcmp function returns the following values of comparison result:

```
Value          Meaning
-------------------------------
< 0            s1 is less than s2
= 0            s1 is equal to s2
> 0            s1 is greater than s2
```

9.3.5.259 Function strncmpi

 Declaration:

int strncmpi(char dest[], char src[], int n, int dest_index=0, int src_index=0);

Description

The strncmpi function compares the first n bytes of lines s1 and s2 letter-by-letter regardless of the character case and returns the comparison result.

Returned value

The strncmpi function returns the following values of the lines s1 and s2

< 0   s1 is less than s2

= 0   s1 is equal to s2

> 0   s1 is greater than s2

9.3.5.260 Function strncpy

Declaration:

void strncpy(char dest[], char src[], int n, int dest_index=0, int src_index=0);

Description

The strncpy function copies the maximum of n characters from line scrn characters in line src, then the zero characters will be added to line dest to extend it up to the size of n.

Returned value

None.

9.3.5.261 Function strnicmp

Declaration:

int strnicmp(char dest[], char src[], int n, int dest_index=0, int src_index=0);

The same as strncmpi.

9.3.5.262 Function strnset

Declaration:

void strnset(char s[], int c, int n, int index=0);

Description

The strnset function sets the maximum of n characters from line s to zero.

Returned value

None.

9.3.5.263 Function strpbrk

Declaration:

int strpbrk(char s1[], char s2[], int s1_index=0, int s2_index=0);

Description

Function strpbrk searches for the first occurrence of any character from line s2 in line s1. The zero character is not the search element.

Returned value

The strpbrk function returns the number of the found character in line s1. If line s1 does not contain any characters from line s2, then -1 will be returned.

9.3.5.264 Function strrchr

Declaration:

```
int strchr(char s[], int c, int index=0);
```

Description

The strchr function searches the first entry of character c in line s. The zero characters also participate in the search.

Returned function

The strchr function returns the number of the found character to s and returns -1, if there is no such character there.

9.3.5.265 Function strrev

Declaration:

void strrev(char s[], int index=0);

Description

The strrev function reverses the byte order in line s. For example, if we write:

> char s[] = "1234"; strrev(s);

then the lines will contain "4321".

Returned value

9.3.5.266 Function strset

Declaration:

void strset(char s[], int c, int index=0);

Description

The strset function sets all characters in line s to the value of c.

Returned value

None.

9.3.5.267 Function strspn

Declaration:

int strspn(char s1[], char s2[], int s1_index=0, int s2_index=0);

The strspn function searches in the line s21 for symbols, which are absent in line s2.

Returned value

The strspn function returns the number of the first character in line s1, which is known to be absent in line s2. If there are no such symbols in line s1, then the length of line s1 will be returned (the zero character is not taken into account).

9.3.5.268 Function strstr

 Declaration:

int strstr(char s1[], char s2[], int s1_index=0, int s2_index=0);

Description

The strstr function searches for the first occurrence of the string from s2 in line s1 (the zero character is not taken into account).

Returned value

The strstr function returns the number of the first byte of the string from s2, or returns -1, if there is no such string there.

9.3.5.269 Function strtol

Declaration:

long strtol(char s[], int endptr[], int radix, int index=0);

Converts an ASCII-string (the s parameter; index specifies shift in the line) into a long number. The radix parameter is the radix used for conversion (2...36).

String s may include the following components:

[ws] [sn] [0] [x] [ddd]

[ws] - Optional spaces or tabulation symbols

[sn] - Optional sign (+ or -)

[0] - Optional zero (0)

[x] - Optional x or X

 - Optional digits

endptr array contains that character number).

If radix is equal to 0, then radix will be selected by the first few characters of the s string:

First character   Second character String interpretation

   0   1 - 7   Octal

0	x or X	Hexadecimal
1 - 9		Decimal

Returned value

The converted long integer number.

9.3.5.270 Function strtoul

Declaration:

unsigned long strtoul(char s[], int endptr[], int radix, int index=0);

Description

The strtoul function is the same as strtol, except that it returns the unsigned long integer.

9.3.5.271 Function strupr

Declaration:

void strupr(char s[], int index=0);

Description

The strupr function converts line s to the upper case.

Returned value

None.

9.3.5.272 Function tan

Declaration:

```
float tan(float x);
```

Description

The tan function calculates the tangent of the floating-point number x.

Returned value

The tan function returns the tangent of argument x.

9.3.5.273 Function tanh

Declaration:

```
float tanh(float x);
```

Description

The tanh function calculates the hyperbolic tangent of the floating-point number x. The argument should range from -88.72280 to 88.72280.

Returned function

The tanh function returns the hyperbolic tangent of argument x.

9.3.5.274 Function tell

Declaration:

long tell(long handle);

Description

Gets the current position of the file pointer.

tell gets the current position of the file pointer associated with handle and expresses it as the number of bytes from the beginning of the file.

Returned Value

tell returns the current file pointer position. Returned -1 (long) indicates an error, and the errno global variable is set to

9.3.5.275 Function TerminateAllScripts

Declaration:

void TerminateAllScripts();

Description

Stops execution of all script files (except the script called by this function).

9.3.5.276 Function TerminateScript

Declaration:

void TerminateScript(char file_name[]);

Description

Stops execution of the specified script file and unloads it from the memory, if possible. The file name parameter is the script file name without path and extension.

9.3.5.277 Function Text

Declaration:

void Text(char text[]);

Description

text text from the cursor position, as if it were typed from the keyboard.

9.3.5.278 Function toascii

Declaration:

int toascii(unsigned char c);

Description

The toascii function cuts off the high bit of parameter c.

Returned value

The toascii function returns the value of c cut down to 7 bits

9.3.5.279 Function Tof

Declaration:

void Tof();

Description

Move the cursor to the top of the file (position (1:1)).

9.3.5.280 Function tolower

Declaration:

int tolower(unsigned char c);

Description

tolower function converts character c to the lower case. If c is not an alphabetic character, then it will not be converted.

Returned value

The tolower function returns character c in the lower case.

9.3.5.281 Function toupper

Declaration:

int toupper(unsigned char c);

Description

The toupper function converts character c to the upper case. If c is not an alphabetic character, then it will not be converted.

Returned value

The toupper function returns character c in the upper case.

9.3.5.282 Function ultoa

Declaration:

void ultoa(unsigned long value, char string[], int radix);

Description Converts an unsigned long integer (value) into the character string (string). The radix parameter is the radix used for conversion (2...36).

9.3.5.283 Function unlink

Declaration:

int unlink(char file_name[]);

Description

Deletes a file.

unlink deletes the file specified by file_name. Any drive, path, and file name can be used as the filename. Wildcards are not allowed. This call cannot delete read-only files.

Note. If your file is open, be sure to close it before unlinking it.

Returned Value

On success, unlink returns 0. On error, it returns -1 and sets the errno global variable to one of the following values:

EACCES   Permission denied

ENOENT   Path or file name not found

9.3.5.284 Function unlock

Declaration:

int unlock(long handle, long offset, long length);

Description

Releases file-sharing locks.

unlock provides interface to the operating system file-sharing mechanism. unlock removes a lock previously placed with a call to lock. To avoid error, all locks must be removed before closing a file. The program must release all locks before completing.

Returned Value

On success, unlock returns 0. On error, it returns -1.

9.3.5.285 Function Up

Declaration:

void Up(int count=1);

Description

Move the cursor count lines up. The same result can be achieved by decrementing the CurLine built-in variable.

9.3.5.286 Function UpdateWindow

Declaration:

```
void UpdateWindow(unsigned long handle);
```

Description

Draws an image in the specified window. The image is cached in the memory during graphical output function calls. Calling this function makes sense only when selecting the mode of drawing with the SetUpdateMode function call with the UM_ONREQUEST parameter

9.3.5.287 Function Wait

Declaration:

```
void Wait(unsigned long microseconds);
```

Description

Suspends execution of the script file until the specified interval of the time is up.

The <%CM%> cannot trace extremely short time intervals, because some time is needed for data transmission through the serial channel.

Example:

```
while (1)      // endless cycle
{
   Wait(100);  // to wait for 100 microseconds.
   $P1 ^= 1;   // to invert bit 0 in port P1
}
 }
```

9.3.5.288 Function WaitExprChange

Declaration:

void WaitExprChange(char str[]);

Description

Suspends execution of the script file until the expression specified in the str line changes its value.

The peculiarities of this function for the CPI2-B1 are the same as for .

Note that you should not precede the variable names with '$' sign in the expression string.

Example:

while (1)   // the endless cycle

{

 WaitExprChange("P1 & 2"); // to wait until value of bit 1

                 // of port P1 changes

 P2 |= P1 & 2;   // to execute certain action

}

9.3.5.289 Function WaitExprTrue

Declaration:

```
void WaitExprTrue(char str[]);
```

Description

Suspends execution of the script file until the expression specified in the **str** line becomes True as the result of executing.

The expression operands should be available in the continuous emulation mode, otherwise the expression is always False.

An operand value poll is executed within the specified time interval. Therefore, the expression should remain True during this interval, otherwise the programmer cannot trace the moment, when the expression becomes True.

Note. You should not precede the variable names with '$' sign in the expression string.

Example:

```
while (1)     // the endless cycle
{
   WaitExprTrue("Counter > 200");           // to wait for the condition to
become True
   Stop();                                  // to stop the program
   printf("Counter overflow at %04X", $PC); // to display the message
}
```

9.3.5.290 Function WaitGetMessage

Declaration:

void WaitGetMessage(int id);

Description

WaitSendMessage

9.3.5.291 Function WaitMemoryAccess

Declaration:

```
void WaitMemoryAccess(unsigned long addr, int addr_space, int num_bytes, int
flags);
```

Description

Suspends execution of the script file until the processor (the program being executed) accesses the specified memory area. Parameters:

```
addr        - the memory area address.
addr_space  - the address space. Constants with prefix AS_
              are given in the system.h file.
num_bytes   - the amount of bytes in the memory area.
flags       - the flags that define the type of memory access:
```

```
                        MA_READ – reading, MA_WRITE – writing,
                   MA_READ | MA_WRITE – both reading and writing.
```

This function does not work in the emulators.

After return from the function, the built-in variables contain information on the latest traced memory access:

LastMemAccAddr   the memory address
LastMemAccAddrSpace   the type of address space
LastMemAccLen   the amount of bytes
LastMemAccType   the type of access (MA_READ, MA_WRITE).

Example:

```
while (1)      // endless cycle
{
   WaitMemoryAccess(0x80, AS_DATA, 1, MA_WRITE);
               // to wait for write to the data memory cell with the address of
0x80 (bytes).
   $P1 ^= 1;   // to invert bit 0 in port P1
}
```

9.3.5.292 Function WaitSendMessage

Declaration:

void WaitSendMessage(int id, unsigned int int_data, unsigned long long_data);

Description

The WaitSendMessage and WaitGetMessage functions provide a mechanism for message exchange between two copies of the CPI2-B1 program (or other Phyton products) running simultaneously. These functions are used mostly for simulators and allow simulation of multi-processor systems that exchange data with each other.

To simulate, say, a two programmers system, you should launch two copies CPI2-B1 and set up the exchange of data between them. You can start the second copy of CPI2-B1 by copying the UprogNT2.EXE file to a file with another name and then starting it.

The WaitSendMessage function "sends a message" to another copy of CPI2-B1 and waits until the message is "delivered", i.e. the receiver copy of CPI2-B1 calls the WaitGetMessage function. If the receiver has already called WaitGetMessage and is waiting for an incoming message, the WaitSendMessage function returns immediately, otherwise it will return, when a period of model time is passed. The model time flows, when the simulated program runs.

When calling WaitSendMessage and WaitGetMessage, you supply the id parameter that identifies the message. The message will be delivered to the copy of CPI2-B1 that is waiting for message with the same id.

The int_data and long_data parameters are the user data. You may set these parameters to any values you wish. When the receiver's WaitGetMessage returns the control, the transmitter's int_data value is copied to the receiver's LastMessageInt built-in variable and long_data is copied to LastMessageLong.

Note that CPI2-B1 uses its own internal means for message exchange, not the message

mechanism of Windows.

Example

#define SecondCopyMsg 0

#define InitExchange 0

#define InitExchangeOk 0

 Run(); // start model time

 WaitSendMessage(SecondCopyMsg, InitExchange, 0);

 WaitGetMessage(SecondCopyMsg);

 if (LastMessageInt != InitExchangeOk)

 {

   printf("Exchange failed");

   return;

 }

9.3.5.293 Function WaitStop

Declaration:

```
void WaitStop();
```

Description

Suspends execution of the script file until the program stops. The program can be stopped either by a breakpoint or manually.

9.3.5.294 Function WaitWindowEvent

Declaration:

void WaitWindowEvent(unsigned long handle);

Description

Allows to organize interaction between user and the User window and the _____ / _____. The function waits for an event associated with the specified window and returns control to the script file, when the event occurs. The function locates type of the occurred event and places relevant data into the internal variables accessible with the following functions:

LastEvent

LastEventInt{1...4}

Example

ulong handle = OpenUserWindow("Interactive Window");

while (1)

{

```
WaitWindowEvent(handle);

switch (LastEvent(handle))

{

  case WE_CLOSE:       return;          // window is closed, script file is being completed
  case WE_REDRAW:      Redraw(handle);  // to call our function Redraw,


  case WE_MOUSEBUTTON:  Change(handle);  // to call our function Change,

                break;           // that responds to the clicked

                        // mouse button



}
```

9.3.5.295 Function wgetchar

Declaration:

void wgetchar(unsigned long handle);

Description

Waits for pressing an alphanumeric key on the keyboard, when the specified window has input focus, that is, is active. The pressed key code can be obtained with the LastChar function.

The entered character is automatically displayed in the window.

Example

unsigned long handle = OpenStreamWindow("Serial port");

wprintf(handle, "Press \"E\" for exit");

wgetchar(handle);

if (toupper(LastChar(handle)) == 'E') return

9.3.5.296 Function wgethex

Declaration:

void wgethex(unsigned long handle);

Description

Waits for two hexadecimal digits (a byte value) to be entered from the keyboard. The entered number can be obtained with the LastChar function.

The entered characters are automatically displayed in the window. The Enter key moves the window cursor to the beginning of the new line.

9.3.5.297 Function wgetstring

Declaration:

void wgetstring(unsigned long handle);

Description

Waits until the character string is ended by pressing the Enter key. The entered string can be obtained with the LastString function.

The entered characters are automatically displayed in the window.

9.3.5.298 Function WindowHotkey

Declaration:

void WindowHotkey(unsigned long handle, int key);

Description

Sends the local menu command corresponding to the hot key (parameter key) to the specified window. The local window menu lists the hot keys. key is the ASCII value of the key without indicating Ctrl: for example, to imitate pressing Ctrl+T in the window, the key parameter shall be equal to 'T'.

Example

 unsigned long handle = OpenWindow(WIN_WATCHES);

 WindowHotkey(handle, 'A');   // imitates pressing Ctrl+A

9.3.5.299 Function WordLeft

Declaration:

void WordLeft();

Description

Moves the cursor to the next word (on the right).

9.3.5.300 Function WordRight

Declaration:

void WordRight();

Description

Moves the cursor to the previous word (on the left).

9.3.5.301 Function wprintf

Declaration:

void wprintf(unsigned long handle, char format[], ...);

Displays the values of transferred parameters in the window in accordance with the format line.

Attention! You are responsible for matching the arguments transferred to wprintf function into the line format. A mismatch may bring CPI2-B1 to failure.

Example

 unsigned long handle = OpenStreamWindow("Serial port");

 wprintf(handle, "SP = %04X", $SP\n");

9.3.5.302 Function write

Declaration:

int write(long handle, void buf[], int len);

Description

Writes to a file.

write writes the buffer of data to the file or device specified by handle. The handle file handle is obtained from the creatopen, dup, or dup2 call.

This function attempts to write bytes from the buffer pointed to by buf to the file associated with handle. Except for the case, when write is used to write to a text file, the amount of bytes written to the file will be no more than the amount requested. On text files, when write sees a linefeed (LF) character, it outputs a CR/LF pair.

If the amount of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disks or disk files, the writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device.

Returned Value

write returns the number of bytes written. A write to a text file does not count the generated carriage returns. In case of error, write returns -1 and sets the errno global variable to one of the following values:

EACCES   Permission denied

EBADF   Bad file number

9.3.5.303 lock

Declaration:

int lock(long handle, long offset, long length);

Description

Sets file-sharing locks. lock provides interface to the operating system file-sharing mechanism. The lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into the locked region will retry the operation three times. If all three retries fail, then the call will fail with error.

Returned Value

lock returns 0 on success. On error, lock returns -1 and sets the errno global variable to

EACCES   Locking violation

9.3.5.304 Variable _fmode

Declaration:

extern int _fmode;

This is the file operation mode (text or binary).

9.3.5.305 Variable ApplName

Declaration:

extern char ApplName[];

This is the program name, i.e. the string of "CM-ARM".

Available only for reading.

9.3.5.306 Variable BlockCol1

Declaration:

int BlockCol1;

This is the number of the left column of block in the current window. BlockCol1 is zero for the line blocks. If no block is marked, BlockCol1 will also be zero.

Also, see Text Editor Functions.

9.3.5.307 Variable BlockCol2

Declaration:

int BlockCol2;

This is the number of the right column of block in the current Source . BlockCol2 is zero for the line blocks. If no block is marked, BlockCol2 will also be zero.

Also, see Text Editor Functions

9.3.5.308 Variable BlockLine1

Declaration:

int BlockLine1;

This is the number of the upper line of block in the current Source.

If no block is marked, BlockCol2 will be zero.

Also, see Text Editor Functions

9.3.5.309 Variable BlockLine2

Declaration:

int BlockLine2;

This is the number of the lower line of block in the current Source

If no block is marked, BlockCol2 will be zero.

Also, see Text Editor Functions

9.3.5.310 Variable BlockStatus

Declaration:

int BlockStatus;

This is the type of block in the current Source The system.h system header file contains definitions of constants:

EB_NONE   - no block

EB_LINE   - line block

EB_VERT   - vertical block

EB_STREAM - stream block

Also, see Text Editor Functions.

9.3.5.311 Variable CaseSensitive

Declaration:

int CaseSensitive;

Source

Also, see Text Editor Functions.

9.3.5.312 Variable CurCol

Declaration:

int CurCol;

This is the number of the current column (the column the cursor is in) in the current Source. Columns are numbered with 1.

If the cursor is beyond the line end, then CurCol will contain 0.

Assigning a value to CurCol changes the cursor position. Also, see functions GotoXY, Up, Down, LeftRight, Tof, EofEol.

9.3.5.313 Variable CurLine

Declaration:

int CurLine;

Source. Lines are numbered with 1.

Assigning a value to CurLine changes the cursor position. Also, see functions GotoXY, Up, Down, , Right, Tof, Eof, Eol.

9.3.5.314 Variable DesktopName

Declaration:

extern char DesktopName[];

This string is the name of the current screen configuration file (see Configuration Files).

Available only for reading.

9.3.5.315 Variable errno

Declaration:

extern int errno;

This is the error code set up by some built-in functions such as read.

9.3.5.316 Variable InsertMode

Declaration:

int InsertMode;

This is the insert mode for the current Source . Assigning a value to InsertMode toggles the insert mode for the window.

Also, see Text Editor Functions.

9.3.5.317 Variable LastFoundString

Declaration:

char LastFoundString[];

This is the string with the text that was last found in the current Source . Because the search argument may contain regular expretion, the string found may not be the same as the search argument.

Also, see Text Editor Functions.

9.3.5.318 Variable LastMemAccAddr

Declaration:

extern unsigned long LastMemAccAddr;

This is the microcontroller memory address accessed at the last return from the WaitMemoryAccess function.

9.3.5.319 Variable LastMemAccAddrSpace

Declaration:

extern unsigned int LastMemAccAddrSpace;

This is the type of microcontroller address space accessed at the last return from the WaitMemoryAccess function

9.3.5.320 Variable LastMemAccLen

Declaration:

extern int LastMemAccLen;

WaitMemoryAccess function.

9.3.5.321 Variable LastMemAccType

Declaration:

extern int LastMemAccType;

This is the microcontroller memory access type that caused a return from the WaitMemoryAccess function. For example, MA_READ, MA_WRITE or a combination of them.

9.3.5.322 Variable LastMessageInt

Declaration:

unsigned int LastMessageInt;

LastMessageInt keeps the 16-bit parameter received by the WaitGetMessage function.

9.3.5.323 Variable LastMessageLong

Declaration:

unsigned long LastMessageLong;

LastMessageInt keeps the 32-bit parameter received by the WaitGetMessage function.

9.3.5.324 Variable MainWindowHandle

Declaration:

extern unsigned long MainWindowHandle;

This is HWND of the main window of CPI2-B1. It is only for experienced programmers.

9.3.5.325 Variable NumWindows

Declaration:

extern int NumWindows;

This is the number of windows opened in CPI2-B1. Its value changes dynamically, as windows are opened or closed.

9.3.5.326 Variable RegularExpressions

Declaration:

int RegularExpressions;

Sets up the use of regular expretions for the operation of search in the current Source .

Also, see Text Editor Functions.

9.3.5.327 Variable SelectedString

Declaration:

extern char SelectedString[];

This is the string selected from the menu at the last call of the built-in ExecMenu function.

9.3.5.328 Variable SystemDir

Declaration:

extern char SystemDir[];

This string is the name of the directory, where the CPI2-B1 package is installed.

Available only for reading.

9.3.5.329 Variable WholeWords

Declaration:

int WholeWords;

Sets up the whole words option for the operation of search in the current **Source** window.

Also, see Text Editor Functions.

9.3.5.330 Variable WindowHandles

Declaration:

extern unsigned long WindowHandles[];

This is the listing of the CPI2-B1 window handles organized as an array of the NumWindows size. It is only for experienced programmers.

9.3.5.331 Variable WorkFieldHeight

Declaration:

extern unsigned int WorkFieldHeight;

This is the height of the CPI2-B1 window user area in pixels. It may be useful for locating windows from script files.

Available only for reading.

9.3.5.332 Variable WorkFieldWidth

Declaration:

extern unsigned int WorkFieldWidth;

This is the width of the CPI2-B1 window user area in pixels. It may be useful for locating windows from script files.

Available only for reading.

9.4   ACI Fuctions and Structures

This section contains detailed descriptions of ACI functions and structures.

9.4.1   ACI Fuctions

This sections contains alphabetical list of all ACI functions.

9.4.1.1   ACI_Launch

ACI_FUNC ACI_Launch(ACI_Launch_Params * params);

Description

This function launches the ChipProg-02 software. Optionally this ACI function can launch the programmer with a specified command line key and load the file that will configure the CPI2-B1 hardware.

Note! This ACI function must always be called before any other ACI function !

9.4.1.2   ACI_Exit

ACI_FUNC ACI_Exit();

Description

Call of this function stops the ChipProg-02 software. In most cases the programmer practically immediately stops running. Sometimes, after calling the **ACI_Exit** function, it continues working for a while to correctly complete an earlier launched process. After all, the CPI2-B1 will stop and quit itself after finding that the controlling process has ended.

It is possible, however, that the ChipProg-02 software will keep running even after the control process has completely stopped. This is an abnormal situation and, as a result, it will be impossible to re-establish communication with the programmer hardware by launching the ACI_Launch function. In this case you should manually close the ChipProg-02 program via the Windows Task Manager.

.

9.4.1.3   ACI_ErrorString

ACI_FUNC ACI_ErrorString(ACI_ErrorString_Params * params);

Description

Get the string describing the result of the last ACI function call.

All ACI functions return the ACI_ERR_xxx error code but this is may not be enough to find out the exact reason of the error. The string returned by ACI_ErrorString describes the error in detail.

9.4.1.4   ACI_LoadConfigFile

ACI_FUNC ACI_LoadConfigFile(ACI_Config_Params * params);

Description

This function loads the CPI2-B1 configuration parameters that include all the settings available via the ChipProg-02 dialogs (memory buffer configurations, programming options, test of the device insertion, etc.).

The ChipProg-02 program automatically saves some programming options and settings, including the type of selected device, the device parameters, the start and end addresses of the device being programmed, the buffer start address, and a set of the Auto Programming commands. Then it automatically restores these parameters when the user changes the device type.

See also: ACI_SetProgrammingParams, ACI_SetProgOption, ACI_GetProgrammingParams, ACI_GetProgOption, ACI_SaveConfigFile

9.4.1.5   ACI_SaveConfigFile

ACI_FUNC ACI_SaveConfigFile(ACI_Config_Params * params);

Description

This function saves the CPI2-B1 options specified in the tab Option of the Program Manager window (memory buffer configurations, programming options, test of the device insertion, etc.).

The ChipProg-02 program automatically saves some programming options and settings including a type of the selected device, the device parameters, the start and end addresses of the device being programmed, the buffer start address, and a set of the Auto Programming commands and then automatically restores these parameters when the user changes the device type.

 .   : ACI_SetProgrammingParams, ACI_SetProgOption, ACI_GetProgrammingParams, ACI_GetProgOption, ACI_LoadConfigFile

9.4.1.6   ACI_LoadProject

ACI_FUNC ACI_LoadProject(ACI_Project_Params * params);

Description

Load the project. The path to the project file is specified in the ProjectName member of the ACI_Project_Params structure. The project must be previously prepared and saved manually in the programmer shell application.

Using this function is convenient because loading a project automatically performs the following:

- The programmer shell settings are loaded;
- The device chosen in the project is loaded;
- The programming options are set to the values specified in the project;
- Files specified in the project are loaded to the buffers;
- Settings for the Checksum, SerialNumber, Shadow areas, etc. are loaded.

Loading a project with ACI_LoadProject() is the same as loading a project in the programmer shell.

9.4.1.7   ACI_SetDevice

ACI_FUNC ACI_SetDevice(ACI_Device_Params * params);

Description

This function chooses the device to be programmed. Along with the device type, the function automatically loads the device parameters, start and end addresses and the buffer start address. Also, it restores the **Auto Programming** command list if the selected device type has ever been selected earlier, but the parameters listed above were changed during the programming session.

9.4.1.8   ACI_GetDevice

ACI_FUNC ACI_GetDevice(ACI_Device_Params * params);

Description

This function gets the device's part number (name) and the name of the manufacturer of the device being programmed now (for example: AT89C51, Atmel; 28F128J3C, Numonyx, etc.).

9.4.1.9 ACI_GetLayer

ACI_FUNC ACI_GetLayer(ACI_Layer_Params * params);

Description

This function gets the parameters of a specified memory buffer and buffer's layer.

See also the ACI_Layer_Params structure description.

9.4.1.10 ACI_CreateBuffer

ACI_FUNC ACI_CreateBuffer(ACI_Buffer_Params * params);

Description

This function creates a buffer with the parameters specified by the ACI_Buffer_Params structure. The ChipProg-02 program automatically assigns the buffer #0 so it is not necessary to create this buffer by a separate command.

See also the ACI_Buffer_Params structure description.

9.4.1.11 ACI_ReallocBuffer

ACI_FUNC ACI_ReallocBuffer(ACI_Buffer_Params * params);

Description

This function changes the size of the layer #0 in the memory buffer specified in the ACI_Buffer_Params structure.

See also the ACI_Buffer_Params structure description.

9.4.1.12 ACI_ReadLayer

ACI_FUNC ACI_ReadLayer(ACI_Memory_Params * params);

Description

This function reads data from a specified memory buffer. The data size is limited by 16M Bytes.

Note! This function reads the data from the programmer's memory buffer but **does not physically**

read out the content of the selected target device. In order to physically read out the device memory content, execute the programmer command (function) **Read** by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes.

9.4.1.13 ACI_WriteLayer

ACI_FUNC ACI_WriteLayer(ACI_Memory_Params * params);

Description

This function writes data to a specified memory buffer. The data size is limited by 16M Bytes.

Note! This function writes the data to the programmer's memory buffer but **does not physically program the device**. In order to physically write data from the buffer to the device's memory, execute the programmer command (function) **Program** by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes.

9.4.1.14 ACI_FillLayer

ACI_FUNC ACI_FillLayer(ACI_Memory_Params * params);

Description

This function fills a whole active layer of a specified memory buffer with a specified data pattern. This function works much faster than the ACI_WriteLayer function which writes data to the buffer layer.

Note! This function fills the programmer's memory buffer with a specified data pattern but **does not physically write them to the device being programmed**. In order to physically write data from the buffer to the device execute the programmer command (function) **Program** by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes.

9.4.1.15 ACI_GetProgrammingParams

ACI_FUNC ACI_GetProgrammingParams(ACI_Programming_Params * params);

Description

This function gets current programming parameters specified in the tab Option of the Program Manager window (memory buffer configurations, programming options, test of the device insertion, etc.).

See the ACI_Programming_Params structure description.

9.4.1.16 ACI_SetProgrammingParams

ACI_FUNC ACI_SetProgrammingParams(ACI_Programming_Params * params);

Description

This function sets programming parameters specified in the tab Option of the Program Manager window (memory buffer configurations, programming options, test of the device insertion, etc.).

See also the ACI_Programming_Params structure description.

9.4.1.17 ACI_GetProgOption

ACI_FUNC ACI_GetProgOption(ACI_ProgOption_Params * params);

Description

This function gets current settings from the **Device and Algorithm Parameters Editor** window. As an example see this window for one of the microcontrollers below.

Note! This function **does not physically read the specified information from the device being programmed**. It reads from some virtual memory locations in the host PC's RAM, associated with physical locations in the target device's memory and registers. If the option that you would like to check is a property of the device's memory or registers, then first you have to execute the programmer command (function) **Read** in the command group **Device Parameters** by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes. Then you can read the execute the ACI_GetProgOption function.
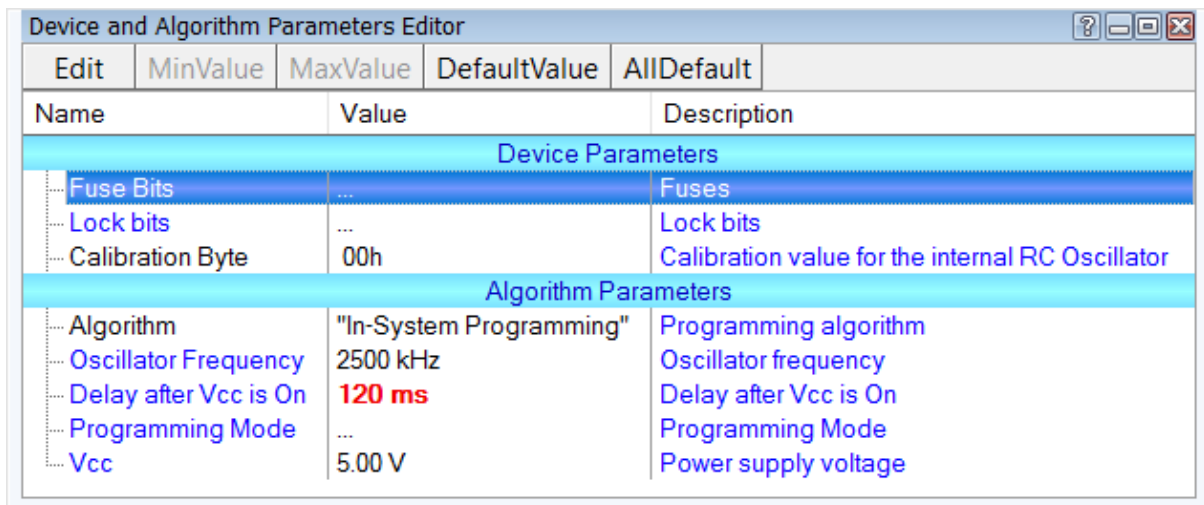
See also the ACI_ProgOption_Params structure description.

9.4.1.18 ACI_SetProgOption

ACI_FUNC ACI_SetProgOption(ACI_ProgOption_Params * params);

Description

This function sets device-specific options and parameters, which are specified in the **Device and Algorithm Parameters Editor** window. As an example see this window for one of the microcontrollers below.

> **Note!** This function **does not physically write the specified information into the device being programmed**. It actually writes to some virtual memory locations in the host PC's RAM, associated with physical locations in the target device's memory and registers. In order to complete programming the device parameters and to physically program them into the device's memory you should execute an appropriate **Program** command (function) in the command group **Device Parameters**, by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes.

See also the ACI_ProgOption_Params structure description.

9.4.1.19 ACI_AllProgOptionsDefault

ACI_FUNC ACI_AllProgOptionsDefault();

Description

This function sets default device-specific options and parameters specified in the **Device and Algorithm Parameters Editor** window. These default parameter sets vary. They are defined by the device manufacturers in the device data sheets.

> **Note!** This function **does not physically restore the default settings into the device being programmed**. It actually writes to some virtual memory locations in the host PC's RAM, associated with physical locations in the target device's memory and registers. In order to complete programming the device parameters and to physically fix them in the device's memory you should execute an appropriate **Program** command (function) in the **Device Parameters** command group by means of the ACI_ExecFunction or ACI_StartFunction with appropriate attributes.

9.4.1.20 ACI_ExecFunction

ACI_FUNC ACI_ExecFunction(ACI_Function_Params * params);

Description

This function launches one of the programming operation (**Read**, **Erase**, **Verify**, etc.) specified by the ACI_Function_Params. During execution the **ACI_ExecFunction** does not allow calling any other ACI function until the programming operation, initiated by the **ACI_ExecFunction** function, completes the job. The ACI_ExecFunction from the ACI_StartFunction that returns control immediately after it was called.

9.4.1.21 ACI_StartFunction

ACI_FUNC ACI_StartFunction(ACI_Function_Params * params);

Description

This function launches one of the programming operation (**Read**, **Erase**, **Verify**, etc.) specified by the ACI_Function_Params and immediately returns control to the external application no matter whether the programming operation, initiated by the **ACI_StartFunction**, has or has not completed. The **ACI_StartFunction** is different from the **ACI_ExecFunction**. It is possible to check if the operation has completed by the **ACI_GetStatus** function call. This allows monitoring the execution of programming operations if they last for a long time.

9.4.1.22 ACI_GangStart

ACI_FUNC ACI_GangStart(ACI_GangStart_Params * params);

Description

This function is used to control multiple device programmers only when the ChipProg-02 program was launched from the command line with the **/gang** key to drive a CPI2-B1 gang programmer or a cluster of multiple programmers connected to one PC! See also the ACI_Launch function. For controlling a single CPI2-B1 device programmer use ACI_StartFunction or ACI_ExecFunction.

The **ACI_GangStart** function launches Auto Programming on multiple CPI2-B1 device programmers for

the programming socket specified in the **SiteNumber** parameter of the ACI_PStatus_Params structure. The function returns control immediately. In order to detect the ending time of the **ACI_GangStart** execution, use the ACI_GetStatus function.

9.4.1.23 ACI_GetStatus

ACI_FUNC ACI_GetStatus(ACI_PStatus_Params * params);

Description

This function gets the programmer status that includes:

1) The status of the programming operation initiated by the ACI_StartFunction call (whether it has completed or it is still in progress);
2) The device insertion status (certainly if this option is enabled in the tab Option of the Program Manager window).

9.4.1.24 ACI_TerminateFunction

ACI_FUNC ACI_TerminateFunction();

Description

This function terminates a current programming operation initiated by the ACI_StartFunction call.

9.4.1.25 ACI_GangTerminateFunction

ACI_FUNC ACI_GangTerminateFunction(ACI_GangTerminate_Params * params);

Description

This function, similar to the ACI_TerminateFunction which is applicable for stopping a single device programmer, is intended for terminating a current programming operation on one programming site belonging to the multiprogramming cluster or a gang programmer. The programming site (or socket) number is specified by the **SiteNumber** parameter from the **ACI_GangTerminate_Params** structure.

This function can be used **only** for the CPI2-B1 programmers launched in the gang mode (see the **/gang** parameter among other command line keys for the ACI_Launch function). In order to terminate an operation for a running single-site CPI2-B1 programmer use the ACI_TerminateFunction.

When the ACI_GangTerminateFunction initiates stopping a current operation it returns the control either when the operation was successfully stopped or with a delay defined by the **Timeout** parameter.

9.4.1.26 ACI_FileLoad

ACI_FUNC ACI_FileLoad(ACI_File_Params * params);

Description

This function loads a specified file into a specified buffer's layer. The control program running on the host PC should not worry about the file's format settings - the ChipProg-02 software takes care of this.

9.4.1.27 ACI_FileSave

ACI_FUNC ACI_FileSave(ACI_File_Params * params);

Description

This function saves a specified file from a specified buffer's layer. The ChipProg-02 software enables saving files in all popular formats: HEX, Binary, etc..

9.4.1.28 ACI_SettingsDialog

ACI_SettingsDialog();

Description

This macro opens the **Configure > Preferences** setting dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the **Configure > Preferences** setting dialog will appear on the computer screen, thus allowing manipulations in the dialog.

9.4.1.29 ACI_SelectDeviceDialog

ACI_SelectDeviceDialog();

Description

This macro sends a command that opens the **Select Device** dialog. The dialog will appear on the screen irrespective of the ChipProg-02 main window status; the main window can remain closed but the **Select Device** dialog will appear on the computer screen.

9.4.1.30 ACI_BuffersDialog

ACI_BuffersDialog();

Description

This macro opens the **Memory Dump Window Setup** dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the **Memory Dump Window Setup** dialog will appear on the computer screen to allow the buffer setup. See the dialog example below.
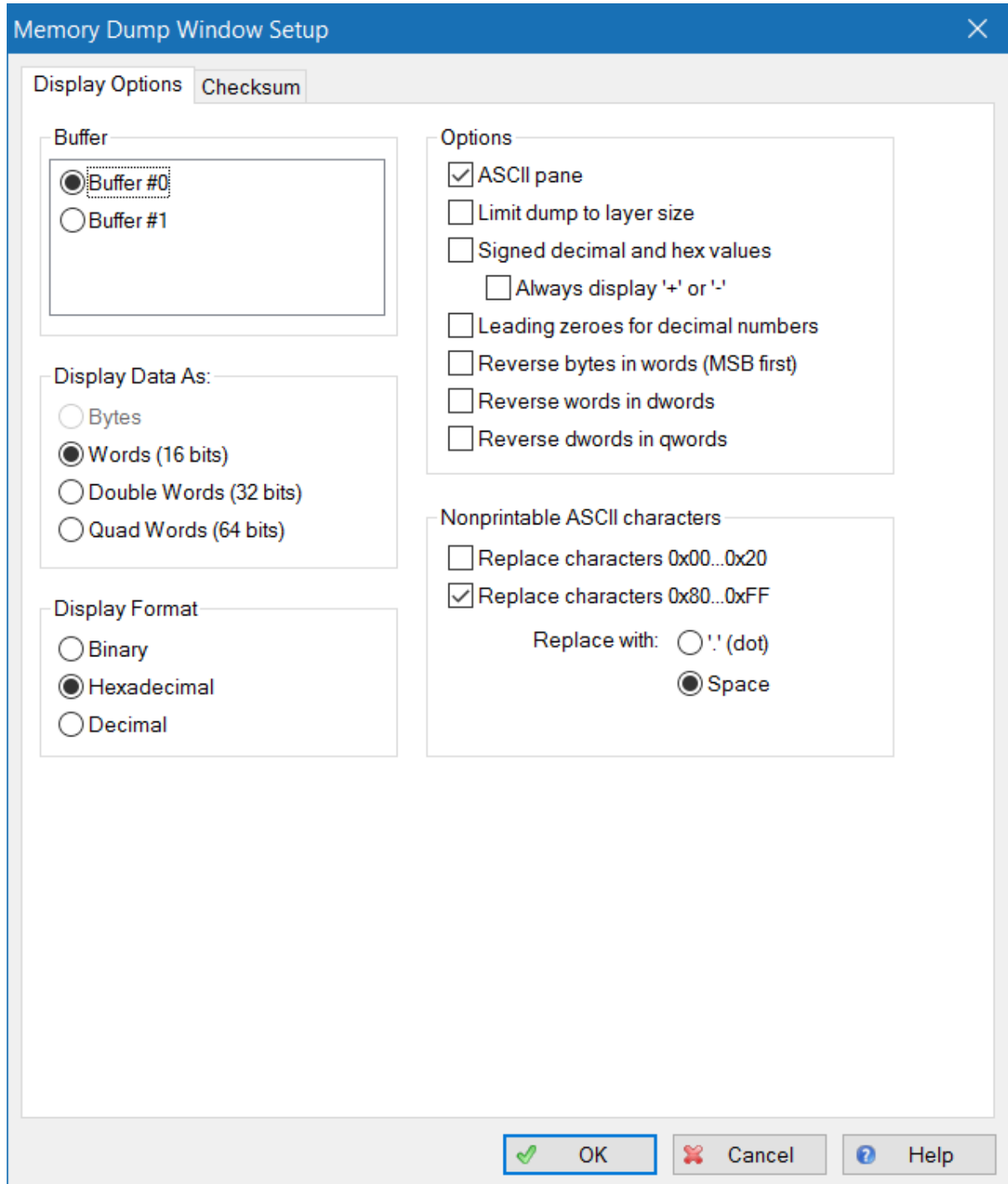
9.4.1.31 ACI_LoadFileDialog

ACI_LoadFileDialog();

Description

This macro opens the **Load File** dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the **Load File** dialog will appear on the computer screen. See the dialog example below.
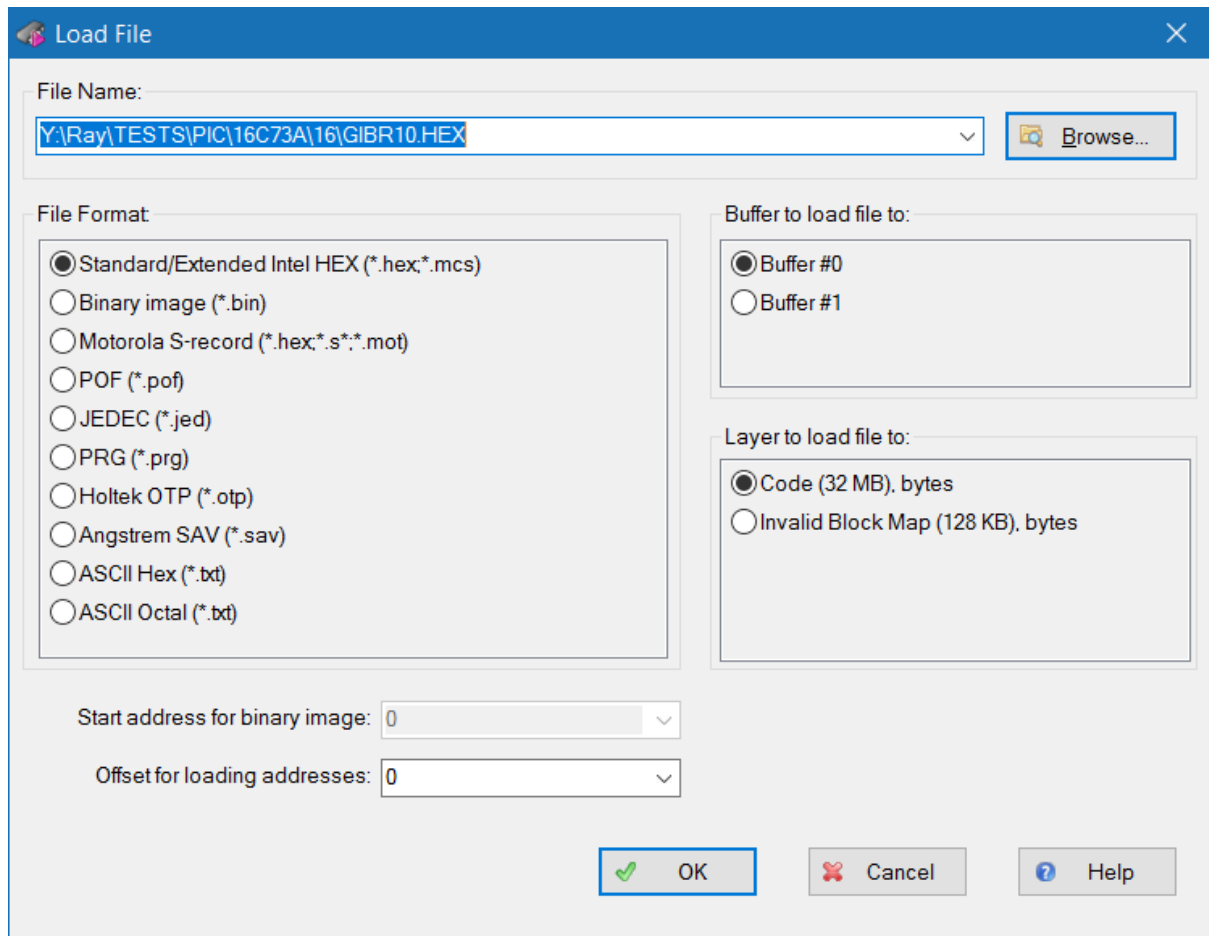
9.4.1.32 ACI_SaveFileDialog

ACI_SaveFileDialog();

Description

This macro sends a command that opens the **Save File** dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the **Save File** dialog will

appear on the computer screen. See the dialog example below.
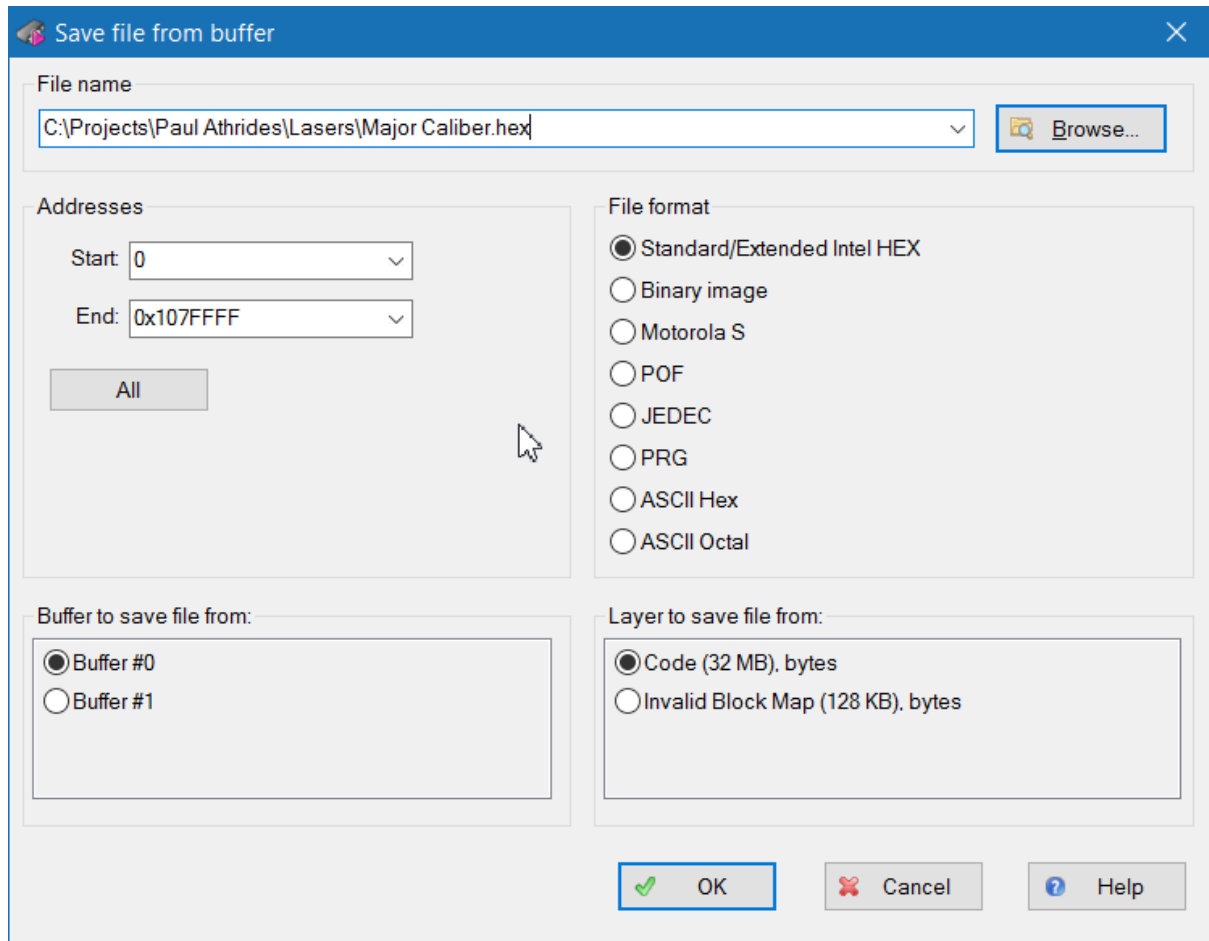
9.4.1.33 ACI_SerializationDialog

ACI_SerializationDialog();

Description

This macro sends a command that opens the **Serialization, Checksum, and Log Dialog**.

9.4.1.34 ACI_SetConnection

ACI_FUNC ACI_SetConnection(ACI_Connection_Params * params);

Description

This function identifies a current device programmer connection. Use this function when you control a number of device programmers by means of multiple calls of the ACI_Launch function. Each connection gets its own unique identifier. Executing of the ACI_Launch function returns the ConnectionId as part of the ACI_Launch_Params structure.

After establishing the connection, all the ACI functions following the ACI_SetConnection function will work exclusively with the established connection.

When ACI controls only one CPI2-B1 programmer it is not necessary to execute the ACI_SetConnection function; the ACI_Launch function automatically assigns a ConnectionId that is the next one in order.

The ConnectionId can be always checked by executing the function ACI_GetConnection.

9.4.1.35 ACI_GetConnection

ACI_FUNC ACI_GetConnection(ACI_Connection_Params * params);

Description

This function allows getting the identifier of a current device programmer connection. If a number of single CPI2-B1 programmers were launched, one after another, by multiple executions of the ACI_Launch function, then executing the ACI_GetConnection function returns a current ConnectionId parameter as a part of the ACI_Launch_Params structure.

See also ACI_SetConnection.

9.4.1.36 ACI_ConnectionStatus

ACI_FUNC ACI_ConnectionStatus();

.   ,   . .
            ,   ACI_ERR_SUCCESS.   -
                        ,   ACI_ERR_NOT_CONNECTED.

    .   ACI_SetConnection, ACI_GetConnection.

9.4.2   ACI Structures

This sections contains alphabetical list of all ACI structures.

9.4.2.1   ACI_Launch_Params

```
typedef struct tagACI_Launch_Params
{
  UINT    Size;              // (in)  Size of structure, in bytes
  LPCSTR ProgrammerExe;      // (in)  Programmer executable file name
  LPCSTR CommandLine;        // (in)  Optional programmer command-line parameters
  BOOL    DebugMode;         // (in)  Debug mode. Programmer window is not hidden
} ACI_Launch_Params;
```

ProgrammerExe	This is the name of the programmer executable file. If the parameter does not include a full path then the program will search for the UprogNT2.EXE file into the folder where the ACI.DLL resides. The target folder name, where the the UprogNT2.EXE file resides, is defined by the parameter "Folder" of the ""HKLM\SOFTWARE\Phyton\Phyton ChipProgUSB Programmer\x.yy.zz" key. It is supposed that multiple ChipProg-02 versions can be installed on the host computer.
CommandLine	This structure member specifies the command line options. One of the option is NULL (no keys). If the host computer drives a cluster of multiple programmers then the only way to launch a certain programmer is to specify the **/N<serial number>** for the **CommandLine** structure member.
DebugMode	This key controls the ChipProg-02 main window visibility. Setting TRUE for this structure member makes the ChipProg-02 main window visible. Then you can manipulate with the programmer using its user interface - open windows, set any programmer resources, execute programming operations, etc..

See also: ACI_Launch

9.4.2.2 ACI_ErrorString_Params

```
typedef struct tagACI_ErrorString_Params
{
  UINT   Size;                // (in)  Size of structure, in bytes
  CHAR   ErrorString[256];    // (out) Error string describing error code ACI_ERR_... returned by
                             //       call to ACI function
} ACI_ErrorString_Params;
```

ErrorString	String describing the error returned by the last ACI function call.

See also: ACI_ErrorString

9.4.2.3 ACI_Buffer_Params

```
typedef struct tagACI_Buffer_Params
{
  UINT   Size;           // (in)  Size of structure, in bytes
  DWORD  Layer0SizeLow;  // (in/out) Low 32 bits of layer 0 size, in bytes
  DWORD  Layer0SizeHigh; // (in/out) High 32 bits of layer 0 size, in bytes
                         //       Layer size is rounded up to a nearest value supported by progr
  LPCSTR BufferName;     // (in)  Buffer name
  UINT   BufferNumber;   // For ACI_CreateBuffer(): out: Created buffer number
                         // For ACI_ReallocBuffer(): in: Buffer number to realloc
  UINT   NumBuffers;     // (out) Total number of currently allocated buffers
  UINT   NumLayers;      // (out) Total number of layers in a buffer
} ACI_Buffer_Params;
```

`Layer0SizeLow,` `Layer0SizeHigh`	This structure member represents buffer layer #0's size in Bytes. This size lies in the range between 128K Bytes and 32G Bytes (may be changed in the future). The ChipProg-02 allows assigning buffers with fixed sizes only (see the list on the picture below). Any intermediate value will be automatically rounded up to one of the reserved buffer sizes. For example, if you enter '160000' the programmer will assign a 1MB buffer layer.
`BufferName`	Since it is used with the ACI_CreateBuffer function this structure member represents the name of the buffer that will be created. If used with the ACI_ReallocBuffer function will be ignored.
`BufferNumber`	After calling the ACI_CreateBuffer function this structure member returns the created buffer's number. After calling the ACI_ReallocBuffer function - the number of the buffer, size of which should be changed (re-allocate).
`NumBuffers`	This structure member represents the current number of memory buffers being opened.
`NumLayers`	This structure member represents the number of layers in memory buffers. This value is the same for all opened buffers.
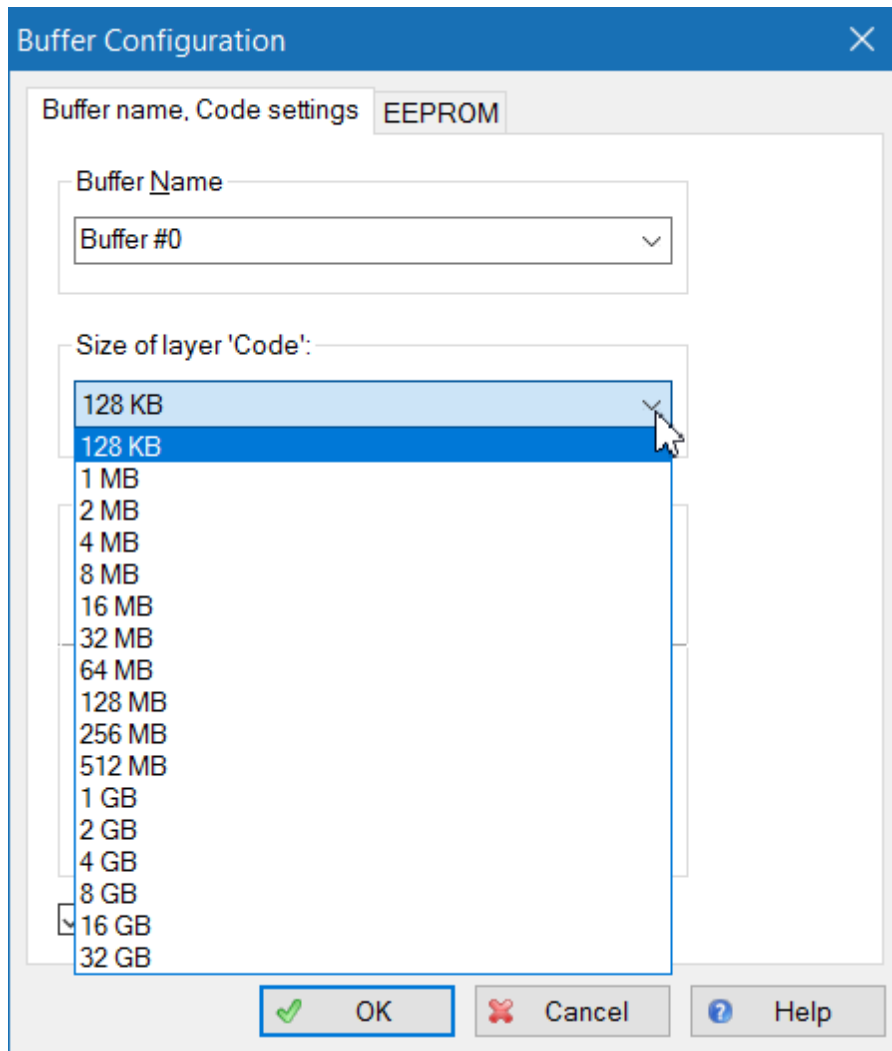
See also: ACI_CreateBuffer, ACI_ReallocBuffer

9.4.2.4 ACI_Config_Params

```
typedef struct tagACI_Config_Params
{
  UINT    Size;         // (in)  Size of structure, in bytes
  LPCSTR FileName;      // (in)  Options file name to load/save configuration
} ACI_Config_Params;
```

FileName	This is the name of the file that configures the programmer.

See also: ACI_LoadConfigFile, ACI_SaveConfigFile

9.4.2.5 ACI_ProjectParams

```
typedef struct tagACI_Project_Params
{
  UINT   Size;                 // (in)  Size of structure, in bytes
  LPCSTR ProjectName;          // (in)  Project file name
} ACI_Project_Params;
```

ProjectName	Project file name with extension.

See also: ACI_LoadProject.

9.4.2.6 ACI_Connection_Params

```
typedef struct tagACI_Connection_Params
{
  UINT   Size;                 // (in)  Size of structure, in bytes
  LPVOID ConnectionId;         // ACI_SetConnection(): (in), ACI_GetConnection(): (out)
                               // Connection identifier
} ACI_Connection_Params;
```

ConnectionId	An identifier of the connection with a particular device programmer. This is an abstract parameter that means nothing for the ACI user.

See also: ACI_SetConnection, ACI_GetConnection.

9.4.2.7 ACI_Device_Params

```
typedef struct tagACI_Device_Params
{
  UINT   Size;               // (in)       Size of structure, in bytes
  CHAR   Manufacturer[64]; // (in || out) Device Manufacturer
  CHAR   Name[64];         // (in || out) Device Name
} ACI_Device_Params;
```

Manufacturer	The manufacturer of the device being programmed
Name	The device part number as it is displayed in the programmer's device list

See also: ACI_SetDevice, ACI_GetDevice

9.4.2.8 ACI_File_Params

```
typedef struct tagACI_File_Params
{
```

```
    UINT    Size;               // (in)  Size of structure, in bytes
    LPCSTR  FileName;           // (in)  File name
    UINT    BufferNumber;       // (in)  Buffer number
    UINT    LayerNumber;        // (in)  Layer number
    UINT    Format;             // (in)  File format: see ACI_PLF_... and ACI_PSF_xxx constants
    DWORD   StartAddressLow;    // (in)  Low 32 bits of start address for ACI_FileSave().
                                //       For ACI_FileLoad(): Ignored if Format != ACI_PLF_BINARY
    DWORD   StartAddressHigh;   // (in)  High 32 bits of start address for ACI_FileSave().
                                //       For ACI_FileLoad(): Ignored if Format != ACI_PLF_BINARY
    DWORD   EndAddressLow;      // (in)  ACI_FileSave(): Low 32 bits of end address
    DWORD   EndAddressHigh;     // (in)  ACI_FileSave(): High 32 bits of end address
    DWORD   OffsetLow;          // (in)  Low 32 bits of address offset for ACI_FileLoad()
    DWORD   OffsetHigh;         // (in)  High 32 bits of address offset for ACI_FileLoad()
} ACI_File_Params;
```

`FileName`	The name of the file to be loaded to the CPI2-B1 buffer.
`BufferNumber`	The ordinal number of the destination buffer. Buffer numbers begins from zero.
`LayerNumber`	The ordinal number of the memory layer in the buffer. Layer numbers begins from zero.
`Format`	The loadable file's format. See the description of the **ACI_PLF_XXX*** constants in the aciprog.h header file (see below).
`StartAddressLow,` `StartAddressHigh`	1) If used with the ACI_FileSave function this parameter specifies the first (start) address in the source memory layer, from which the file will be saved. 2) If used with the ACI_FileLoad function, but only when it loads a file in the binary format (Format == ACI_PLF_BINARY), this parameter specifies the first (start) address of the destination memory layer, in which the file will be loaded. Binary images do not carry any addresses for the file loading.
`EndAddressLow,` `EndAddressHigh`	If used with the ACI_FileSave function this parameter defines the last (end) address of the source memory layer, from which the file will be saved.
`OffsetLow,` `OffsetHigh`	The address offset that shifts the file position in the destination memory layer. The offset can be negative as well as positive.

This is the bit definition from the aciprog.h header file:

*// ACI File formats for ACI_FileLoad()

```
#define ACI_PLF_INTEL_HEX       0  // Standard/Extended Intel HEX
#define ACI_PLF_BINARY          1  // Binary image
#define ACI_PLF_MOTOROLA_S      2  // Motorola S-record
#define ACI_PLF_POF             3  // POF
#define ACI_PLF_JEDEC           4  // JEDEC
#define ACI_PLF_PRG             5  // PRG
#define ACI_PLF_OTP             6  // Holtek OTP
#define ACI_PLF_SAV             7  // Angstrem SAV
#define ACI_PLF_ASCII_HEX       8  // ASCII Hex
#define ACI_PLF_ASCII_OCTAL     9  // ASCII Octal
```

See also: ACI_FileLoad, ACI_FileSave.

9.4.2.9 ACI_Function_Params

```
typedef struct tagACI_Function_Params
{
  UINT   Size;               // (in)  Size of structure, in bytes
  LPCSTR FunctionName;       // (in)  Name of a function to execute. If a function is under a sub
                             //       To execute Auto Programming, set FunctionName to NULL, empt
  UINT   BufferNumber;       // (in)  Buffer number to use
  BOOL   Silent;             // (in)  On error, do not display error message box, just copy error
  CHAR   ErrorMessage[512];  // (out) Error message string if ACI_ExecFunction() fails
} ACI_Function_Params;
```

FunctionName	The name of the CPI2-B1 function is one of those listed in the window **Functions** of the ChipProg-02 [Program Manager tab](). They are divided in two group (see the picture below): (1) the main functions applicable to a majority of the target devices (**Blank Check, Erase, Read, Program, Verify**) and (2) the device-specific lower level functions accessible through expandable sub-menus (for example, **Program Device Parameters**, **Erase Sectors**, **Lock Bits > Program Lock Bit 1, EEPROM > Read**, etc.). For such device-specific functions the **FunctionName** should be specified in the following way: **<List name>^<Function name>** (for example, **Device Parameters^Program**). To launch the **AutoProgramming** batch set the **FunctionName** either to NULL, a blank string, or the "Auto Programming". There is no restrictions in use of uppercase and lowercase characters in the function names.
BufferNumber	The ordinal number of the buffer the function operates with.
Silent	If this parameter is TRUE, then the error message dialog will be suppressed, the function execution will be terminated and will return the **ACI_ERR_FUNCTION_FAILED** code, and the error message will be copied to the **ErrorMessage**.
ErrorMessage	The destination of the error message that will be issued if the function fails.

See also: ACI_ExecFunction, ACI_StartFunction, ACI_GetStatus

9.4.2.10 ACI_GangStart_Params

```
typedef struct tagACI_GangStart_Params
{
  UINT   Size;              // (in)  Size of structure, in bytes
  UINT   SiteNumber;        // (in)  Site number to start auto programming at
  UINT   BufferNumber;      // (in)  Buffer number to use
  BOOL   Silent;            // (in)  On error, do not display error message box. Use ACI_GetStat
} ACI_GangStart_Params;
```

SiteNumber	The number of the device programmer socket in the gang programmer or in a programming cluster comprised of multiple CPI2-B1 programmers for which the ACI_GangStart function is launched. The site (socket) numbers begin from #0.
BufferNumber	The ordinal number of the memory buffer, content of which is required by the ACI_GangStart function. Numbers of CPI2-B1 memory buffers begin from #0.
Silent	If this parameter is TRUE, then the error message dialog will be suppressed, the function execution will be terminated and the **ACI_ERR_FUNCTION_FAILED** code will be returned.. Use the ACI_GetStatus function to receive the error message.

See also: ACI_GangStart, ACI_GetStatus

9.4.2.11 ACI_GangTerminate_Params

```
typedef struct tagACI_GangTerminate_Params
{
  UINT   Size;              // (in)  Size of structure, in bytes
  INT    SiteNumber;        // (in)  Site number to terminate operation (-1 == all sites)
  INT    Timeout;           // (in)  Timeout in milliseconds (-1 == infinite) to wait for operat
  BOOL   SiteStopped;       // (out) TRUE if operation was stopped, FALSE if timeout occurred
} ACI_GangTerminate_Params;
```

SiteNumber	The site (socket) number you want terminating a current operation on. Socket numbers begin from 0 (zero). If you specify SiteNumber = -1 (minus one) this will terminate operations on all sites of the gang machine.
Timeout	A time interval in milliseconds, during of which the ACI_GangTerminateFunction holds expecting an acknowledgment of the successful operation termination. The function will return control either upon getting such an acknowledgment or upon expiring a specified Timeout. If you specify the Timeout = -1 (minus one) it will never expire.
SiteStopped	This parameter indicates whether the ACI_GangTerminateFunction succeeded. In case of successful termination an operation *before* expiring the Timeout the SiteStopped parameter sets TRUE. Otherwise, it will be set FALSE.

See also: ACI_GangTerminateFunction, ACI_TerminateFunction.

9.4.2.12 ACI_Layer_Params

```
typedef struct tagACI_Layer_Params
{
  UINT    Size;                    // (in)  Size of structure, in bytes
  UINT    BufferNumber;            // (in)  Number of buffer of interest, the first buffer number is
  UINT    LayerNumber;             // (in)  Number of layer of interest, the first layer number is 0
  DWORD   LayerSizeLow;            // (out) Low 32 bits of layer size, in bytes
  DWORD   LayerSizeHigh;           // (out) High 32 bits of layer size, in bytes
  DWORD   DeviceStartAddrLow;      // (out) Low 32 bits of device start address for this layer
  DWORD   DeviceStartAddrHigh;     // (out) High 32 bits of device start address for this layer
  DWORD   DeviceEndAddrLow;        // (out) Low 32 bits of device end address for this layer
  DWORD   DeviceEndAddrHigh;       // (out) High 32 bits of device end address for this layer
  DWORD   DeviceBufStartAddrLow;   // (out) Low 32 bits of device memory start address in buffer for
  DWORD   DeviceBufStartAddrHigh;  // (out) High 32 bits of device memory start address in buffer fo
  UINT    UnitSize;                // (out) Size of layer unit, in bits (8, 16 or 32)
  BOOL    FixedSize;               // (out) Size of layer cannot be changed with ACI_ReallocBuffer()
  CHAR    BufferName[64];          // (out) Buffer name
  CHAR    LayerName[64];           // (out) Layer name, cannot be changed
  UINT    NumBuffers;              // (out) Total number of currently allocated buffers
```

```
    UINT    NumLayers;                   // (out) Total number of layers in a buffer
} ACI_Layer_Params;
```

BufferNumber	The ordinal number of the memory buffer, content of which is required by the ACI_GetLayer function. Numbers of CPI2-B1 memory buffers begin from #0.
LayerNumber	The ordinal number of the layer in the memory buffer, the content of which is required by the ACI_GetLayer function. The layer numbers begins from #0.
LayerSizeLow, LayerSizeHigh	Here the function returns the range of the memory layer's addresses in bytes.
DeviceStartAddrLow, DeviceStartAddrHigh	Here the function returns the device's start address for the selected memory layer. This address is the device's property and strictly depends on the device type; usually this value is zero. Do not mix it up with the start address of a programming operation that can be shifted by a certain offset value.
DeviceEndAddrLow, DeviceEndAddrHigh	Here the function returns the device's end address for the selected memory layer. This address is the device's property and strictly depends on the device type. Do not mix it up with the end address of a programming operation editable in the setup dialog. The selected layer's address range can be defined as a difference between the end address and the start address plus 1.
DeviceBufStartAddrLow, DeviceBufStartAddrHigh	Here the function returns the start address for the selected memory buffer - usually this value is zero.
UnitSize	This structure member specifies formats of the data in a memory layer: 8 for the 8-bit devices, 16 - for 16-bit devices and 32 for 32-bit devices.
FixedSize	This flag, if TRUE, disables resizing the memory layer by the ACI_ReallocBuffer function. There is one restriction on use of this flag: since the layer #0 is always resizeable the **FixedSize** is always FALSE for the layer #0.
BufferName	The name of the memory buffer as it was defined in the CPI2-B1 interface or by the ACI_CreateBuffer function call.
LayerName	Reserved name of the memory buffer's layer. It cannot be changed by the ACI.DLL user.
NumBuffers	The number of the assigned memory buffers.
NumLayers	The number of layers in the programmer's memory buffers. This is a pre-defined device-specific value that is the same for all memory buffers.

See also: ACI_GetLayer

9.4.2.13 ACI_Memory_Params

```
typedef struct tagACI_Memory_Params
{
  UINT   Size;              // (in)  Size of structure, in bytes
  UINT   BufferNumber;      // (in)  Number of buffer of interest, the first buffer number is 0
  UINT   LayerNumber;       // (in)  Number of layer of interest, the first layer number is 0
  DWORD  AddressLow;        // (in)  Low 32 bits of address, in layer units (natural to device addr
  DWORD  AddressHigh;       // (in)  High 32 bits of address, in layer units (natural to device add
  PVOID  Data;              // (in || out) Buffer to data to read to or write from
  DWORD  DataSize;          // (in)  Size of data to read or write, in layer units, max. 16 MB (0x1
  DWORD  FillValue;         // (in)  Value to fill buffer with, used by ACI_FillLayer() only
} ACI_Memory_Params;
```

BufferNumber	The ordinal number of the buffer to read from or to write into. The buffer numerical order begins from zero.
LayerNumber	The ordinal number of the memory buffer's layer to read from or to write into. The layer numerical order begins from zero.
AddressLow, AddressHigh	The start address in the memory layer to read from or to write into represented in the units specified by the chosen device manufacturer - Bytes, Words, Double Words. This structure member is ignored in case of use with the ACI_FillLayer function.
Data	Since these are used with different ACI functions this structure member has different meanings.In case of use with the ACI_ReadLayer function it represents the pointer to the data read out from the CPI2-B1 buffer's layer. In case of use with the ACI_WriteLayer - the pointer to the data to be written to the CPI2-B1 buffer's layer. The **Data** is ignored if it is used with the ACI_FillLayer function.
DataSize	This structure member represents the data format given in memory units specified by the device manufacturer (Bytes, Words or Double Words). The program ignores the **DataSize** if it is used with the ACI_FillLayer function.
FillValue	This is the data pattern that fills an active CPI2-B1 buffer's layer by means of the ACI_FillLayer function. If, for example, the **FillValue** is presented in the DWORD format then the 8-bit memory layers will be filled with the lower byte of the **FillValue** pattern, the 16-bit layers - with the lower 16-bit word and the 32-bit layers - with a whole **FillValue** pattern.

See also: ACI_ReadLayer, ACI_WriteLayer, ACI_FillLayer

9.4.2.14 ACI_ProgOption_Params

```
typedef struct tagACI_ProgOption_Params
{
  UINT   Size;              // (in)  Size of structure, in bytes
  LPCSTR OptionName;        // (in)  Name of the option. For lists, it should be in the form "Lis
  CHAR   Units[32];         // (out) Option measurement units ("kHz", "V", etc.)
```

```
CHAR    OptionDescription[64]; // (out) Description of the option
CHAR    ListString[64];        // (out) For ACI_PO_LIST option: Option string for Value.ListIndex
UINT    OptionType;            // (out) Option type: see ACI_PO_xxx constants
BOOL    ReadOnly;              // (out) Option is read-only
union                          // (in || out) Option value
{
  LONG    LongValue;           // (in || out) Value for ACI_PO_LONG option
  FLOAT   FloatValue;          // (in || out) Value for ACI_PO_FLOAT option
  LPSTR   String;              // (in || out) Pointer to string for ACI_PO_STRING option
  ULONG   CheckBoxesValue;     // (in || out) Value for ACI_PO_CHECKBOXES option
  UINT    StateIndex;          // (in || out) State index for ACI_PO_LIST option
  LPBYTE  Bitstream;           // (in || out) Pointer to bitstream data for ACI_PO_BITSTREAM option
} Value;

UINT  VSize;                   // For ACI_SetProgOption():
                               //    in: Size of Bitstream if OptionType == ACI_PO_BITSTREAM
                               // For ACI_GetProgOption():
                               //    in: Size of buffer pointed by Bitstream if OptionType == ACI_PO
                               //    in: Size of buffer pointed by String if OptionType == ACI_PO_ST
                               //   out: Size of buffer needed for storing Bitstream data if OptionT
                               //        Set Value.Bitstream to NULL to get buffer size without sett
                               //   out: Size of buffer needed for storing String if OptionType == A
                               //        Set Value.String to NULL to get buffer size without setting
UINT  Mode;                    // (in) For ACI_SetProgOption(): SEE ACI_PP_MODE_... constants
} ACI_ProgOption_Params;
```

OptionName	The name of the programming option - for example "Vcc". For the **ACI_PO_LIST - type** options, where the options are grouped into a list, you should specify both the list name and the option name in the following way: **\<List name>^\<Option name> (**For example, **Configuration_bits^Generator**. There are no restrictions on use of uppercase and lowercase characters in the option names.
Units	After executing the ACI_GetProgOption function this structure member returns an abbreviation of the units, in which the programmer represents or measures the **OptionName**. For example, for the Vcc structure member, Units = "V".
OptionDescription	After executing the ACI_GetProgOption function this structure member returns the option description.
ListString	After executing the ACI_GetProgOption function for the **ACI_PO_LIST** - type options this structure member returns a string that describes the current option's value or status. For example, **XT - Standard Crystal** for the option **Configuration bits^Generator**.
OptionType	After executing the ACI_GetProgOption function this structure member returns the option's presentation format - for example: integer, floating point, list, bitstream, etc.. See the **ACI_PO_xxx*** constant description in the aciprog.h header file below.
ReadOnly	Setting **ReadOnly**=TRUE disables modification of the option specified by the ACI_GetProgOption function.
Value	Use of this union depends on the **ACI_PO_LONG*** option type as it is shown in the matrix below: **Option type** / **Use of the Value union** ACI_PO_LONG / The option is in the **Value.LongValue**

	ACI_PO_FLOAT	The option is in the **Value.FloatValue**	
	ACI_PO_STRING	The option is represented as a string, the pointer on which is in the **Value.String**. See the note below.	
	ACI_PO_CHECKBOXES	The option represents a 32-bit integer word, in which you can individually toggle each bit that represents a particular flag in the option setting dialog. The option is in the Value.CheckBoxesValue. See, for example, the Fuse setting dialog for the ATtiny45 MCU implemented as an array of check boxes.	
	ACI_PO_LIST	It represents a list of alternative choices. Only one of them can be selected at a time, so the parameter changes its value in a range 0, 1, 2 to N. The option is in the **Value.CheckStateIndex**. See, for example, the Oscillators setting dialog for the PIC12F509 MCU implemented as an alternatively chosen radio buttons	
	ACI_PO_BITSTREAM	Stream of bits. This option type is not in use yet but can be used for future applications.	

VSize	Size of the buffer assigned for storing the string if the option type is the **ACI_PO_STRING**. See the note below.

Mode	Mode of using of the structure member Value (See the description of the **ACI_PP_xxx**** constants in the aciprog.h<) header file:

The Mode setting (value)	Use of the parameter Value
ACI_PP_MODE_VALUE	1) For measuring (getting): use the **Value** in order to get an actual **Option** value; 2) For setting: use the **Value** to set a particular **Option** value.
ACI_PP_MODE_DEFAULT _VALUE	1) If used with the ACI_GetProgOption function it issues a command to put the default **Option** value into the **Value**. 2) If used with the ACI_SetProgOption function, the Value will be ignored; the **Option** will be set to the default level defined in the CPI2-B1 hardware.
ACI_PP_MODE_MIN_VAL UE	1) If used with the ACI_GetProgOption function it commands to put the minimal **Option** value into the **Value**. 2) If used with the ACI_SetProgOption function the **Value** will be ignored; the **Option** will be set to the minimal level defined in the CPI2-B1 hardware, if it is possible for the **Option** of this type.
ACI_PP_MODE_MAX_VAL UE	1) If used with the ACI_GetProgOption function it commands to put the maximal **Option** value into the **Value**. 2) If it is used with the ACI_SetProgOption function the **Value** will be ignored; the **Option** will be set to the maximal level defined in the CPI2-B1 hardware, if it is possible for the **Option** of this type.

This is the bit definition from the aciprog.h header file:

*// ACI Programming Options defines
#define ACI_PO_LONG 0 // Signed integer option
#define ACI_PO_FLOAT 1 // Floating point option

```
#define ACI_PO_STRING              2 // String option
#define ACI_PO_CHECKBOXES          3 // 32-bit array of bits
#define ACI_PO_LIST                4 // List (radiobuttons)
#define ACI_PO_BITSTREAM           5 // Bit stream - variable size bit array
```

**// ACI Programming Option Mode constants for ACI_GetProgOption()/ACI_SetProgOption()

```
#define ACI_PP_MODE_VALUE          0 // Get/set value specified in Value member of the
ACI_ProgOption_Params structure
#define ACI_PP_MODE_DEFAULT_VALUE  1 // Get/set default option value, ignore Value member
#define ACI_PP_MODE_MIN_VALUE      2 // Get/set minimal option value, ignore Value
member
#define ACI_PP_MODE_MAX_VALUE       3 // Get/set maximal option value, ignore Value
member
```

Note for use of the ACI_GetProgOption**:**
In order to get the buffer size necessary for storing the Option ACI_PO_STRING, you should make the first call of the ACI_GetProgOption function with the Value.String= NULL. Then the function will return the VSize equal to the buffer size, including zero at the string's end. In your program, assign the buffer of this size, put the Value.String into the buffer pointer and call the ACI_GetProgOption again.

Fuse Bits

☐ CKSEL0
☑ CKSEL1
☑ CKSEL2
☑ CKSEL3
☑ SUT0
☐ SUT1
☐ BODEN
☐ BODLEVEL
☐ BOOTRST
☑ BOOTSZ0
☑ BOOTSZ1
☐ EESAVE
☐ CKOPT

Check all	Uncheck all	All default

Note

'Checked' option means logical state '0'

✔ OK   ✖ Cancel   ❓ Help
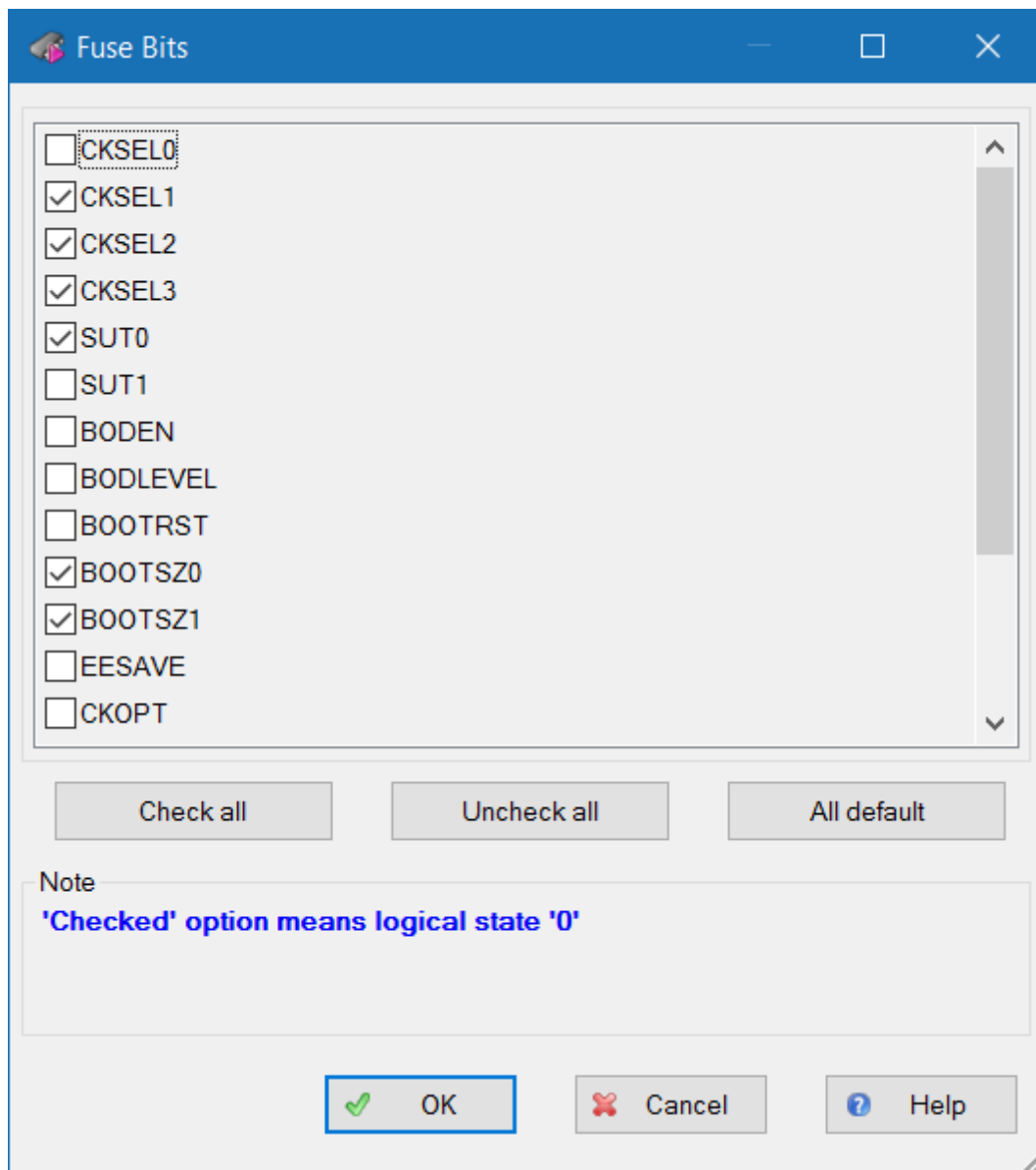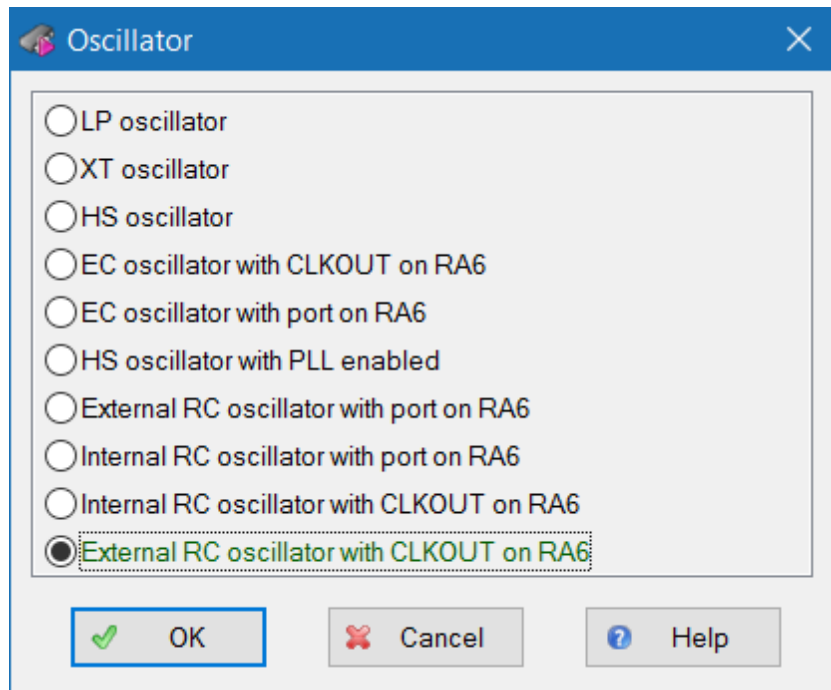
See also: ACI_GetProgOption, ACI_SetProgOption

9.4.2.15 ACI_Programming_Params

```
typedef struct tagACI_Programming_Params
{
  UINT   Size;                    // (in)       Size of structure, in bytes
  BOOL   InsertTest;              // (in || out) Test if device is attached
  BOOL   CheckDeviceId;           // (in || out) Check device identifier
  BOOL   ReverseBytesOrder;       // (in || out) Reverse bytes order in buffer
  BOOL   BlankCheckBeforeProgram; // (in || out) Perform blank check before programming
  BOOL   VerifyAfterProgram;      // (in || out) Verify after programming
  BOOL   VerifyAfterRead;         // (in || out) Verify after read
  BOOL   SplitData;               // (in || out) Split data: see ACI_SP_xxx constants
  BOOL   DeviceAutoDetect;        // (in || out) Auto detect device in socket (not all of the pro
  BOOL   DialogBoxOnError;        // (in || out) On error, display dialog box
  UINT   AutoDetectAction;        // (in || out) Action to perform on device autodetect or 'Start
  DWORD  DeviceStartAddrLow;      // (in || out) Low 32 bits of device start address for programm
  DWORD  DeviceStartAddrHigh;     // (in || out) High 32 bits of device start address for program
  DWORD  DeviceEndAddrLow;        // (in || out) Low 32 bits of device end address for programmin
  DWORD  DeviceEndAddrHigh;       // (in || out) High 32 bits of device end address for programmi
  DWORD  DeviceBufStartAddrLow;   // (in || out) Low 32 bits of device memory start address in bu
  DWORD  DeviceBufStartAddrHigh;  // (in || out) High 32 bits of device memory start address in b
} ACI_Programming_Params;
```

InsertTest (Irrelevant for CPI2-B1)	This is the command to check the device insertion before starting any programming operations on the device. The procedure will check if every chip leads have good contact in the programming socket.
CheckDeviceId	This is the command to check a unique internal device identifier before the device programming.

ReverseBytesOrder	This is the command to reverse the byte order in 16-bit words when programming the device, reading it or verifying the data. This structure member does not effect the data value in the CPI2-B1 memory buffers - these data remain the same as they were loaded.
BlankCheckBeforeProgram	This is the command to check whether the device is blank before executing the Program command.
VerifyAfterProgram	This is the command to verify the data written into the device every time after executing the Program command.
VerifyAfterRead	This is the command to verify the data written into the device every time after executing the Read command.
SplitData	This is the command to split data in accordance with the value of the constants **ACI_SP_xxx*** in the aciprog.h file (see below). This allows 8-bit memory devices to be cascaded in multiple memory chips to be used in the systems with 16- and 32-bit address and data buses.
DeviceAutoDetect (Irrelevant for CPI2-B1)	This is the command to scan all the device's leads in a process of the device insertion into the programming socket. If the **DeviceAutoDetect** is TRUE the programmer will check whether all of the device's leads are reliably gripped by the programmer socket's sprung contacts. Only when the reliable device insertion is acknowledged, the program launches a chosen programming operation, script or a batch of single operations programmed in the Auto Programming dialog. (Irrelevant for CPI2-B1)
DialogBoxOnError	If this structure member is TRUE then any error that occurs in any programming operation will generate error messages and will open associated dialogs. If this attribute is FALSE the error messages will not be issued.
AutoDetectAction (Irrelevant for CPI2-B1)	If the **DeviceAutoDetect** is TRUE then values of the **ACI_AD_xxx**** constants in the aciprog.h file define a particular action triggered either on manual pushing the **Start** button or upon auto detection of reliable insertion of the device into the programmer's socket (see below). (Irrelevant for CPI2-B1) <table><tr><td>**AutoDetectAction value**</td><td>**What to do (action)**</td></tr><tr><td>ACI_AD_EXEC_FUNCTION</td><td>Launch the programming operation (function) currently highlighted in the Program Manager tab.</td></tr><tr><td>ACI_AD_EXEC_AUTO</td><td>Launch a batch of single operations programmed in the Auto Programming dialog.</td></tr><tr><td>ACI_AD_EXEC_SCRIPT</td><td>Perform the script specified in the Script File dialog.</td></tr><tr><td>ACI_AD_DO_NOTHING</td><td>Do not act (ignore). Then it is possible to resume operations only by executing either the ACI_ExecFunction or ACI_StartFunction.</td></tr></table>
DeviceStartAddrLow, DeviceStartAddrHigh	This structure member defines a physical start address of the device to perform a specified programming operation (function). For example: "...read the chip content beginning at the address 7Fh". Not all the functions use this parameter.
DeviceEndAddrLow, DeviceEndAddrHigh	This parameter defines a physical end address, beyond which a specified programming operation (function) will not proceed. For example: "...program the chip until the address 0FFh". Not all the programmer functions use this parameter.

`DeviceBufStartAddrLow,` `DeviceBufStartAddrHigh`	This structure member defines the buffers layer's start address from which to perform a specified programming operation (function). For example: "...read the chip and move the data to the buffer beginning at the address 10h". Not all the programmer functions use this parameter.

This is the bit definition from the aciprog.h header file:

* // ACI Data Split defines
```
#define ACI_SP_NONE          0
#define ACI_SP_EVEN_BYTE     1
#define ACI_SP_ODD_BYTE      2
#define ACI_SP_BYTE_0        3
#define ACI_SP_BYTE_1        4
#define ACI_SP_BYTE_2        5
#define ACI_SP_BYTE_3        6
```

** // ACI Device Auto-Detect or 'Start' button action
#define ACI_AD_EXEC_FUNCTION   0 // Execute the function currently selected in the list
#define ACI_AD_EXEC_AUTO   1 // Execute the **Auto Programming** command
#define ACI_AD_EXEC_SCRIPT   2 // Execute the script chosen in the programmer Script File
dialog
#define ACI_AD_DO_NOTHING   3 // Do nothing

See also: ACI_SetProgrammingParams, ACI_GetProgrammingParams

9.4.2.16 ACI_PStatus_Params

```
typedef struct tagACI_PStatus_Params
{
  UINT   Size;               // (in)  Size of structure, in bytes
  UINT   SiteNumber;         // (in) For the Gang mode: site number to get status of, otherwise i
  BOOL   Executing;          // (out) The function started by ACI_StartFunction() is executing
  UINT   PercentComplete;    // (out) Percentage of the function completion, valid id Executing !
  UINT   DeviceStatus;       // (out) Device/socket status, see the ACI_DS_XXX constants
  BOOL   NewDevice;          // (out) New device inserted, no function has been executed yet. Val
  BOOL   FunctionFailed;     // (out) TRUE if last function failed
  CHAR   FunctionName[128];  // (out) Name of a function being executed if Executing != FALSE. If
  CHAR   ErrorMessage[512];  // (out) Error message string if FunctionFailed != FALSE
} ACI_PStatus_Params;
```

`SiteNumber`	If the ChipProg-02 was launched in the Gang mode (with the command line key / gang) and controls either the gang programmer or a cluster of single programming machines, then before starting the ACI_GetStatus function the **SiteNumber** parameter must contain the ordinal number of the programming site (socket) for which the status is required. The site numbers begin from #0.
`Executing`	This parameter is TRUE while the CPI2-B1 operation, launched by the

	ACI_StartFunction, is in progress.
PercentComplete	While the **Executing** == TRUE this parameter represents a percentage of the function completion - from 0 to 100.
DeviceStatus (Irrelevant for CPI2-B1)	This structure member defines insertion of the device into the programmer ZIF socket if the device insertion auto detection function is enabled. See the description of the **ACI_DS_XXX*** constants in the aciprog.h file. See the matrix below:

Status	Description
ACI_DS_OK	The device is in the socket and the device's leads are reliably gripped by the programmer's ZIF socket's sprung contacts.
ACI_DS_OUT_OF_SOCKET	There is no device in the programmer's ZIF socket.
ACI_DS_SHIFTED	The device's leads are reliably inserted into the socket but the device is incorrectly positioned in the socket (shifted or inserted upside down). The same status may indicate that the device type selected in the Select Device does not correspond to the type of chip in the programmer's socket.
ACI_DS_BAD_CONTACT	The device's leads are not reliably gripped by the socket's sprung contacts. In most cases this is an intermediate situation while an operator is inserting the chip to the socket or is removing it.
ACI_DS_UNKNOWN	It is impossible to detect the status because the device insertion auto detection feature is disabled or this feature is not supported by this programmer at all.

NewDevice (Irrelevant for CPI2-B1)	This structure member is a flag that acknowledges replacing a programmed device in the programmer's socket by a new, presumably a blank device. It works only when the device insertion auto detection function is enabled. The **NewDevice** == FALSE while the already programmed chip is still in the socket and has not been replaced by a new one. After removing the programmed device from the socket the **NewDevice** toggles to TRUE.
FunctionFailed	This is an indicator of the function execution's result. It is set to FALSE when the ACI_StartFunction launches a programming operation and remains FALSE while the operation is in progress. If the programming operation fails and the parameter **Executing** becomes FALSE the **FunctionFailed** flag toggles to TRUE.
FunctionName	This is either the name of the programming operation (function) being currently executed or the name of the failed function, if the **FunctionFalied** == TRUE.
ErrorMessage	The destination of the error message if the function fails, i.e. the **FunctionFalied** == TRUE.

This is the bit definition from the aciprog.h header file:

***// ACI Device Status**

```
#define ACI_DS_OK                0  // Device detected, pin contacts are ok
#define ACI_DS_OUT_OF_SOCKET     1  // No device in the socket
#define ACI_DS_SHIFTED           2  // Wrong device insertion is detected (shifted or inserted
upside down)
#define ACI_DS_BAD_CONTACT       3  // Bad pin contact(s)
#define ACI_DS_UNKNOWN           4  // Unknown (Auto Detect is probably off)
```

See also: ACI_ExecFunction, ACI_StartFunction, ACI_GetStatus

Index

- D -

- E -

- F -

Back Cover