



CPI2-B1

In-System Device Programmers

© 2021 Phyton, Inc. Microsystems and Development Tools

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: May 2021 in (wherever you are located)

Table of Contents

Foreword	0
Part I Introduction	17
1 Terminology	17
2 CPI2-B1 device programmer	19
Features Overview	20
Hardware characteristics	20
Software features	21
Communication Interfaces	22
Connector TARGET	22
Connector CONTROL	24
Single- and Gang-site programming	26
Part II Installation and Launching	28
1 Getting Assistance.....	28
2 Hardware installation.....	29
3 System Requirements.....	32
4 Software Installation.....	32
5 Launching device programmers.....	36
6 Setup Wizard and Startup Dialog.....	38
Part III Control Interfaces	46
1 Using Projects.....	47
2 Graphical User Interface.....	48
User Interface Overview	48
Toolbars	49
Menus	50
The File Menu	51
Configuration Files	52
The View Menu	52
The Project Menu	52
The Project Options Dialog.....	53
The Open Project Dialog.....	54
Export and Import Project Dialogs.....	54
Project Repository	56
The Configure Menu	57
The Select Device Dialog.....	58
The Buffers Dialog	61
The Buffer Configuration Dialog.....	61
The Serialization, Checksum, and Log Dialog.....	63
Shadow Areas	64
General settings	68
Device Serialization	69
Checksum	69
Signature string	70

Custom Shadow Areas.....	71
Log file	73
The Sata Caching, Standalone... Dialog.....	74
IP Address Setting Dialog.....	74
Simplified User Interface Editor.....	77
The Preferences Dialog	78
The Environment Dialog.....	79
Fonts	80
Colors	80
Mapping Hot Keys	81
Toolbar	82
Messages	82
Miscellaneous Settings	82
The Editor Options Dialog.....	83
The General Tab	83
The Key Mappings Tab	85
The Edit Key Command Dialog.....	86
The Commands Menu.....	86
Calculator	88
The Script Menu	88
The Window Menu	89
The Help Menu	90
License Management Dialog.....	90
Windows	92
The Device Information Window.....	92
The Device and Algorithm Parameters Window.....	93
The Buffer Dump Window.....	95
The 'Configuring a Buffer' dialog.....	97
The 'Buffer Setup' dialog.....	98
The 'Display from address' dialog.....	100
The 'Modify Data' dialog.....	100
The 'Memory Blocks' dialog.....	100
The 'Load File' dialog	102
File Formats	103
The 'Save File' dialog	104
The Console Window.....	104
The Program Manager Window.....	105
The Program Manager tab.....	107
Auto Programming	108
The Options tab	109
Split data	110
The Statistics tab	111
The Memory Card Window.....	113
Windows for Scripts.....	113
3 Simplified User Interface	113
Settings of Simplified User Interface	116
Operations with Simplified User Interface	120
4 Command Line Interface.....	120
Command Line Options	120
Command Line Option Files	125
5 On-the-Fly Control Interface.....	126
On-the-Fly Command Line Options	127
On-the-Fly utility return codes	131

On-the-Fly Control Examples	132
Part IV Standalone Operation Mode	132
1 Preparing Standalone Mode Jobs.....	133
Data Caching	134
Standalone Jobs	136
Standalone mode settings	137
Device serialization	138
Permissions and setting limits	141
SD card window	142
2 Switching to Standalone Mode	143
3 Standalone Mode Monitor	145
4 Example of Setting Up Standalone Mode.....	147
Part V Software Development Kit (SDK)	158
1 ACI Components.....	158
2 Using ACI	159
3 ACI Functions.....	160
4 ACI Structures.....	164
5 Examples	165
6 API Explorer.....	166
Part VI Integration with NI LabVIEW	168
1 LabVIEW Integration Using Command Line.....	169
2 LabVIEW Integration Using ACI.....	172
LabVIEW Integration Examples	173
Part VII Scripting	176
1 Scripting Overview.....	176
Simple example	176
2 The Startup Script.....	177
3 Running Scripts.....	177
The Script Files Dialog	178
The User Window	180
The I/O Stream Window	180
4 Debugging a Script.....	180
The Script Window	181
Menu and Toolbar.....	181
The AutoWatches Pane.....	182
The Watches Window	182
The Display Watches Options Dialog.....	183
The Add Watch Dialog.....	184
5 Script Editor.....	184
The File Menu	186
The Edit Menu	186
Block Operations	187
Condensed Mode	188

Syntax Highlighting	189
Automatic Word Completion	189
The Quick Watch Function	190
Dialogs	190
The Search for Text Dialog.....	190
The Replace Text Dialog.....	191
The Confirm Replace Dialog.....	192
The Multi-File Search Results Dialog.....	192
Search for Regular Expressions.....	193
The Set/Retrieve Bookmark Dialogs.....	193
The Condensed Mode Setup Dialog.....	194
The Display from Line Number Dialog.....	194

Part VIII Reference 194

1 How to	194
How to check if device is blank	194
How to erase a device	195
How to read data from device	195
How to program a device	195
How to load a file into a buffer.....	195
How to edit data before programming.....	196
How to configure target device.....	196
How to write information into the device.....	196
How to verify programming	197
How to save data to disc	197
Multi-Target Programming	197
2 Error Messages.....	198
Error Load/ Save File	198
Error Addresses	198
Error sizes	199
Error command-line option	199
Error Programming option	199
Error DLL	200
Error USB	200
Error programmer hardware	200
Error internal	201
Error configuration	201
Error device	201
Error check box	201
Error mix	201
Warning	202
3 Expressions.....	202
Operations	202
Operands	204
Expression Examples	204
4 Scripting Reference.....	204
Scripting Language Description	205
Difference Between Scripting and C Languages	205
Scripting Language Syntax.....	206
Format	206
Comments	206
Identifiers	207

Reserved words	207
Integer constants	207
Long integer constants.....	208
Floating-point constants.....	208
Character constants	209
String constants	209
Basic Data Types.....	209
Data byte order	210
Operations and Expressions.....	210
Operand Metadesignation.....	211
Arithmetic Operations	211
Assignment Operations.....	212
Relation Operations	214
Logical Operations	215
Array Operations	216
Bit Operations	216
Other Operations	217
Operation Execution Priorities and Order.....	218
Operand Execution Order.....	219
Arithmetic Conversions in Expressions.....	219
Operators	220
Format and nesting	220
Operator label	220
Composite operator	220
Operator-expression	221
Operator Break	221
Operator Continue	222
Operator Return	222
Operator Goto	222
Conditional Operator If-Else.....	222
Cycle Operator While	224
Cycle Operator Do-While.....	224
Cycle Operator For	225
Functions	225
Function Definition	225
Function Call	226
The main Function	226
Descriptions	227
Basic Types	227
Arrays	227
Local Variable Definition.....	228
Global Variable Definition.....	228
Variable Initialization	229
External Object Description.....	229
Directives of the Script Language Preprocessor.....	230
Identifier Change (#define).....	230
Inclusion of Files (#include).....	230
Conditional Compilation.....	231
Predefined Symbols in the Script File Compilation.....	231
Built-in Functions by Group	231
Buffer access functions.....	232
Checksum	232
GetByte	233
GetDword	233

GetMemory	233
GetWord	234
LoadProgram	234
MaxAddr	235
MinAddr	235
ReloadProgram	235
SaveData	235
SetByte	236
SetDevice	236
SetDword	236
SetMemory	237
SetWord	237
Device programming control functions and variables	237
Function AllProgOptionsDefault	238
Function ExecFunction	238
Function GangExecute	239
Function GangGetError	239
Function GangStatus	239
Function GangWaitComplete	239
Function GetBadDeviceCount	240
Function GetGoodDeviceCount	240
Function GetProgOptionBits	240
Function GetProgOptionFloat	240
Function GetProgOptionList	240
Function GetProgOptionLong	241
Function GetProgOptionString	241
Function mprintf	241
Function OpenProject	241
Function ProgOptionDefault	241
Function ReadShadowArea	241
Function SetProgOption	242
Function WriteShadowArea	242
Variable BlankCheck	243
Variable BufferStartAddr	243
Variable Checksum	243
Variable ChipEndAddr	243
Variable ChipStartAddr	243
Variable DeviceBatchSize	243
Variable DialogOnError	244
Variable GangMode	244
Variable InsertTest	244
Variable LastErrorMessage[]	244
Variable NumSites	244
Variable ReverseBytesOrder	244
Variable SerialNumber	244
Variable Signature	245
Variable VerifyAfterProgram	245
Variable VerifyAfterRead	245
Mathematical functions	245
String operation functions	246
Character operation functions	247
Functions for file and directory operation	247
Stream file functions	249
Formatted input-output functions	250

Script File Manipulation Functions.....	250
Text editor functions.....	250
Debug shell control functions.....	252
Windows operation functions and other system functions.....	253
Graphical output functions	253
I/O Stream window operation functions.....	254
Event Wait Functions.....	255
Other Various Functions.....	255
Built-in Variables by Group	256
List of Built-in Functions and Variables	256
Scripting Functions	263
fnmerge	263
Function _ff_attrib.....	264
Function _ff_date	264
Function _ff_name.....	264
Function _ff_size	265
Function _ff_time	265
Function _fullpath.....	265
Function _GetWord.....	265
Function _printfv	266
Function abs	266
Function acos	266
Function ActivateWindow.....	266
Function AddButton.....	266
Function AddrExpr.....	267
Function AddWatch.....	267
Function API	268
Function asin	268
Function atan	268
Function atof	268
Function atoi	269
Function BackSpace.....	269
Function BlockBegin.....	269
Function BlockCopy.....	269
Function BlockDelete	269
Function BlockEnd.....	270
Function BlockFastCopy.....	270
Function BlockMove.....	270
Function BlockOff.....	270
Function BlockPaste.....	270
Function CallLibraryFunction.....	270
Function ceil	271
Function chdir	271
Function CheckSum	271
Function chsize	271
Function ClearAllBreaks.....	272
Function ClearBreak.....	272
Function ClearBreaksRange.....	272
Function clearerr	272
Function ClearWindow.....	273
Function close	273
Function CloseProject.....	273
Function CloseWindow.....	273
Function cos	274

Function Cr	274
Function creat	274
Function creatnew.....	275
Function creattemp.....	275
Function CurChar.....	276
Function Curcuit	276
Function delay	276
Function DelChar.....	276
Function DelLine	277
Function difftime	277
Function DisplayText.....	277
Function DisplayTextF.....	277
Function Down	278
Function dup	278
Function dup2	278
Function Ellipse	279
Function eof	279
Function Eof	279
Function Eol	280
Function exec	280
Function ExecMenu.....	280
Function ExecScript.....	281
Function exit	281
Function ExitProgram.....	282
Function exp	282
Function Expr	282
Function fabs	282
Function fclose	282
Function fdopen	283
Function feof	283
Function ferror	284
Function fflush	284
Function fgetc	284
Function fgets	285
Function FileChanged.....	285
Function filelength.....	285
Function fileno	285
Function FillRect	286
Function findfirst	286
Function findnext	286
Function FindWindow.....	287
Function FirstWord.....	287
Function FloatExpr.....	287
Function floor	287
Function fmod	288
Function fnsplit	288
Function fopen	288
Function ForwardTill.....	289
Function ForwardTillNot.....	289
Function fprintf	289
Function fputc	290
Function fputs	290
Function FrameRect.....	290
Function fread	291

Function FreeLibrary.....	291
Function freopen	291
Function frexp	292
Function fscanf	292
Function fseek	293
Function ftell	293
Function fwrite	294
Function GetByte	294
Function getc	294
Function getcurdir.....	295
Function getcwd	295
Function getdate	295
Function getdfree.....	296
Function getdisk().....	296
Function getenv	296
Function GetFileName.....	296
Function getftime	296
Function GetLine	297
Function GetMark.....	297
Function GetMemory.....	297
Function GetScriptFileName.....	298
Function gettime	298
Function getw	299
Function GetWindowHeight.....	299
Function GetWindowWidth.....	299
Function GetWord.....	299
Function GetWord.....	300
Function GotoXY	300
Function HStep	300
Function inport	301
Function inportb	301
Function Inspect	301
Function InvertRect.....	301
Function isalnum	302
Function isalpha	302
Function isascii	302
Function isatty	302
Function isctrl	303
Function isdigit	303
Function isgraph	303
Function islower	303
Function isprint	304
Function ispunct	304
Function isspace	304
Function isupper	304
Function isxdigit	304
Function itoa	305
Function LastChar.....	305
Function LastEvent.....	305
Function LastEventInt{1...4}.....	306
Function LastString.....	306
Function LineTo	306
Function LoadDesktop.....	306
Function Left	307

Function LoadLibrary.....	307
Function LoadOptions.....	307
Function LoadProgram.....	307
Function LoadProject.....	308
Function locking	308
Function log	309
Function log10	309
Function lseek	309
Function ltoa	310
Function MaxAddr.....	310
Function memccpy.....	310
Function memchr.....	310
Function memcmp.....	311
Function memcpy.....	311
Function memicmp.....	311
Function memmove.....	312
Function memset.....	312
Function MessageBox.....	312
Function MessageBoxEx.....	312
Function MinAddr	313
Function mkdir	313
Function MoveTo	314
Function MoveWindow.....	314
Function movmem.....	314
Function open	314
Function OpenEditorWindow.....	315
Function OpenStreamWindow.....	315
Function OpenUserWindow.....	316
Function OpenWindow.....	316
Function outport	317
Function outportb.....	317
Function peek	317
Function peekb	317
Function poke	317
Function pokeb	318
Function Polyline	318
Function pow	318
Function pow10	318
Function printf	319
printf Conversion Type Characters.....	319
printf Flag Characters	320
printf Format Specifier Conventions.....	320
%e or %E Conversions.....	321
%f Conversions	321
%g or %G Conversions.....	321
%x or %X Conversions	321
Alternate Forms for printf Conversion.....	322
printf Format Specifiers	322
printf Format String	323
printf Input-size Modifiers.....	323
printf Precision Specifiers.....	323
printf Width Specifiers	325
Function pscanf	325
Function putc	326

Function putenv	326
Function putw	326
Function rand	327
Function random	327
Function randomize.....	327
Function read	327
Function Rectangle.....	328
Function RedrawScreen.....	328
Function ReloadProgram.....	328
Function RemoveButtons.....	328
Function rename	329
Function rewind	329
Function Right	329
Function rmdir	329
Function SaveData.....	330
Function SaveDesktop.....	330
Function SaveFile.....	331
Function SaveOptions.....	331
Function scanf	331
Function Search	332
Function searchpath.....	332
Function SearchReplace.....	333
Function SelectBrush.....	333
Function SelectFont.....	333
Function SelectPen.....	333
Function SetBkColor.....	334
Function SetBkMode.....	334
Function SetBreak.....	334
Function SetBreaksRange.....	334
Function SetByte	334
Function SetCaption.....	335
Function setdisk	335
Function SetDword.....	335
Function SetFileName.....	335
Function setftime	336
Function SetMark	336
Function setmem.....	336
Function SetMemory.....	337
Function setmode.....	337
Function SetPixel	337
Function SetTextColor.....	337
Function SetToolbar.....	338
Function SetUpdateMode.....	338
Function SetWindowFont.....	338
Function SetWindowSize.....	339
Function SetWindowSizeT.....	339
Function SetWord.....	339
Function sin	340
Function sprintf	340
Function sqrt	340
Function srand	341
Function sscanf	341
Function Step	341
Function Stop	341

Function strcpy	342
Function strcat	342
Function strchr	342
Function strcmp	342
Function strcmpi	343
Function strcpy	343
Function strcspn	343
Function stricmp	343
Function strlen	344
Function strlwr	344
Function strncat	344
Function strncmp	345
Function strncmpi	345
Function strncpy	345
Function strnicmp	346
Function strnset	346
Function strpbrk	346
Function strrchr	346
Function strrev	347
Function strset	347
Function strspn	347
Function strstr	347
Function strtol	348
Function strtoul	348
Function strupr	349
Function tan	349
Function tanh	349
Function tell	349
Function TerminateAllScripts	350
Function TerminateScript	350
Function Text	350
Function toascii	350
Function ToF	350
Function tolower	351
Function toupper	351
Function ultoa	351
Function unlink	351
Function unlock	352
Function Up	352
Function UpdateWindow	352
Function Wait	352
Function WaitExprChange	353
Function WaitExprTrue	353
Function WaitGetMessage	354
Function WaitMemoryAccess	354
Function WaitSendMessage	355
Function WaitStop	356
Function WaitWindowEvent	356
Function wgetchar	356
Function wgethex	357
Function wgetstring	357
Function WindowHotkey	357
Function WordLeft	358
Function WordRight	358

Function wprintf	358
Function write	358
lock	359
Variable _fmode	359
Variable ApplName.....	359
Variable BlockCol1.....	360
Variable BlockCol2.....	360
Variable BlockLine1.....	360
Variable BlockLine2.....	360
Variable BlockStatus.....	360
Variable CaseSensitive.....	361
Variable CurCol	361
Variable CurLine	361
Variable DesktopName.....	361
Variable errno	361
Variable InsertMode.....	362
Variable LastFoundString.....	362
Variable LastMemAccAddr.....	362
Variable LastMemAccAddrSpace.....	362
Variable LastMemAccLen.....	362
Variable LastMemAccType.....	362
Variable LastMessageInt.....	363
Variable LastMessageLong.....	363
Variable MainWindowHandle.....	363
Variable NumWindows.....	363
Variable RegularExpressions.....	363
Variable SelectedString.....	364
Variable SystemDir.....	364
Variable WholeWords.....	364
Variable WindowHandles.....	364
Variable WorkFieldHeight.....	364
Variable WorkFieldWidth.....	364
5 ACI Fuctions and Structures.....	365
ACI Fuctions	365
ACI_AllProgOptionsDefault.....	365
ACI_BuffersDialog.....	365
ACI_ConnectionStatus.....	366
ACI_CreateBuffer.....	367
ACI_ErrorString	367
ACI_ExecFunction.....	367
ACI_Exit	367
ACI_FileLoad	368
ACI_FileSave	368
ACI_FillLayer	368
ACI_GangStart	369
ACI_GangTerminateFunction.....	369
ACI_GetConnection.....	369
ACI_GetDevice	369
ACI_GetLayer	370
ACI_GetProgOption.....	370
ACI_GetProgrammingParams.....	371
ACI_GetStatus	371
ACI_Launch	371
ACI_LoadConfigFile.....	371

ACI_LoadFileDialog.....	372
ACI_LoadProject	373
ACI_ReadLayer	374
ACI_ReallocBuffer.....	374
ACI_SaveConfigFile.....	374
ACI_SaveFileDialog.....	374
ACI_SelectDeviceDialog.....	375
ACI_SerializationDialog.....	376
ACI_SetConnection.....	376
ACI_SetDevice	376
ACI_SetProgOption.....	376
ACI_SetProgrammingParams	377
ACI_SettingsDialog.....	377
ACI_StartFunction.....	378
ACI_TerminateFunction.....	378
ACI_WriteLayer	378
ACI Structures	378
ACI_Buffer_Params.....	378
ACI_Config_Params.....	380
ACI_Connection_Params.....	381
ACI_Device_Params.....	381
ACI_ErrorString_Params.....	381
ACI_File_Params.....	381
ACI_Function_Params.....	383
ACI_GangStart_Params.....	384
ACI_GangTerminate_Params.....	385
ACI_Launch_Params.....	385
ACI_Layer_Params.....	386
ACI_Memory_Params.....	388
ACI_ProgOption_Params.....	389
ACI_Programming_Params.....	393
ACI_ProjectParams.....	395
ACI_PStatus_Params.....	395

1 Introduction



CPI2-B1 In-System Device Programmers User's Guide

Copyright © 2017-2020, 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 written 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. ChipProg-02 stores projects in the computer memory. 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.
Standalone job (or Job)	This is the same as a project above but the ChipProg-ISP2 stores this integrated set of information that completely describes the target device, data to be programmed and other programming options and settings not in a PC memory but on the SD card inside of the programmer hardware.

Then a stored job can be launched by applying appropriate electrical signals from the ATE to the connector CONTROL.

1.2 CPI2-B1 device programmer

ChipProg-ISP2 is a family of in-system device programmers produced by Phyton, Inc. Microsystems and Development Tools. Currently this family consists of two models: a single-channel CPI2-B1 and CPI2-Gx gang device programmer. See the ChipProg-ISP2 portfolio on the www.phyton.com.

CPI2-B1 device programmers are primarily intended for use in test fixtures for programming single- and multi-board panels. For this purpose multiple CPI2-B1 units can be driven from one computer in the [gang](#)^[197] 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 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, PC, UART, and other on-chip programming interfaces.
- Extremely fast.
- Can program some devices at a long distance of up to 5m (~15ft).
- Virtually unlimited number of 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.
- 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) with SDK for developers.
- 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.
- By means of enclosed plastic brackets multiple CPI2-B1 units can be mounted on a standard EN 50022 (TS35) 35 mm DIN rail.

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.

Communication interfaces

- USB 2.0 High-speed.
- Ethernet (LAN) 100 Mbit/s.

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 350mA) 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.
- Five logic inputs for choosing one of 32 preloaded standalone jobs (projects).
- One low-current output for setting that can be used for standalone job 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).

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™ XP, 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.

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 256 standalone jobs can be stored on a built-in memory card.
- 32 of 256 standalone jobs above can be selected and launched by ATE signals.
- Special utility allows monitoring standalone activity on a computer.

1.2.4 Communication Interfaces

CPI2-B1 is equipped with two communication computer interfaces: USB 2.0 and Ethernet (LAN) 100 Mbit/s. Sockets for USB and LAN connections are located on a rear panel of the CPI2-B1 unit.

If the programmer is controlled from a graphical user interface ([GUI](#)⁴⁸), by default, the [Startup](#)³⁸ dialog prompts to connect via USB. The user may select the Ethernet radio button in the this dialog instead. If the programmer is controlled from a command line and no **ETH** (Ethernet) options is specified in the startup command line, the ChipProg-02 will establish connection with the programmer also via USB. The **ETH** command options are listed in the command line option matrix.

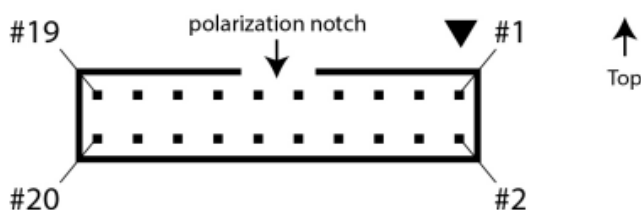
If a CPI2-B1 programmer, or a cluster comprised of multiple CPI2-B1 programmers, is controlled by Ethernet there are two options of assigning IP addresses for each device programmer: dynamic, or static IP addresses. By default, if the programmers are controlled via Ethernet, the ChipProg-02 software is set to get IP addresses dynamically distributed by your Internet router. Within a local network, a DHCP server assigns a local IP address to each device programmer connected to the LAN. However, it is possible to set unique static IP addresses for each CPI2-B1 unit.

1.2.5 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.

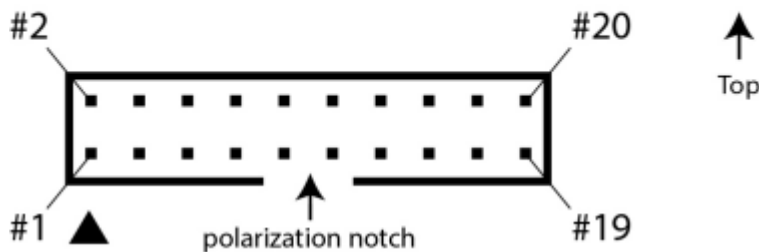
To “cut off” the target in the stand-by mode or after completion of any programming operation, CPI2-B1 programmer leaves the P1...P12 signals in a high impedance state.

1.2.6 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 the connector pin assignment and description of the signals in the matrix below. The programmer may be optionally equipped with a CPI2-ISO IO galvanic isolation board. Therefore, there are two different diagrams shown below, one for each configuration.

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	JOB_SEL0	< Input	Job select 0; active: log 1
4	START	< Input	Control signal that launches/stops programming; active: log 0
5	JOB_SEL1	< Input	Job select 1; active: log 1
6	5V_CHARGE	Output >	5V @ 500 mA sending to battery compartment for charging the battery
7	JOB_SEL2	< Input	Job select 2; active: log 1
8	5V_IN	< Input	5V input - either from external power supply or the CPI2-B1 battery
9	JOB_SEL3	< Input	Job select 3; active: log 1
10	5V_IN	< Input	5V input - either from external power supply or the CPI2-B1 battery
11	JOB_SEL4	< Input	Job 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

- **JOB_SEL[4..0]** – 5-bit selector for choosing one of 32 preloaded standalone jobs - the #0 select code is 000000, the #4 - 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 SA job 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](#)¹³².

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	JOB_SEL0	< Input	Optically isolated job select 0; active: log 1
4	START	< Input	Optically isolated control signal that launches/stops programming; active: log 0
5	JOB_SEL1	< Input	Optically isolated job select 1; active: log 1
6	V_ISO	Output >	Optically isolated 5V @ 10 mA max
7	JOB_SEL2	< Input	Optically isolated job select 2; active: log 1
8	NC	Not connected	Not connected
9	JOB_SEL3	< Input	Optically isolated job select 3; active: log 1
10	NC	Not connected	Not connected
11	JOB_SEL4	< Input	Optically isolated job 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

- **JOB_SEL[4..0]** – 5-bit selector for choosing one of 32 preloaded standalone jobs - the #0 select code is 000000, the #4 - 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 job 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](#)^[132].

1.2.7 Single- and Gang-site programming

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

- **Single-programming** mode - intended for programming one target device at a time by means of one CPI2-B1 programmer.
- **Gang-programming** mode - intended 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](#)^[38] dialog by checking and unchecking the **Gang Mode** checkbox.

The software enables control of one, specific, CPI2-B1 device programmer within a cluster of multiple programmers driven from one PC. In this case, the user the programmers by serial number. This allows the user 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 target device type may be selected for all programmers controlled by one instance of the ChipProg-02 program;
2. In the **Gang-programming** mode all programmers controlled by one instance of the ChipProg-02 program share the same data buffer;
3. Only the [Auto Programming](#)^[108] 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](#)^[108] commands by removing unwanted commands and leaving the single one required for the purpose.

By running several instances 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.

2 Installation and Launching

This chapter covers the following topics.

How to install the CPI2-B1 [hardware](#)^[29]

How to install the ChipProg-02 [software](#)^[32]

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](#)^[48] and [Operating ChipProg programmers](#)^[194] 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](#)^[90]. 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](#)^[48] 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](#)^[90] information box or on a sticker on the CPI2-B1 bottom shell.
- Software version number taken from the [About](#)^[90] 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

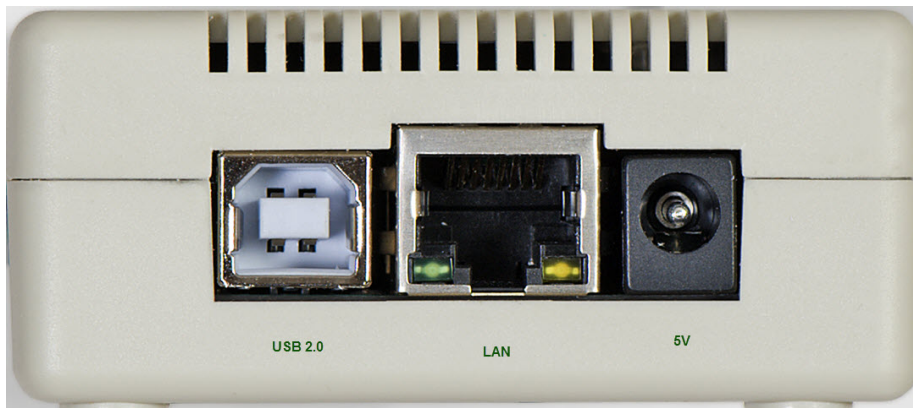
Tel: 1-718-259-3191

Fax: 1-718-259-1539

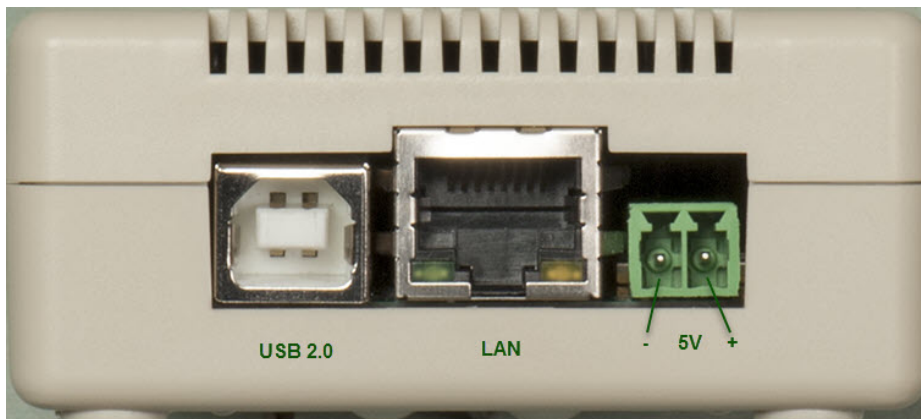
2.2 Hardware installation

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

Version with a coaxial 5V power plug (produced in 2015-2018 years)



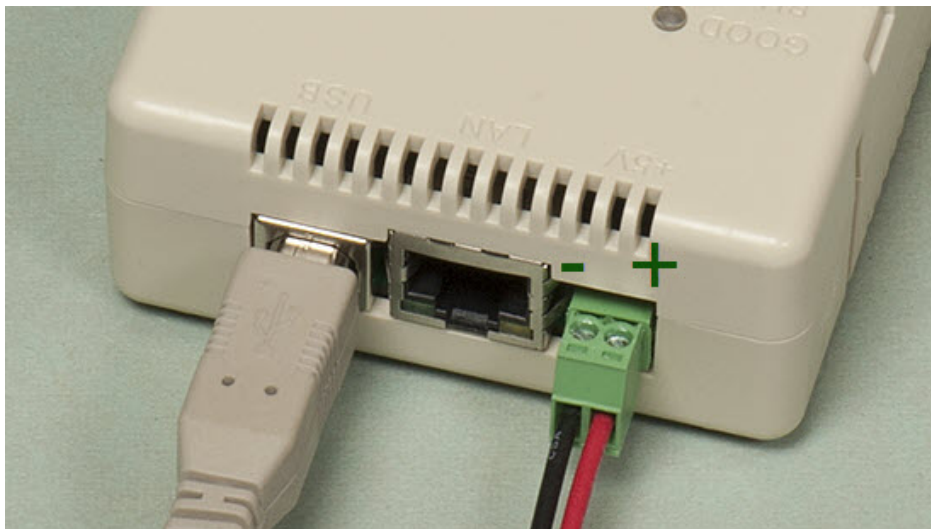
Version with a latching 5V power plug (in production beginning 2019)



Powering the programmer

A power adapter is not included into the CPI2-B1 kit. If the CPI2-B1 is controlled via a USB 2.0 port, it may get enough power via a USB port and therefore powering the programmer from an external supply is not mandatory. However, powering the programmer from an external 5V power adapter ensures more stable programming operations. Driving a CPI2-B1 device programmer via Ethernet (LAN) port always requires use of an external 5V power supply.

To power a CPI2-B1 device programmer, use any regulated 5V/500mA+ adapter. In 2015-2018 Phyton produced CPI2-B1 with coaxial power plugs (2.1 mm, center positive). Beginning 2019 Phyton produces device programmers with latching rectangular connectors that insure more reliable powering of the CPI2-B1 units that is important for use in production environment. Complimentary female terminal block plug OSTTJ0211530 is included into the CPI2-B1 kit. This block is available for purchasing from Digi-Key (p/n ED10554-ND) and Farnell Element14 corporations.

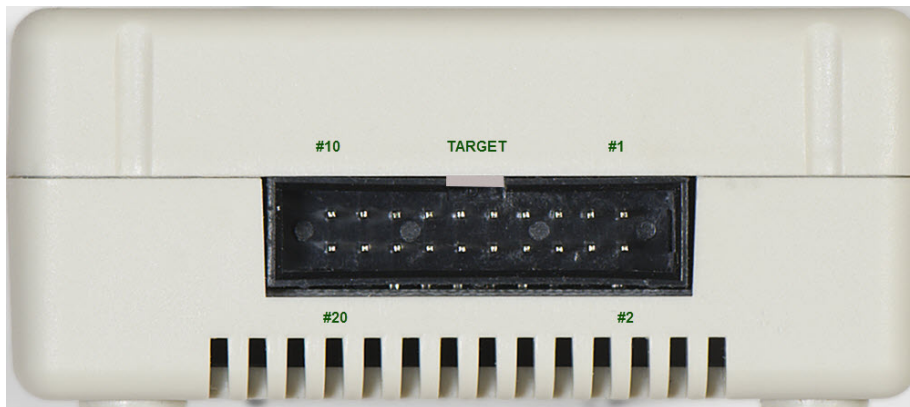


Connecting to a computer

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 not to 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. Do not use passive USB hubs.

Connecting to the target

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



On the picture above a polarization notch of the male connector is situated on the top edge of the connector.

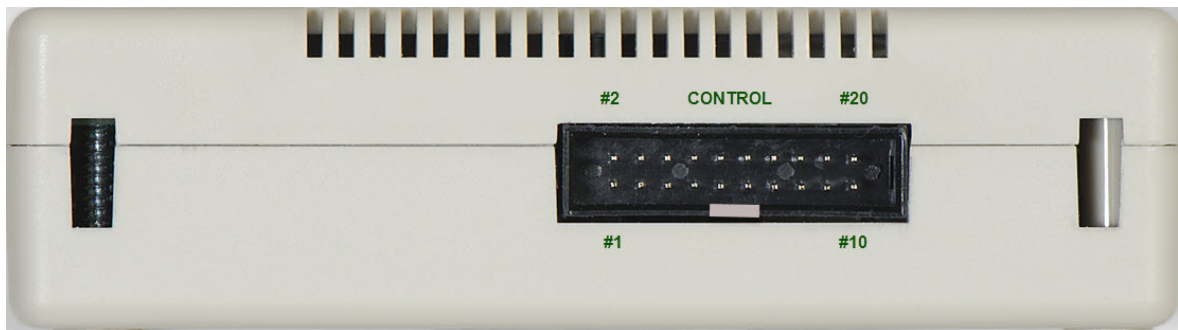
A ribbon cable with both-end mounted 20-pin headers is included in the CPI2-B1 kit. This cable is intended for connecting a CPI2-B1 device programmer to the target device (board) in accordance to the device-specific connection diagram. After a target device was selected in the programmer GUI, the diagram is also accessible by clicking the **Connection to the target device** link in the [Device Information](#) ⁹²⁾ window. Refer to the [Connector TARGET](#) ²²⁾ 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 ensure 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 the programmer to crash or unstable functioning!

Connecting to ATE controls

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



On the picture above a polarization notch of the male connector is situated on the bottom edge of the connector. To control a CPI2-B1 device programmer from your test fixture or other ATE use the

CONTROL port. The CPI2-B1 kit does not include a cable with a 20-pin header to connect this port. Refer to the **CONTROL** connector pinout to customize connection to your ATE.

Mechanical mounting

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

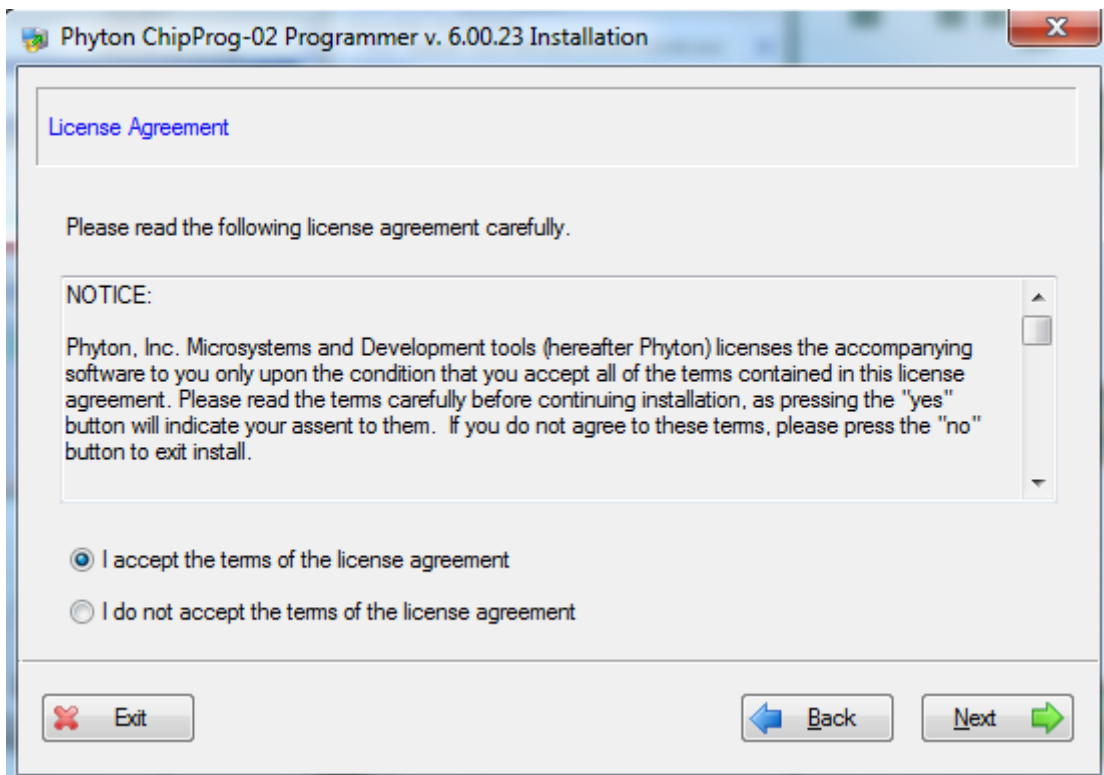
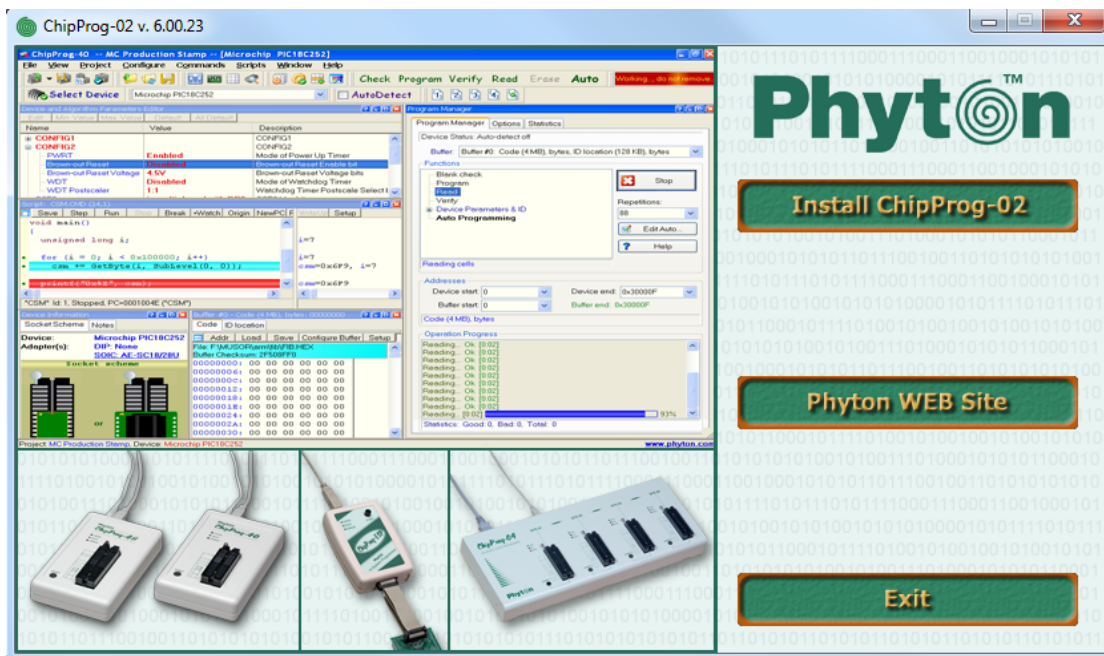
2.3 System Requirements

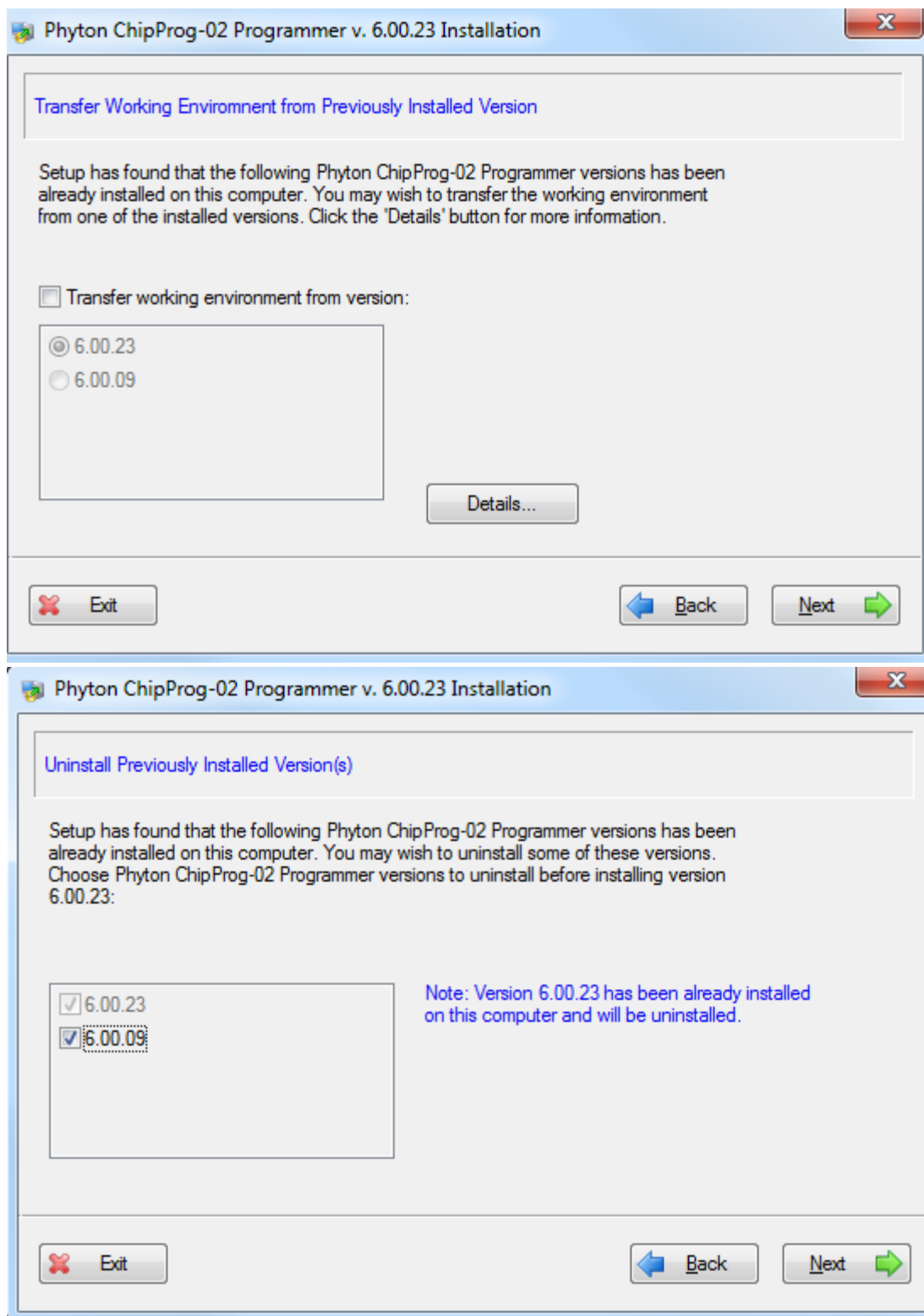
To run ChipProg-02 and control a CPI2-B1 device programmer, you need a personal computer (PC) with the following components:

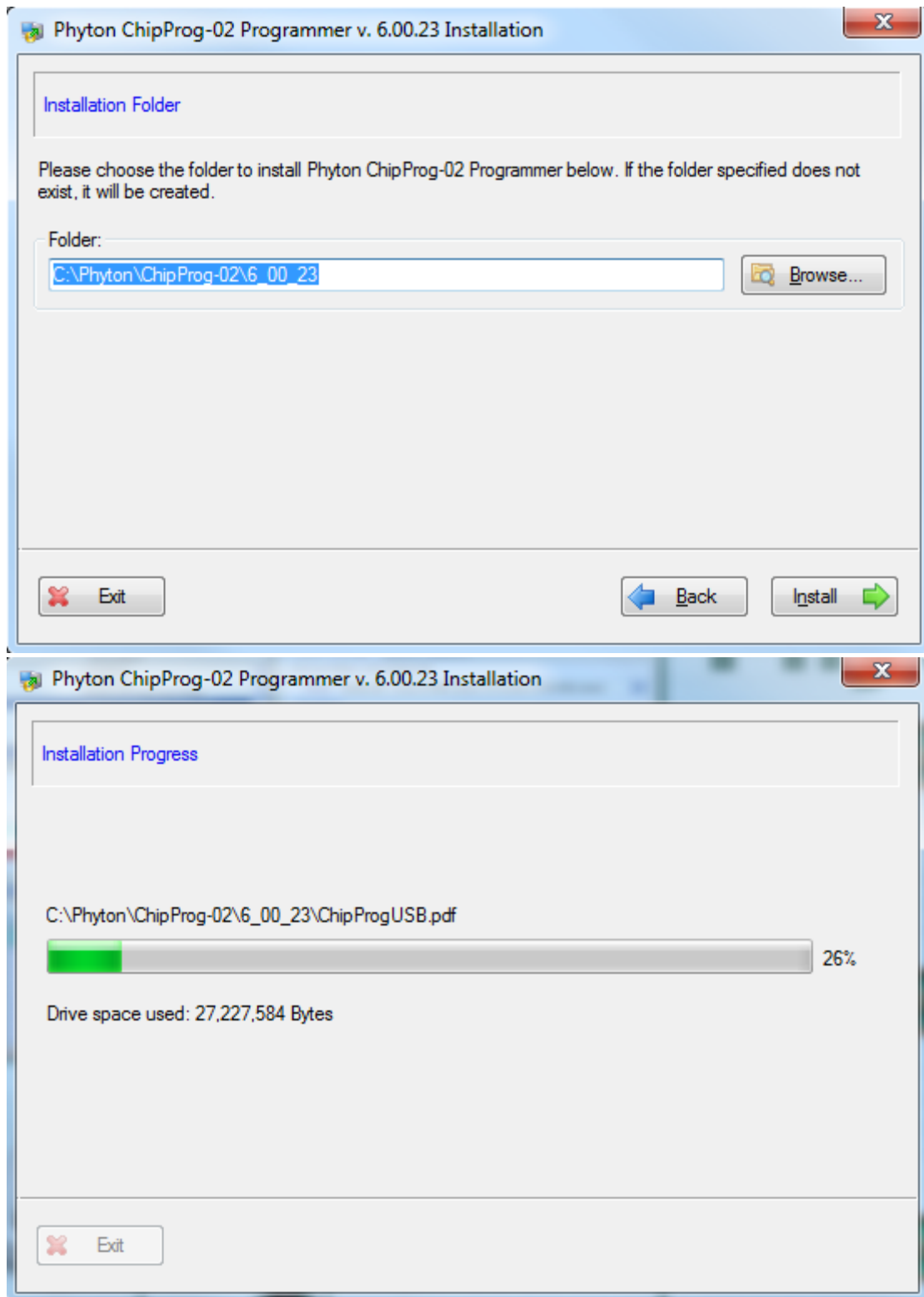
- Pentium-V or higher CPU.
- Microsoft Windows XP, 7, 8 or 10 operating system.
- A hard drive with at least 200MB of free space.
- In case of use the USB communication: at least one USB 2.0 port.
- In case of use the Ethernet communication: at least one LAN port or an Ethernet router with the Dynamic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

2.4 Software Installation

Since beginning of 2020 Phyton does not supply device programmer kits with CD ROMs with the ChipProg-02 software. Users should download the latest software version from the <https://phyton.com/support/updates> webpage. To begin the software installation launch the **cp-02.exe** self-extracting executable file. Or, if you have a CD ROM, insert it into a CD drive on your PC. When installer launches, 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 folder

At the end of the software installation 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

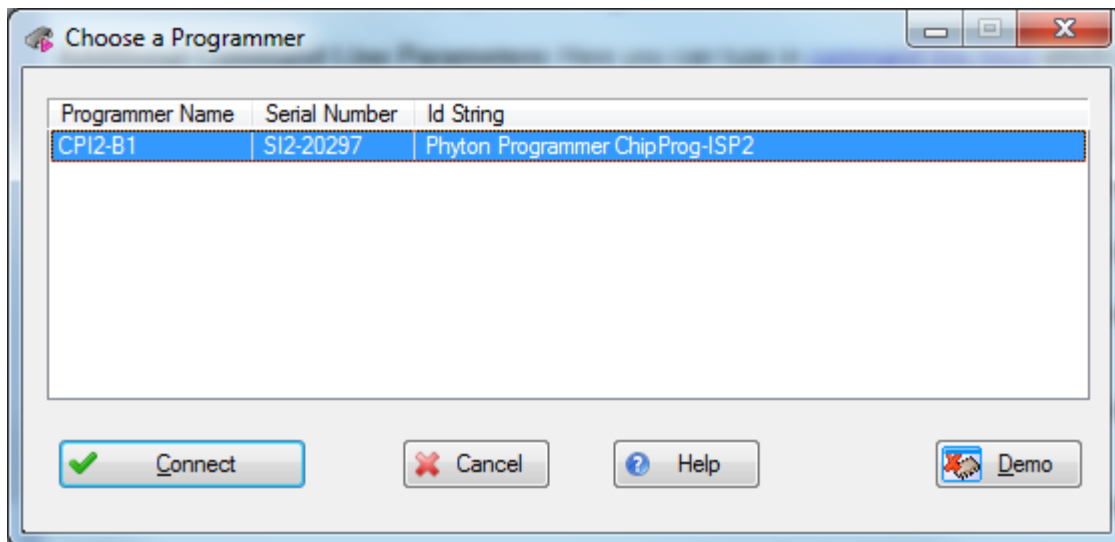
The first shortcut - **Phyton ChipProg-02** opens the [setup wizard](#)^[38] ending with the [startup dialog](#)^[38]. In this dialog you can create multiple shortcuts for launching the device programmer(s) with different startup settings. All of them are accessible from the Phyton ChipProg-02 folder.

2.5 Launching device programmers

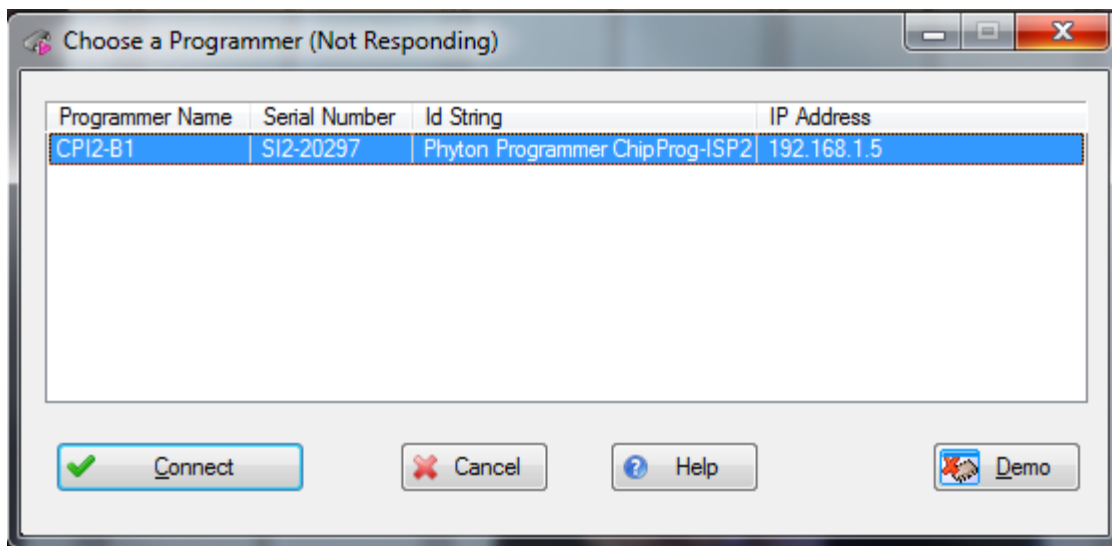
Launching a single CPI2-B1 device programmer.

By default, a single CPI2-B1 programmer starts in the [Single Programming](#)^[26] 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](#)^[38], 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 in the [Startup Dialog](#)^[38]. 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:



When a CPI2-B1 device programmer is controlled via Ethernet, the default behavior is that it gets an IP addresses dynamically changed by a LAN router. However, it is possible to set [static IP address](#) ⁷⁴.

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](#) ³⁸ or add the **-GANG** key to the [command line](#) ¹²⁰.

The number of CPI2-B1 device programmers driven from one computer in gang-programming mode is limited to 72 units. Each single CPI2-B1 unit has its own unique serial number. Before operating with multiple CPI2-B1 programmers as a gang cluster you must assign **Site Numbers** from 1 to N to serial numbers of the programmers' serial numbers. There are two ways to do this: a) directly specifying a chain of CPI2-B1 serial numbers in the command line or b) manually.

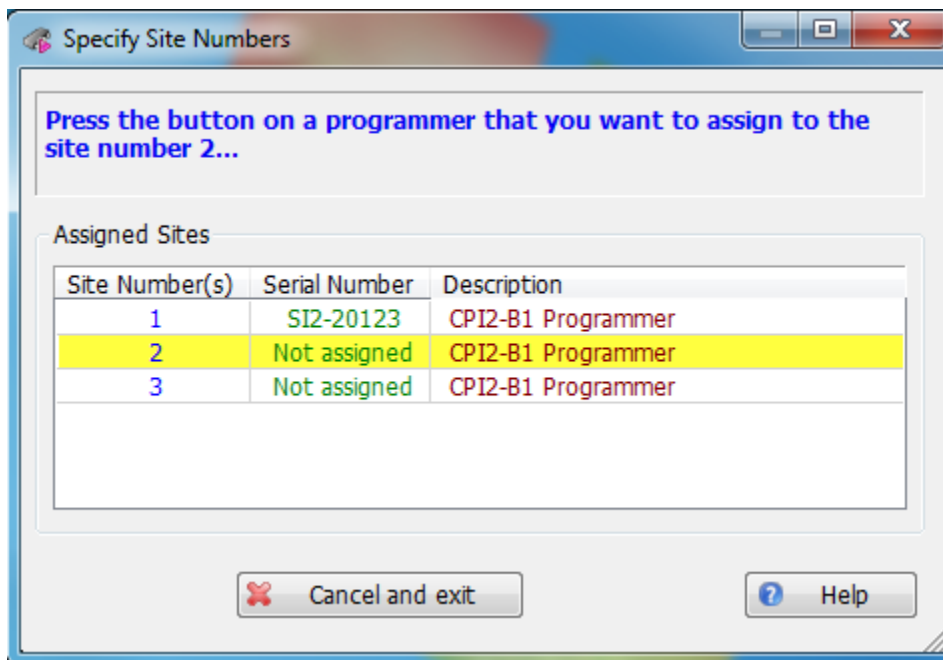
Specifying site numbers in [command line](#) ¹²⁰.

In order to specify the site numbers in the command line, use the **-GANG** key followed by '#' sign with a list of serial numbers separated by semicolons. The application will wait until the number of connected single-site programmers matches the number of serial numbers in the list. Once all programmers are connected, the software 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 assignment of site numbers.

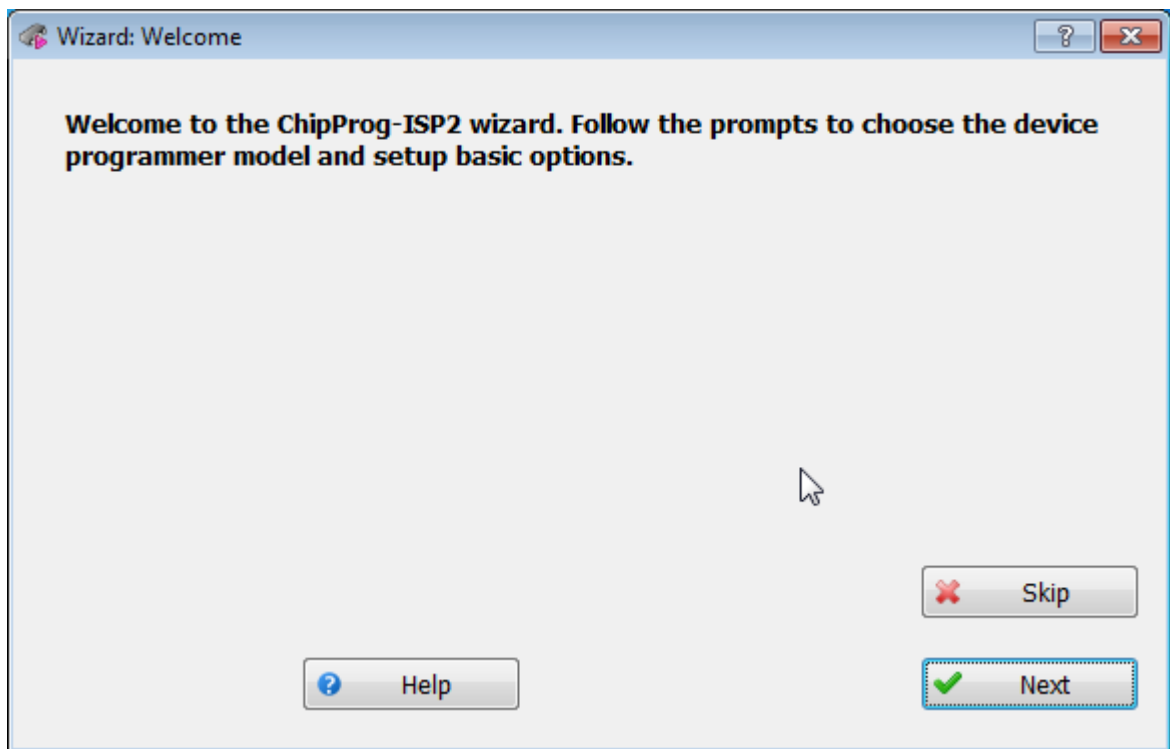
If the **-GANG** key is not followed with CPI2-B1 serial numbers you can assign site numbers manually. Once Windows has detected multiple CPI2-B1 programmers connected to a PC, the ChipProg-02 opens the **Specify Site Numbers** dialog. It prompts the user for assignment of numbers to individual programmers (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 the user to assign the site #2 to another programmer and continue this way until all programmers are assigned a sequence number.

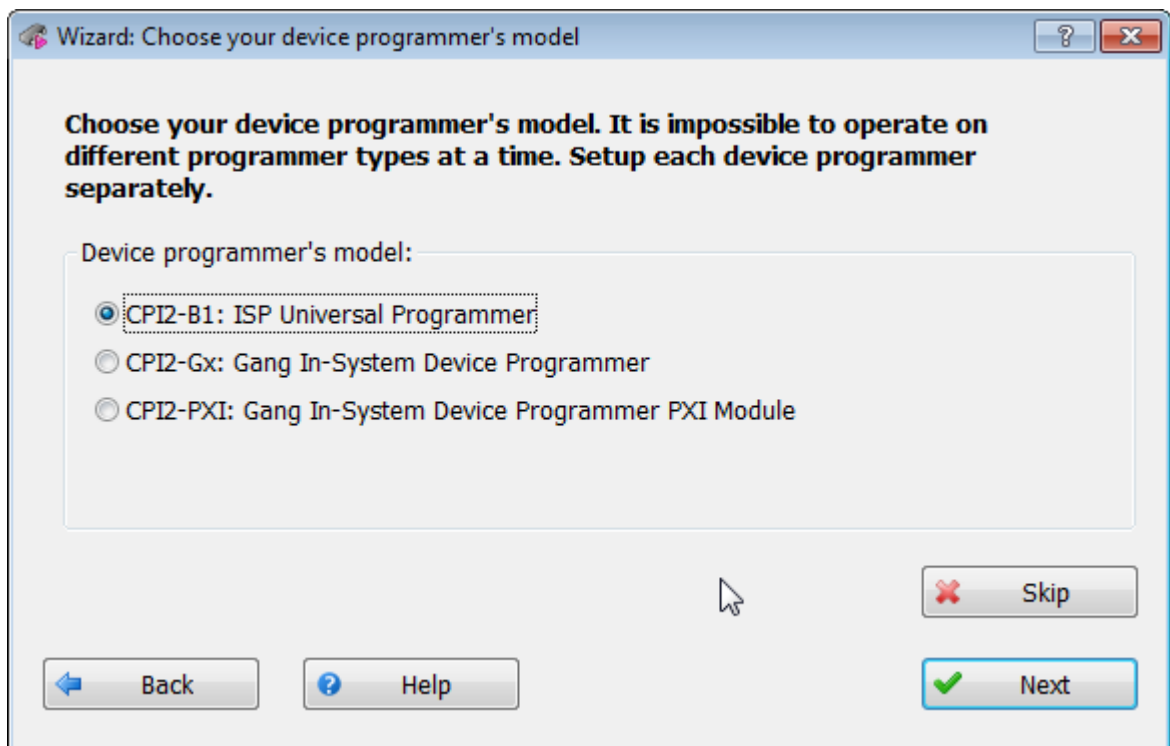


2.6 Setup Wizard and Startup Dialog

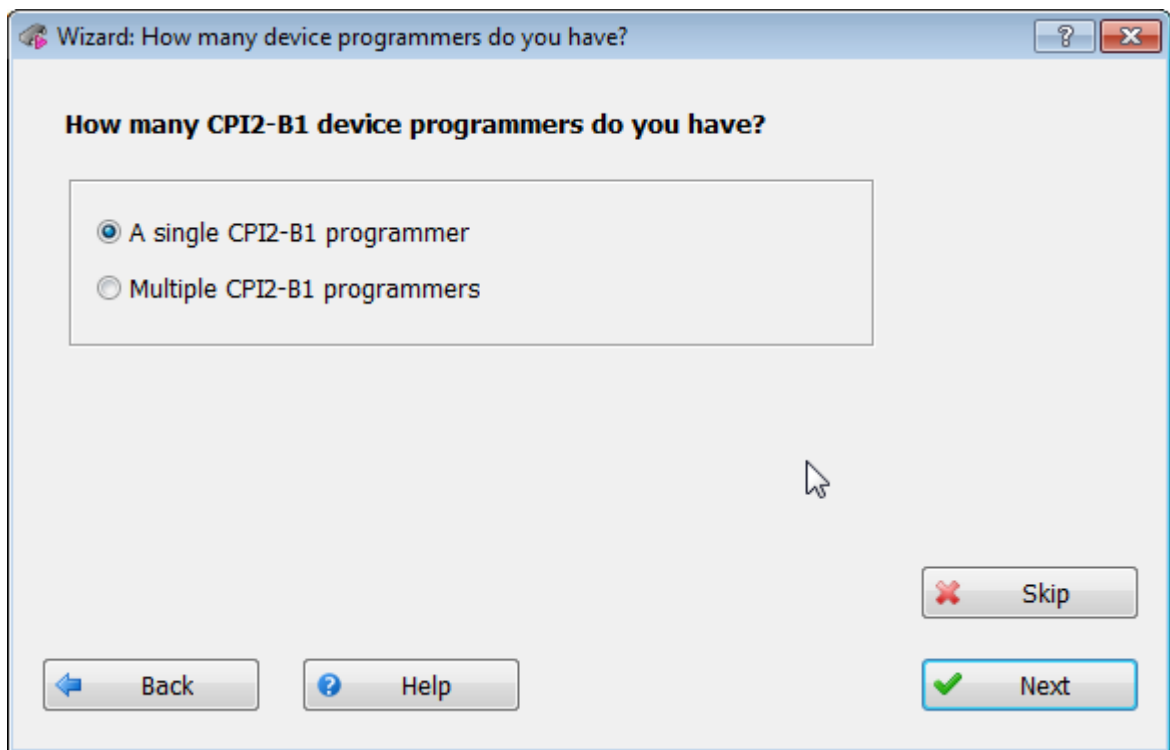
If you launch the programmer first time, the program opens the ChipProg-ISP2 setup wizard welcome page:



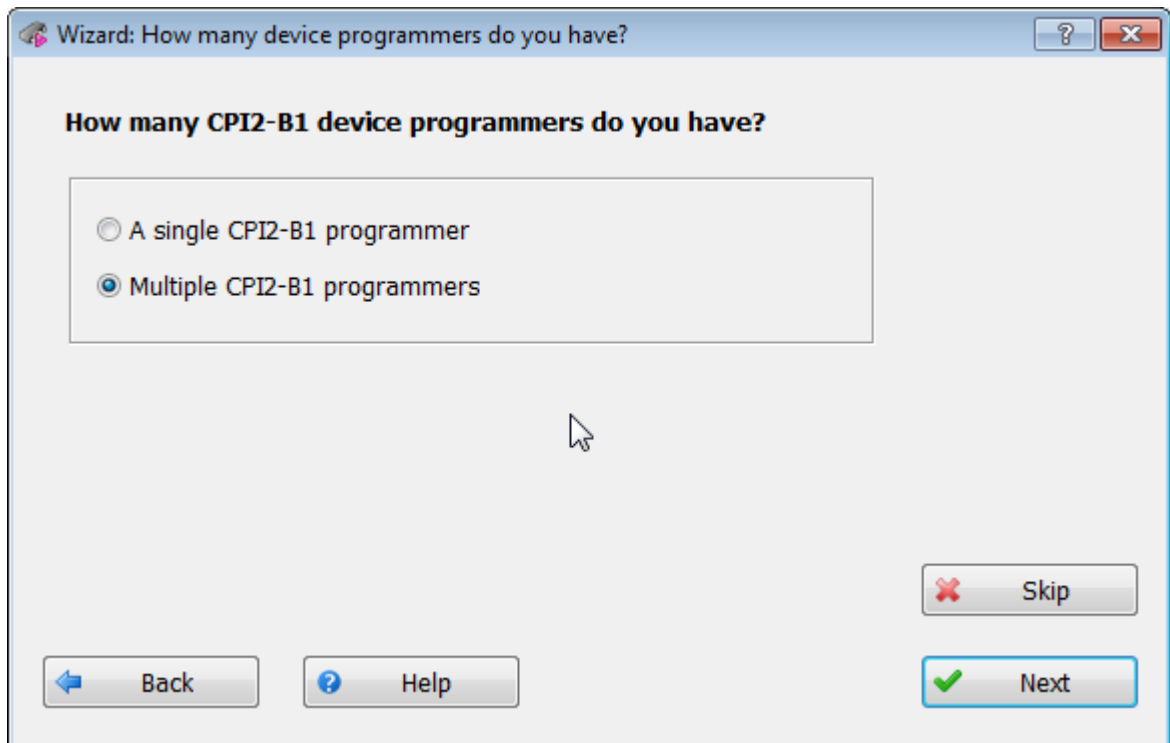
On the next step the wizard prompts you to select the device programmer model:



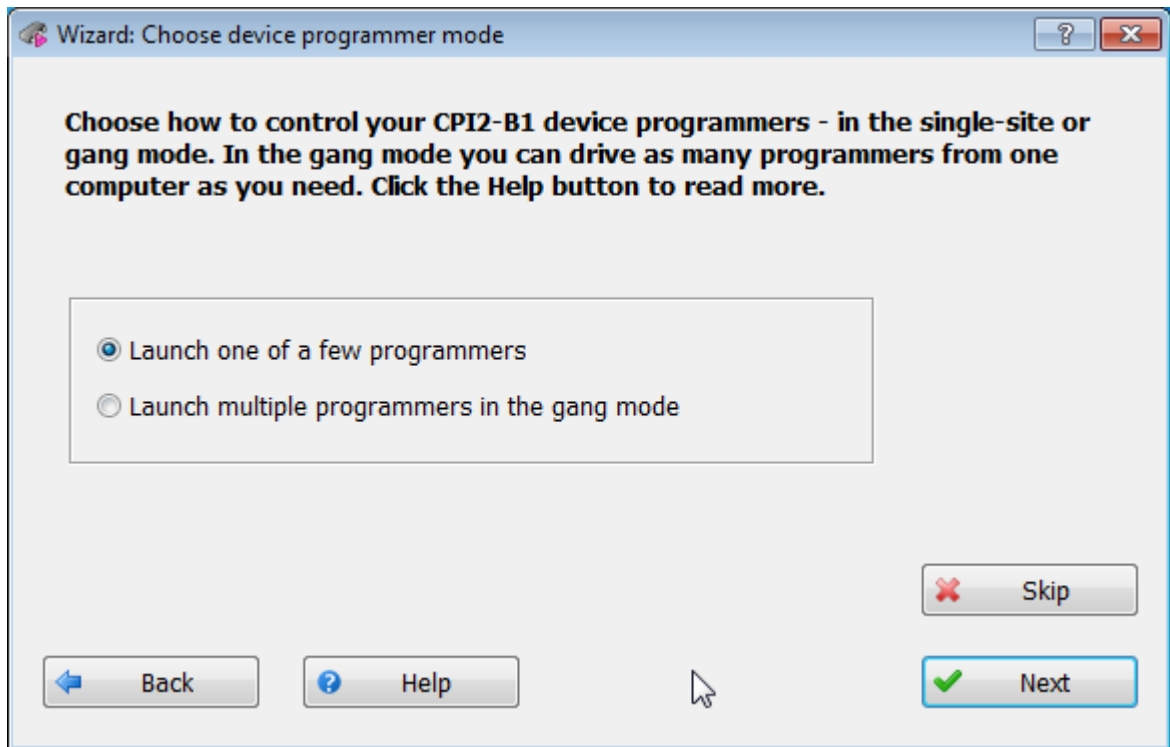
Then opt whether you have just one or multiple CPI2-B1 device programmers controlled by your computer. Let us assume you selected launching a single CPI2-B1:



then, after clicking the **Next** button, you will need to choose the interface type - USB or LAN that will open the **Startup** dialog. But, if you select launching multiple CPI2-B1 device programmers:

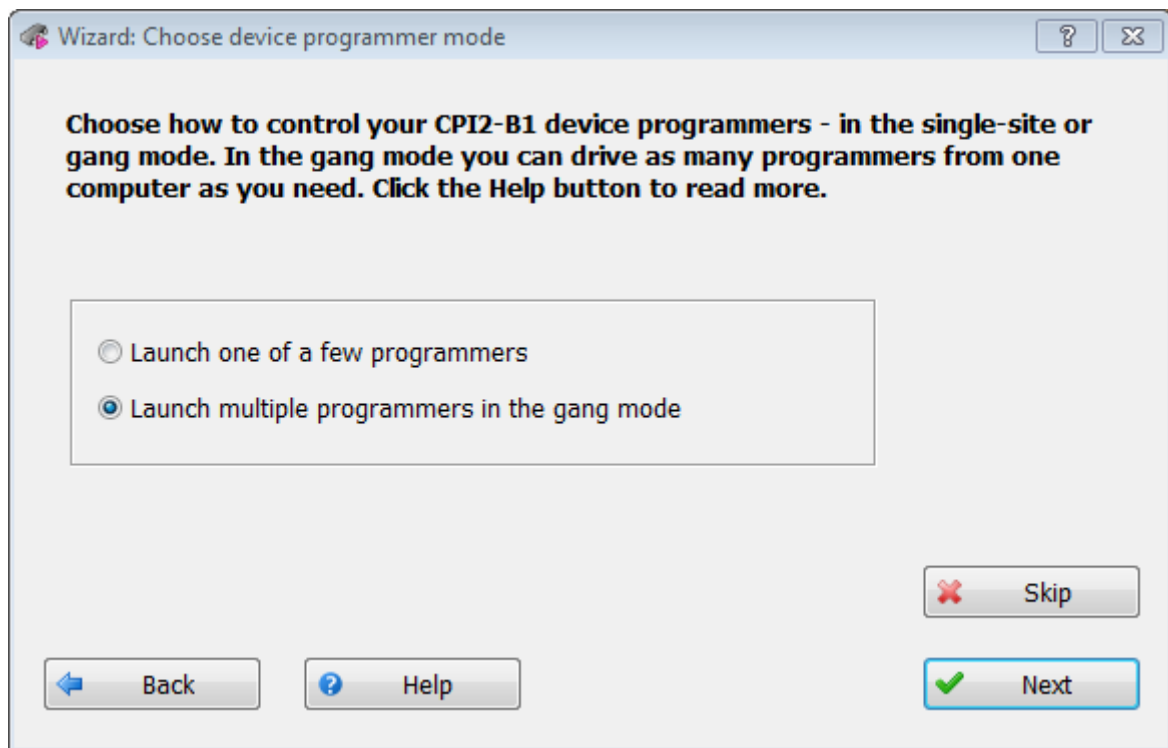


and click the **Next** button then the wizard prompts you to opt whether you want to launch just one of multiple CPI2-B1 programmers connected to your PC or to drive a cluster of them in the [gang](#)¹⁹⁷ mode:

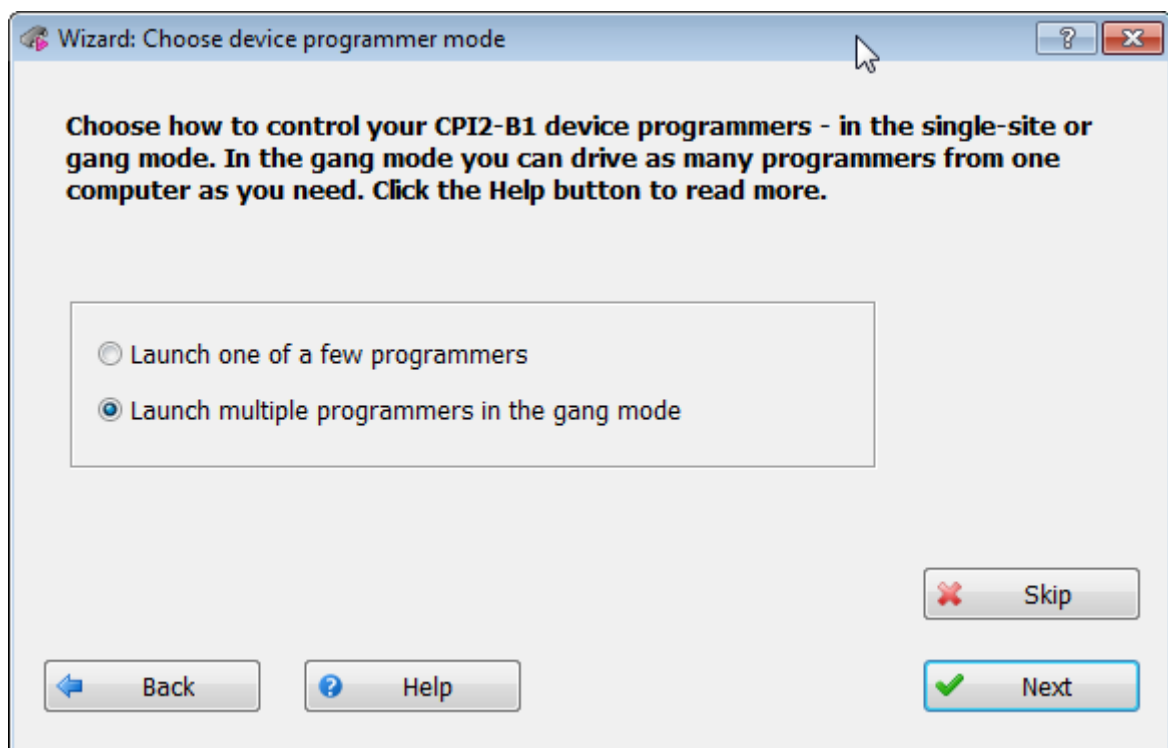


If you selected launching one CPI2-B1 and click the **Next** button, the wizard prompts you to select the interface to a PC and completes with opening the **Startup** dialog.

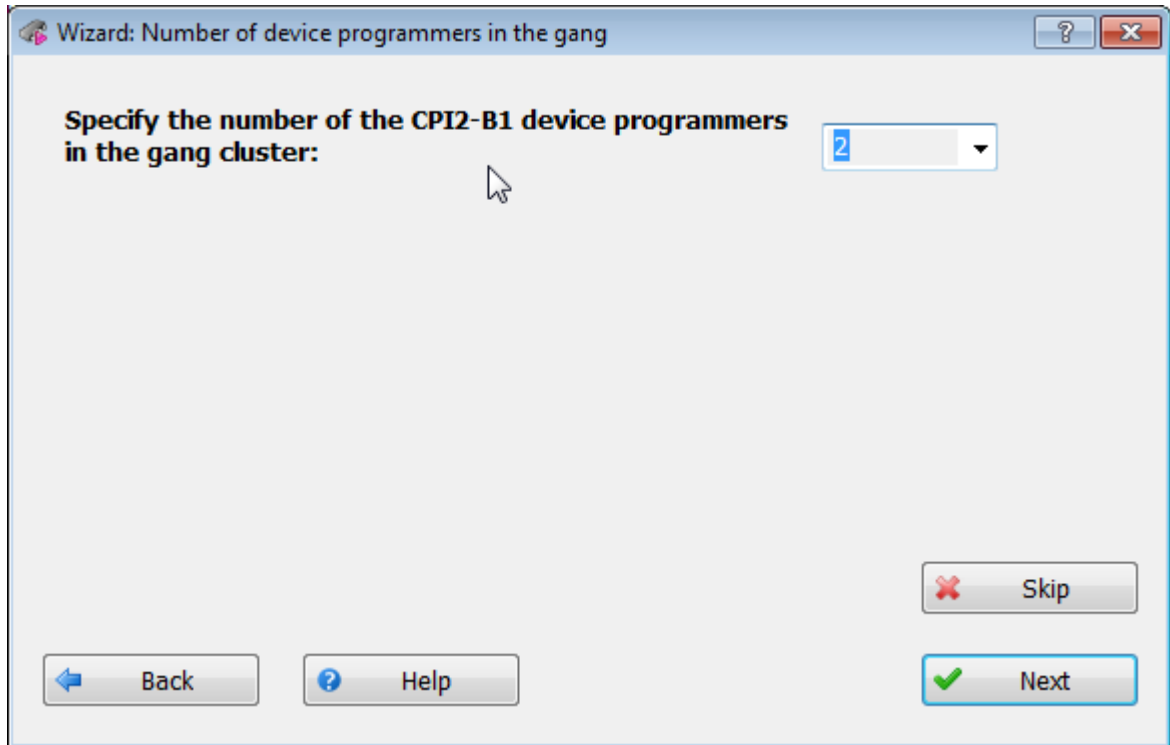
If you selected driving multiple CPI2-B1 programmers in the [gang](#)¹⁹⁷ mode:



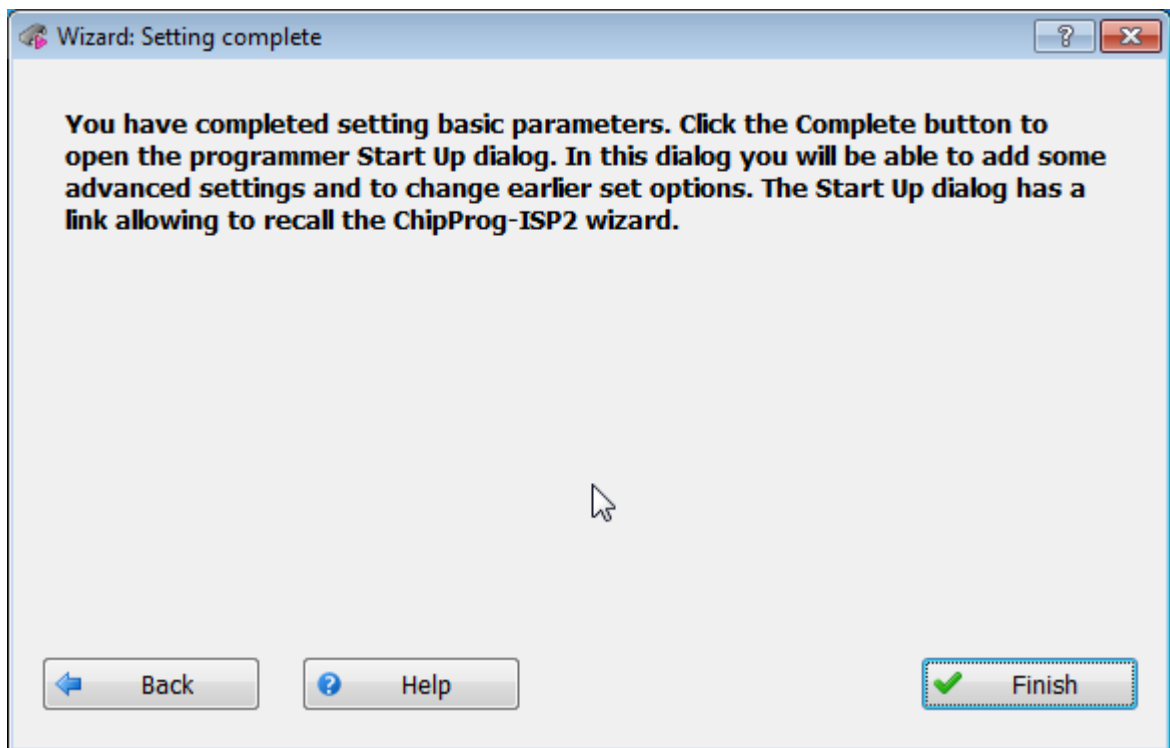
then on the next step the wizard prompts you to specify the number of CPI2-B1 programmers that you would like to drive as a gang cluster. Specify here an actual number of CPI2-B1 programmers controlled by your PC:



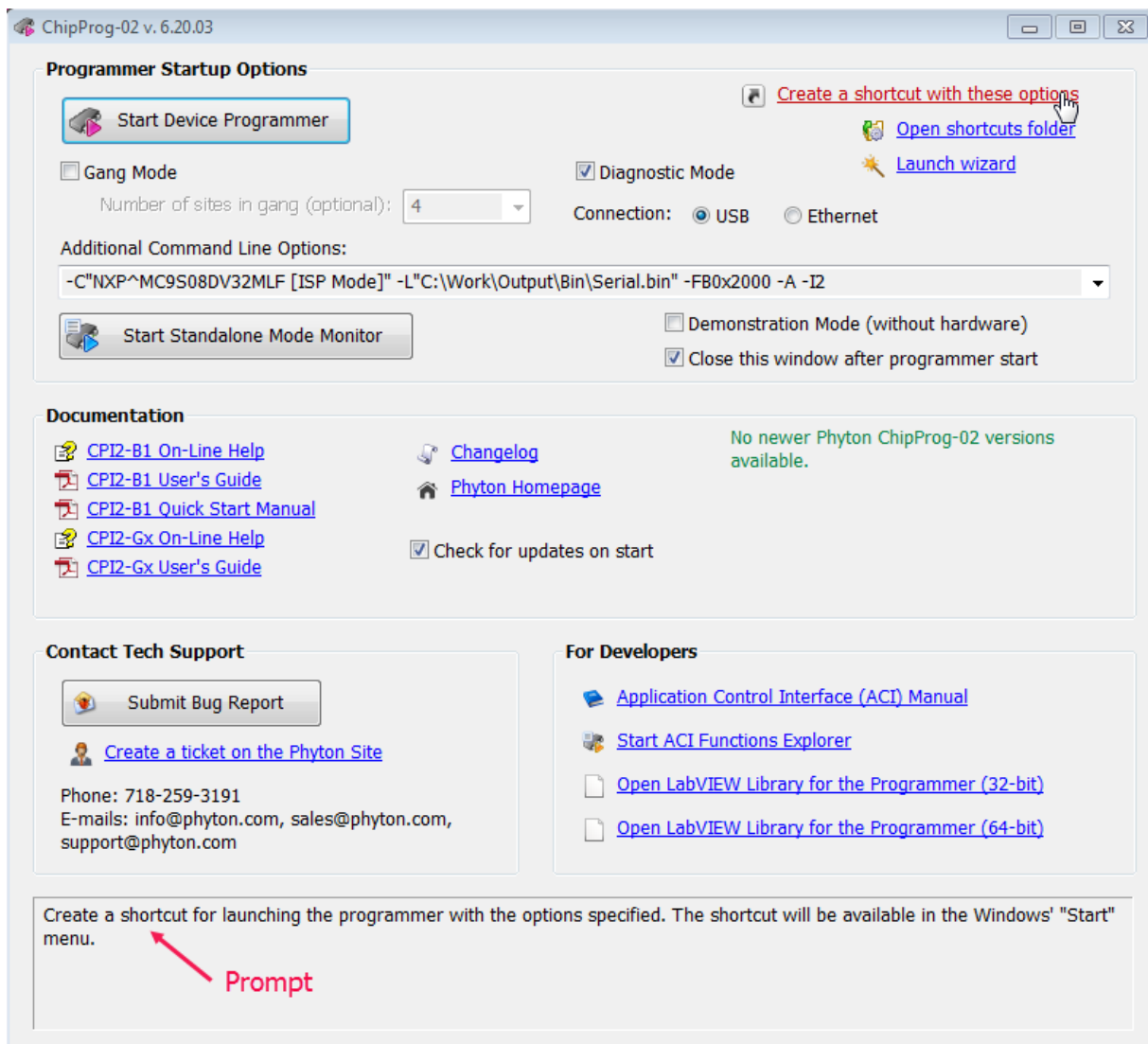
If you leave this box blank, the software will scan the computer USB and LAN ports trying to detect all CPI2-B1 programmers which may be connected to the computer. This will significantly increase the time of polling. After clicking the **Next** button in the dialog above, the wizard will display two communication interface options:



The same prompt completes setting for all other CPI2-B1 configuration options above. After choosing the communication interface, the wizard issues the final prompt:



By clicking the **Finish** button above you open the **Startup** dialog that displays all the settings made in the wizard. This dialog enables to enter some additional settings. The dialog window is divided in several zones: **Program Startup Options, Documentation, Contact Technical Support, For Developers**. The very bottom field displays prompts for the dialog widget pointed to a mouse cursor. In 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 Additional [Command Line Options](#).¹²⁰ See a description of the **-ETH** key and associated parameters (IP addresses, etc.)

Gang Mode¹⁹⁷: 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 gang¹²⁰: In this field you may optionally specify an actual number of CPI2-B1 device programmers to be controlled in the [gang](#)¹⁹⁷ mode.

Diagnostic Mode: This option enables/disables tracing programming operations - i.e. collecting the trace to the **UPROG.LOG** file located in the folder where the the programmer software had been installed. This **UPROG.LOG** file can be shared with Phyton Technical Support for remote

troubleshooting. By default, the **Diagnostic Mode** box is checked and a running programmer permanently updates the diagnostic information into the **UPROG.LOG** file. This slightly slows down a target device programming. If the programming speed is extremely important, a user may uncheck this box. In this case the **UPROG.LOG** remains empty.

Additional Command Line Parameters: Here you can type in [command line options](#)^[120], 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](#)^[132].

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 enables users 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 set of tools for those who develop [applications](#)^[158] for CPI2 device programmer control.

3 Control Interfaces

CPI2-B1 device programmers can be controlled by an operator in one of the **Computer Controlled** modes or in the [Standalone Mode](#)^[132] mode controlled remotely by Automatic Test Equipment (ATE) .

Computer controls include the following:

- Full-capable Graphic User Interface ([GUI](#)^[48]),
- Simplified User Interface ([SUI](#)^[113]),
- [Command Line](#)^[120], [On-the-Fly](#)^[126] control

- Application Control Interface ([ACI](#)^[158])

First three methods above are described in this chapter, the ACI is described in a separate chapter.

The [Standalone](#)^[132] control mode is also described in a separate chapter.

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](#)^[108] 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](#)^[53] any number of projects for programming a variety of devices and store them in the [project repository](#)^[56]. When needed, these files can be loaded and used just by a mouse-click, or by including a project name on [command line](#)^[120]. 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 ^[53]
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 ^[57] - Dialog Select Device ^[58] ; Window Program Manager - tab Options ^[109] Windows Device and Algorithm Parameters Editor ^[93]
Buffers	Buffer name; Buffer size; Default fill value; Swap file settings.	Menu Configure ^[57] – sub menu Buffers ^[61] ; Window Buffer – toolbar; Dialog Buffer Configuration ^[97] ; Window Buffer – toolbar; Dialog Memory Dump Windows Setup ^[98]
Serialization, Check sum, Log files	Algorithm for programming serial numbers; Custom signature patterns; Algorithm of the check sum calculation; Check sum formats;	Menu Configure ^[57] – tabs of the sub menu Serialization, Check sum, Log files ^[63]

Option group	Project options	Where to set up...
	Parameters and locations of log files to be saved.	
Actions on events	Actions triggered by certain events, issuing error messages and sounds, logging results.	Menu Configure ^[57] – sub menu Preferences ^[78]
Graphical User Interface	Screen configuration, fonts and colors of windows, key mappings, messages and miscellaneous settings.	Menu Configure ^[57] – sub menu Environment ^[79]
Statistics	Number of chips to be programmed and related settings.	Window Program Manager - tab Statistics ^[111]

You can create, edit and save projects within the CPI2-B1 **Graphical User Interface** - read about the [Project Menu](#)^[52] 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](#)^[52] 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](#)^[92].
- [Menus](#)^[50] - global and local.
- [Toolbars](#)^[49] - global and local.
- Dialogs.
- [Hot Keys](#)^[81].
- Context-sensitive [help prompts](#)^[90].

The GUI features [several useful additions](#)^[48] designed specifically for the CPI2-B1 operations.

To make your using ChipProg-02 program easier we highly recommend you read the [Menus](#)^[50] and [Windows](#)^[92] chapters in full. You will be able to use the CPI2-B1 device programmers much more effectively.

3.2.1 User Interface Overview

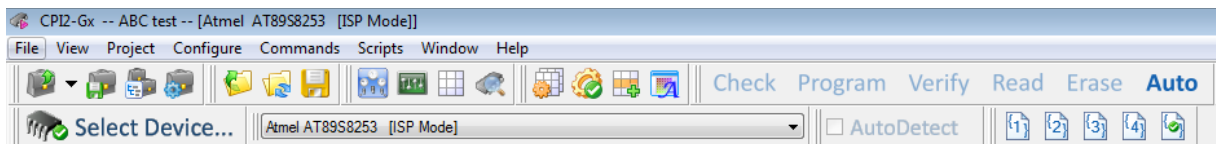
ChipProg-02 features standard Windows interface with several useful additions.

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](#)^[82] 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

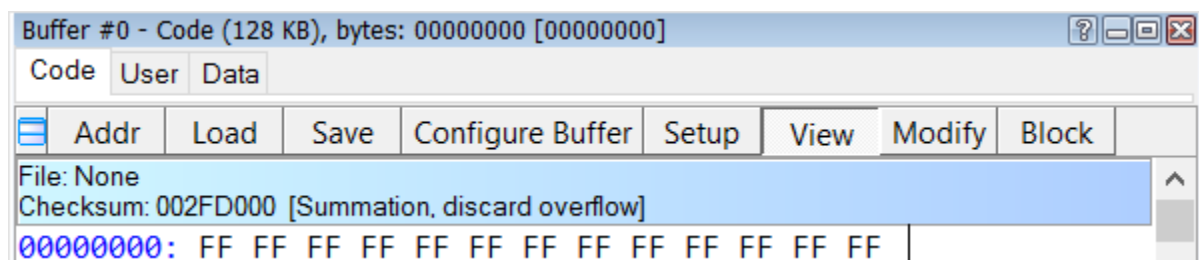
The ChipProg-02 program shows several toolbars at the top of the main window (see below).



The topmost toolbar (right under the CPI2-B1 main window title) includes the [Main menu](#)^[50] 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](#)^[57], [The Environment dialog](#)^[79], [Toolbar](#)^[82].

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](#)^[108] - is available for [gang programmers](#)^[197], 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](#)^[95] 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.

<u>Button</u>	<u>Command</u>	<u>Description</u>
	Load ...	Opens the Load file ^[102] 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 ^[104] dialog.
	Configuration Files	Gives access to operations with configuration files ^[52] .
	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](#) ^[51] [menu](#) ^[51]. 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.

<u>Button</u>	<u>Command</u>	<u>Description</u>
	Program Manager	Opens the Program Manager ^[106] dialog.
	Device and Algorithm Parameters	Opens the Device and Algorithm Parameters ^[93] dialog.
	Buffer Dump	Opens the Buffer ^[95] dialog.
	Memory Card Window	Opens the Memory Card ^[132] window
	Device Information	Opens the Device Information ^[92] dialog.
	Console	Opens the Console ^[104] dialog.

3.2.3.3 The Project Menu

This menu contains commands for working with [projects](#) ^[47].

Button	Command	Description
	New	Opens the Project Options ^[53] dialog.
	Open	Opens the Open Project ^[54] 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 ^[56] dialog for storing current project in project data base for convenient handling.
	Options	Opens the Project Options ^[53] 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](#)^[52] 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 ^[102] 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 ¹⁰² 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

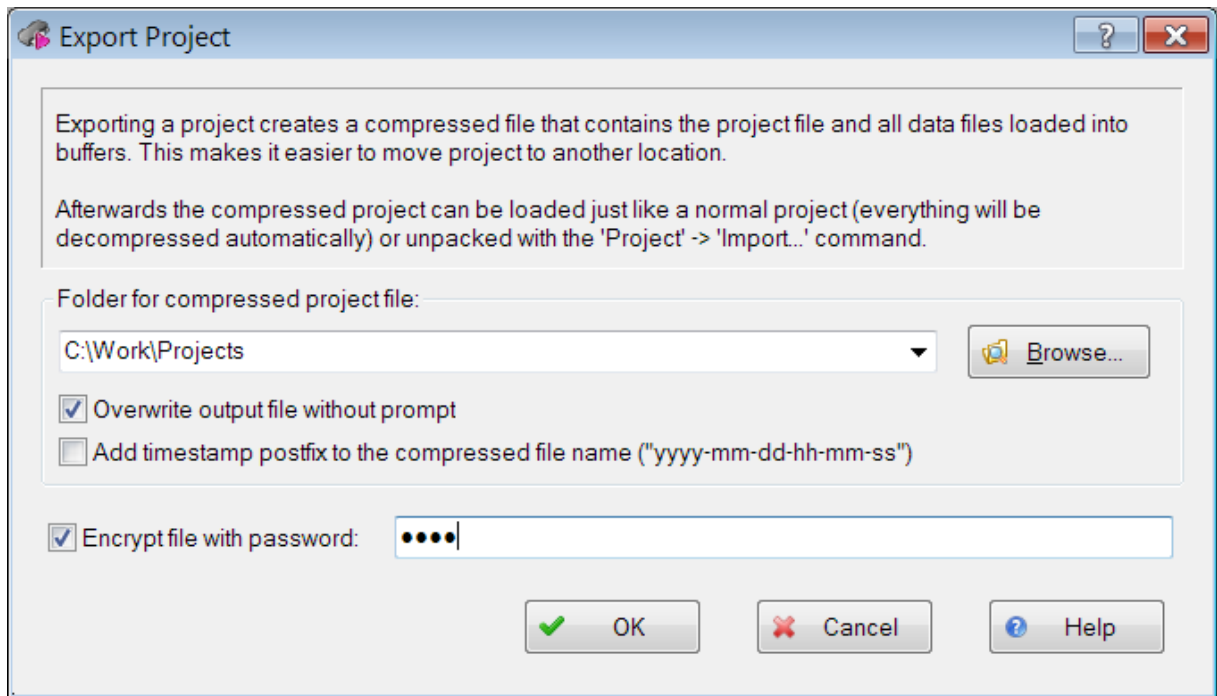
This dialog is used to open a previously created project.

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](#) ⁴⁷ 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.

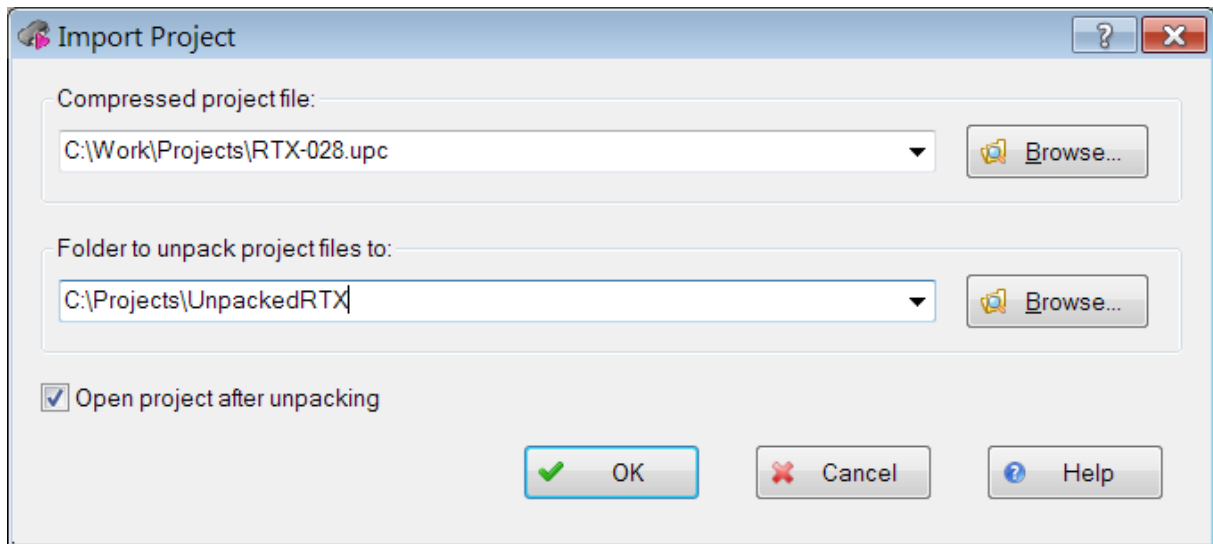
Checking the Overwrite output file without prompt box prevents casual spoiling of a previously stored compressed project.

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](#)^[54] command as well as "simple" .upp project files. When you use the [Open Project](#)^[54] command from the [Project](#)^[52] 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.

3.2.3.3.4 Project Repository

The **Project Repository** command of the [Project menu](#)^[52] 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 ⁵⁴ 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.

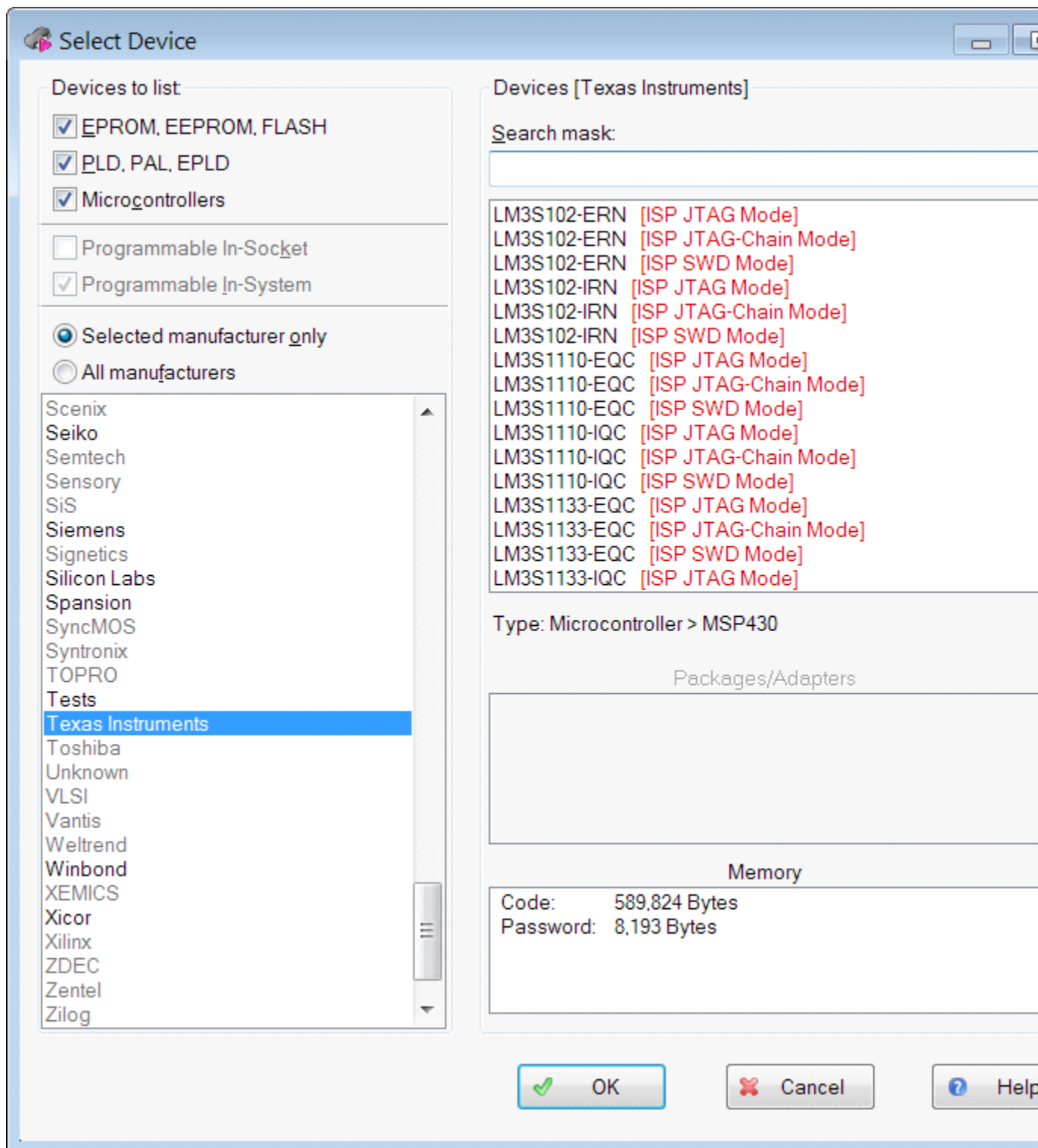
3.2.3.4 The Configure Menu

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

<u>Button</u>	<u>Command</u>	<u>Hot key</u>	<u>Description</u>
 Select Device...	Select device	F3	Opens the Select Device ^[58] dialog.
	Device selection history	Alt+F3	Opens the list of previously selected devices.
	Buffers	F5	Opens the Buffers ^[61] dialog.
	Serialization, Checksum, Log file	F6	Opens the Serialization, Checksum, Log File ^[63] dialog
	Data caching, Standalone jobs...		
	IP address settings...		Opens the dialog for setting static IP addresses of device programmers
	Preferences	Ctrl+F6	Opens the Preferences ^[78] dialog.
	Simplified User Interface editor		Opens the Simplified User Interface ^[113] editor
	Environment		Opens the Environment dialog with tabs: the Fonts ^[80] tab ^[80] , the Colors ^[80] tab ^[80] , the Key Mappings ^[81] tab ^[81] , the Toolbar ^[82] tab ^[82] and the Misc ^[82] tab ^[82] .

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.

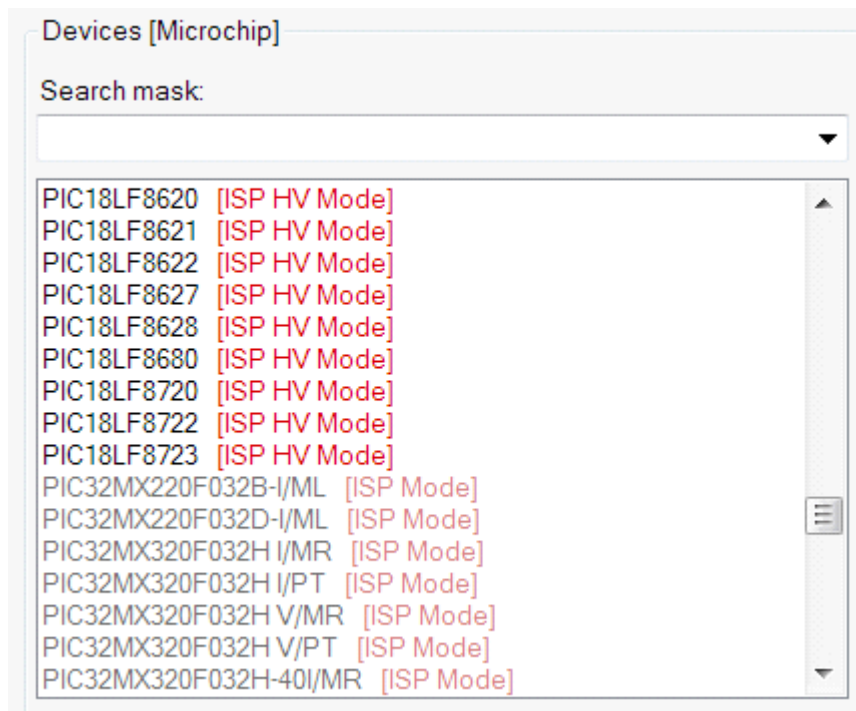


Control

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 requires having appropriate CPI2-D-xxxx device library licenses. After activation a certain CPI2-D-xxxx device library license all the part numbers of the devices covered by this license become visible and can be selected.

3.2.3.4.2 The Buffers Dialog

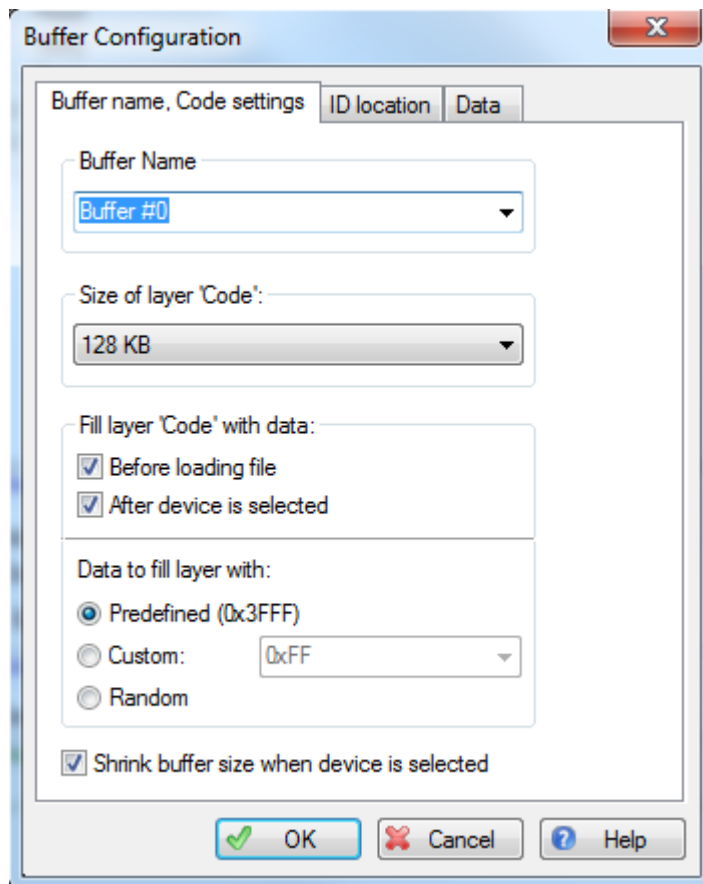
Control	Description
Buffer list:	Displays names, sizes and sub-layers of all open buffers ^[18]
Add...	Opens Buffer Configuration ^[61] dialog to create a new buffer
Delete	Deletes the buffer highlighted in the 'Buffer list' box.
Edit...	Opens Buffer Configuration ^[61] dialog for editing.
View	Switches 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](#) ^[95].

The dialog has one tab for each [sub-layer](#) ^[18] of a particular device. Every buffer has at least one main **Code 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-IPT 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:	<p>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.</p> <p>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</p>

	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

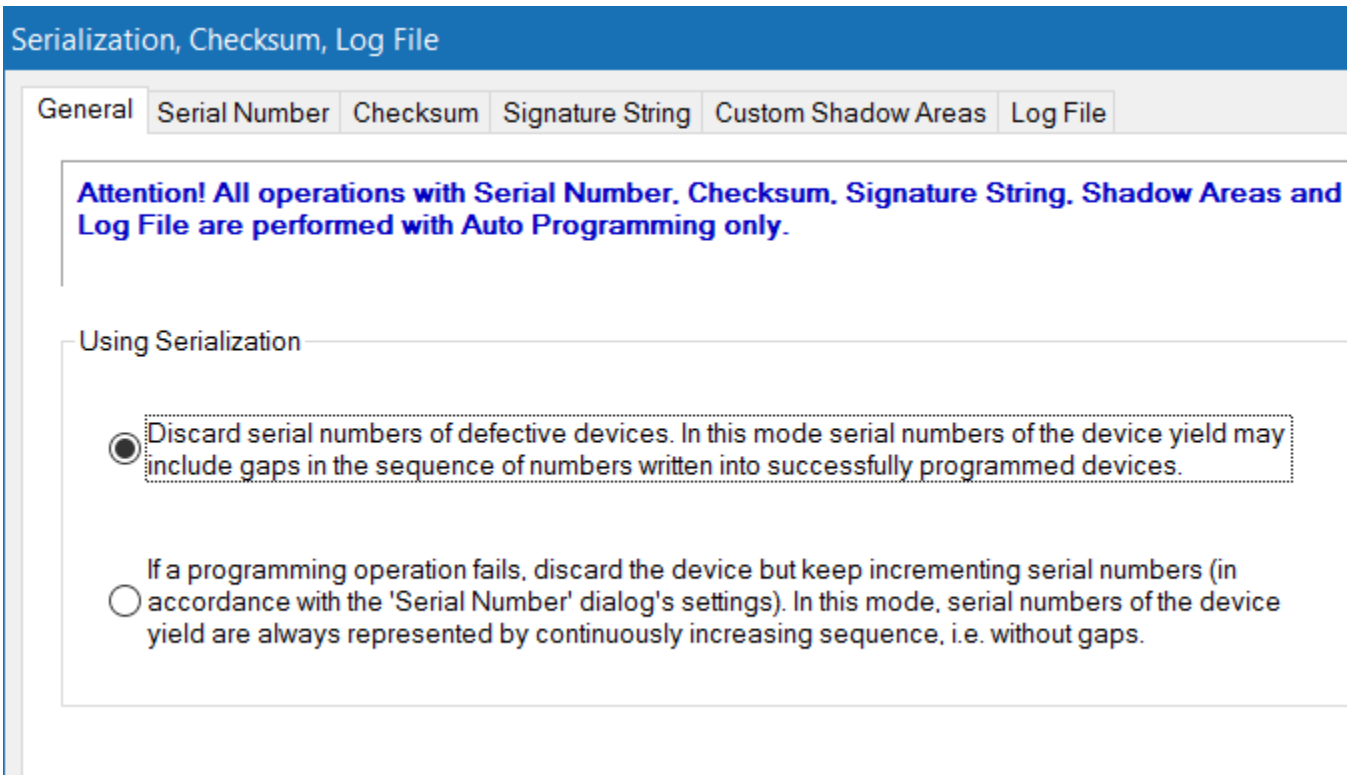
The dialog allows writing serial numbers, unique signatures, checksums and user-specified information into target device memory. It also allows to configure writing log of the process of mass production device programming.

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.

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



[General](#) ⁶⁸

[Serial Number](#) ⁶⁹

[Checksum](#) ⁶⁹

[Signature String](#) ⁷⁰

[Custom Shadow Areas](#) ⁷¹

[Log File](#) ⁷³

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](#) ⁶⁴ to the target device memory.

3.2.3.4.3.1 Shadow Areas

Concept of Shadow Memory Areas

Shadow memory areas are special parts of the computer RAM that the ChipProg-02 program handles in a special way allowing to create unique data images for each single device to be programmed. In most cases such a challenge is essential for Gang-Programming when multiple CPI2-B1 device programmers concurrently flashes identical devices on boards comprising a multi-PCB panel. Then very often, besides the same code, it is necessary to write into each device a unique, device-specific, information: such as a serial number, checksum, bar code scanned from the board, device MAC address, etc.. The ChipProg-02 software is featured with a mechanism allowing to create such unique, dynamically changing data and to merge these data with the code, writing these merged images into specified part of the device memories. The ChipProg **shadow memory** mechanism, implemented in the ChipProg-02 software and CPI2-B1 firmware, enable correct merging of the common data with dynamically changeable portions of data into one data image, unique for each target device. Shadow areas are special memory locations laying away from the [buffer](#)^[18], in the computer RAM. Hereafter in this chapter the "buffer" means a specified [layer](#)^[18] of the device memory (Code, ID parameters, Data, EEPROM, etc.) that contains a common part of data image to be written in the devices on boards.

CPI2-B1 operates with two types of shadow memory areas:

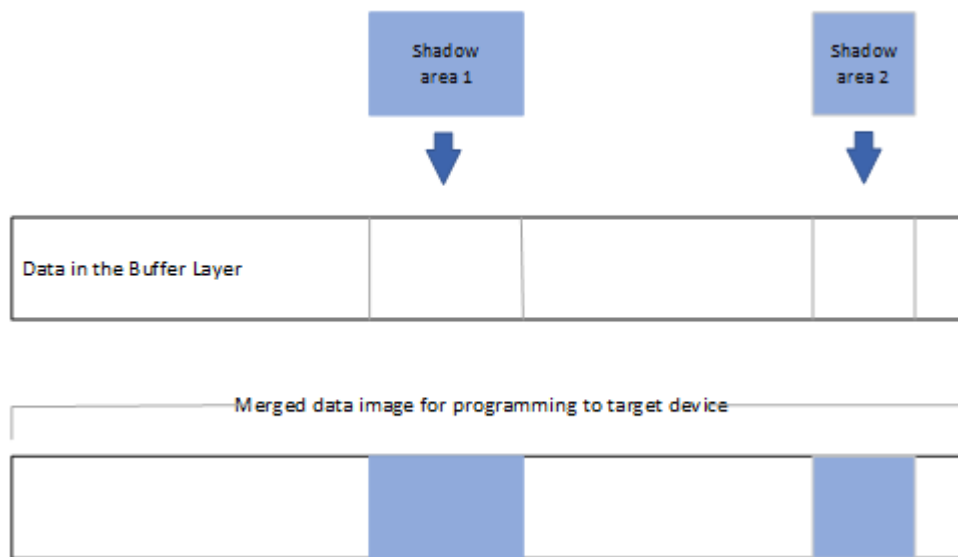
- a) **dedicated** to certain, frequently used parameters;
- b) **custom** shadow areas that can be used for programming custom parameters.

CPI2-B1 has three types of shadow memory areas **dedicated** to the parameters frequently programmed into devices along with the code: [Serial Number](#)^[69], [Checksum](#)^[69], and [Signature String](#)^[70]. The ChipProg-02 setting dialogs for each of these parameters are very specific and the mechanisms of blending these parameters located in dedicated shadow memory areas with the buffer content are built into the ChipProg-02 software and cannot be changed by the programmer user.

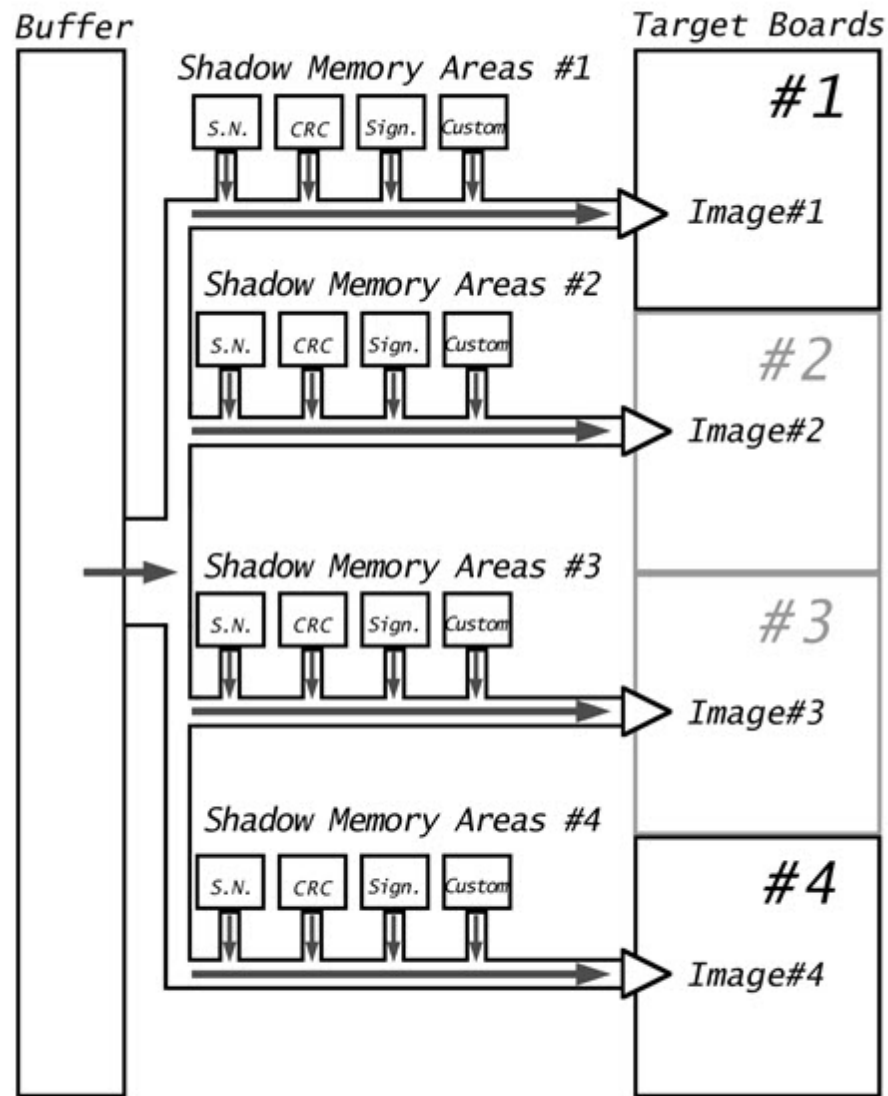
For specifying other parameters, such, for example, as bar codes scanned from target boards, device MAC addresses, parameters exceeding limitations of the dedicated shadow memory settings, etc., ChipProg-02 enables creation virtually unlimited number of [Custom Shadow Areas](#)^[71] and manipulation with them.

How does it work?

When a current programming site initiates a request for the device #N programming, the CPI2-B1 fetches data from the source buffer layer, browses shadow areas predefined for the site #N and replaces the layer data by contents of these area forming the merged data image to be written to the device #N and physically writes this merged image to the device. Then the programmer repeats the operations for the device #N+1 taking content of the shadow areas predefined for the device #N+1 and so on and so on. The addresses of each identical shadow memory areas and their sizes are the same for all devices but the contents vary. The picture below shows how the programmer prepares a data image to be written to a target device.



The diagram below displays shaping data images for four board programming. Each unique data image includes a common part fetched from the buffer [layer](#)¹⁸ merged with contents of three dedicated and one custom shadow areas.



Overlapping of Shadow Areas and Buffer Data

If any addresses in the merged data overlap, the data read from shadow areas overwrite the data read from memory buffer, in the order 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 ?

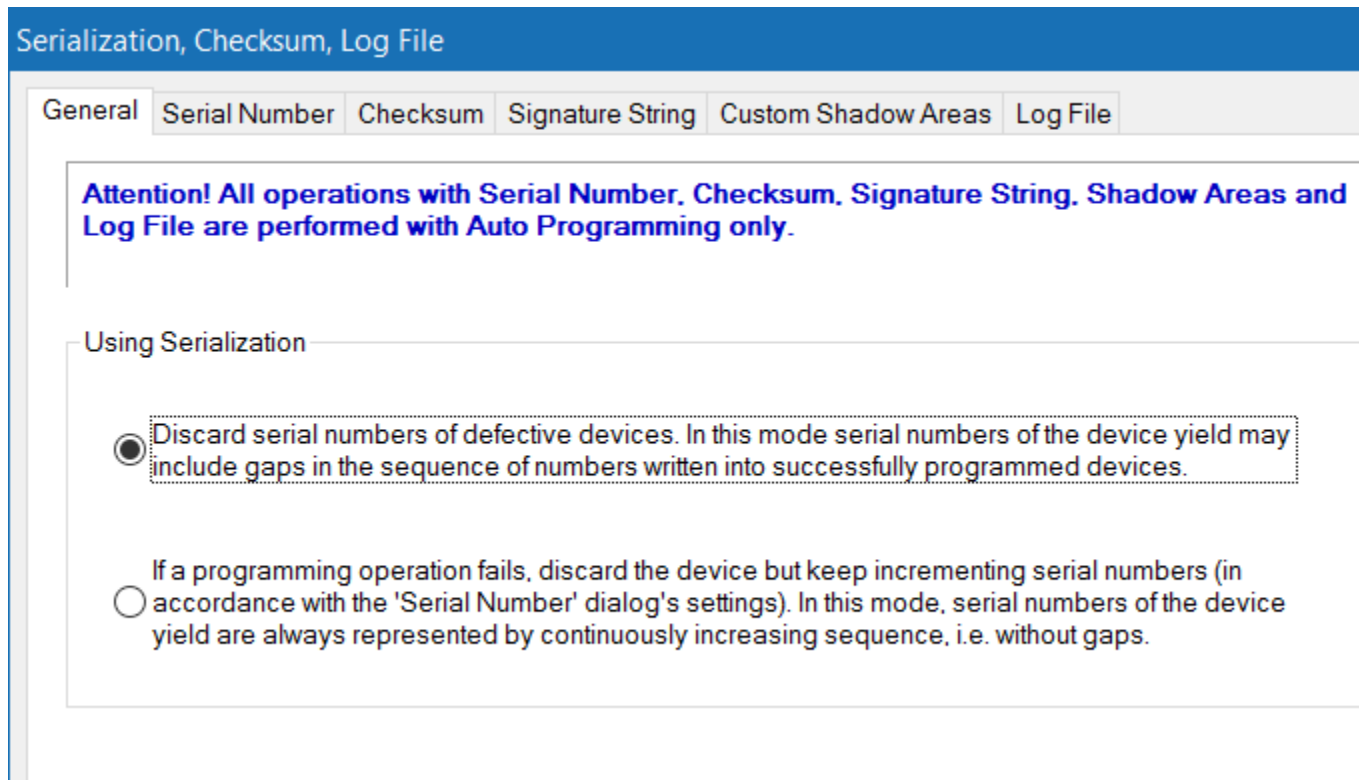
Serial Number ?

Data in memory buffer

The ChipProg-02 software itself does not prevent or warn about the shadow memory overlaps. The user should carefully check correctness of the addresses set in the [Serial Number](#)^[69], [Checksum](#)^[69], [Signature String](#)^[70] and [Custom Shadow Areas](#)^[71] setting dialogs to prevent data image corruption as a result of accidental shadow areas overlapping.

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 vice 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](#)¹⁷⁶.

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 or 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

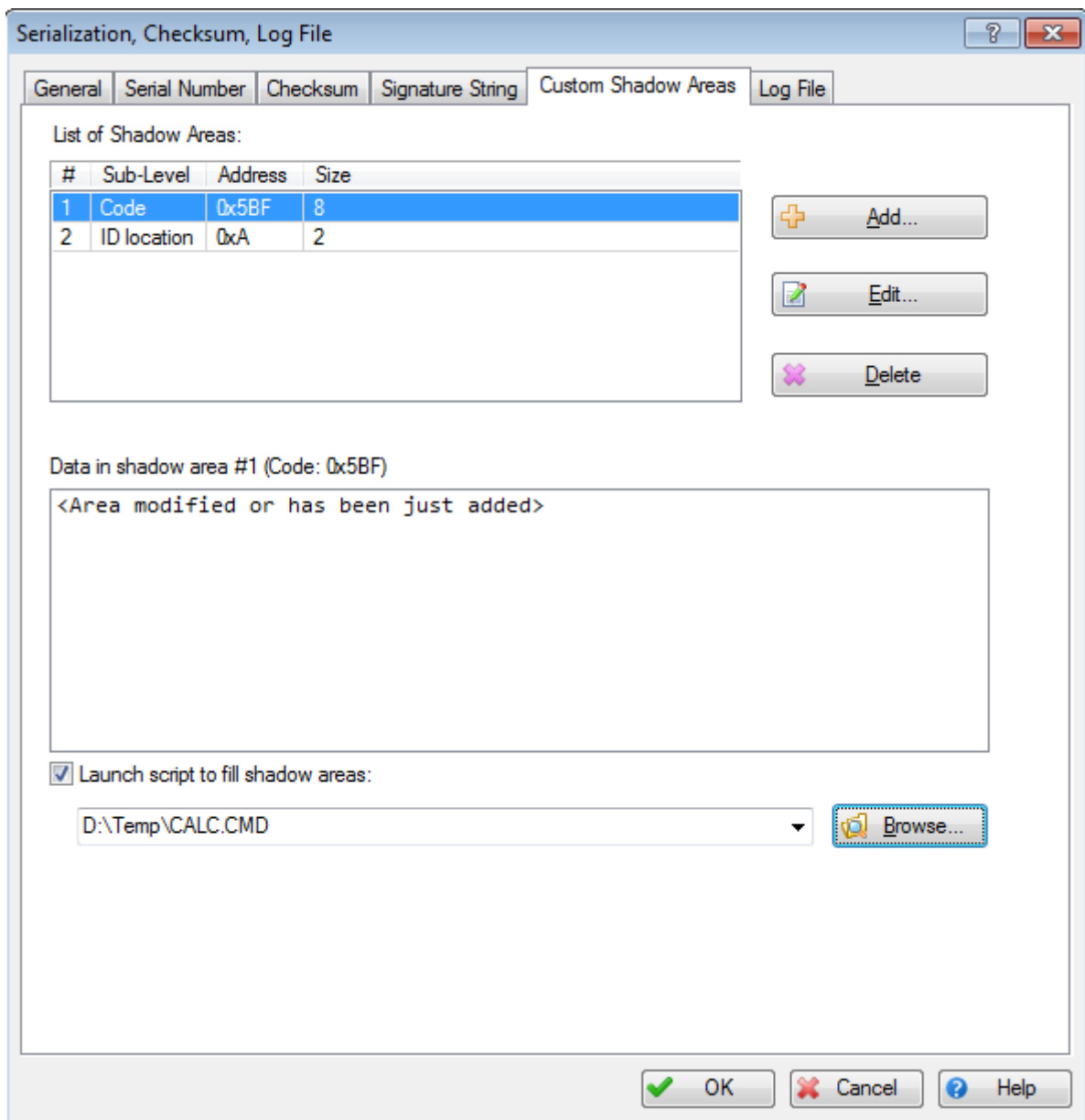
The tab contains settings for writing user-defined signature string into the target device. The signature may include generic data (such as the date when the device was programmed) and unique data (such as project name, operator name, etc.).

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 opens the dialog allowing to set custom shadow areas and to watch content of these areas for debugging of automated device programming.



Click the **+Add** button opens a sub dialog prompting to specify the [buffer layer](#)^[18], content of which will be merged with the custom shadow memory area, the area's address and size. A user may create as many custom shadow areas as needed to be blended to same or different buffer layers. The picture above displays two custom shadow areas reserved for two buffer layers: Code and ID parameters.

The pane **Data** displays current content of the highlighted shadow area. Right after creation it is blank. Then the area can be filled by executing of an ACI function or by a script. To use a script check the box below the Data pane and specify the script name and location. In the example above the area #1 is going to be filled by the script CALC.CMD.

Though, the shadow memory areas are mostly used for mass production in the gang programming mode, sometimes this feature can be used for operation with a single CPI2-B1 device programmer. For example, the [Checksum](#)^[69] setting dialog allows to calculate a single 8-, 16- or 32-bit checksum. Use of the custom shadow areas enables to multiple checksums with the size exceeding 32 bits.

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.

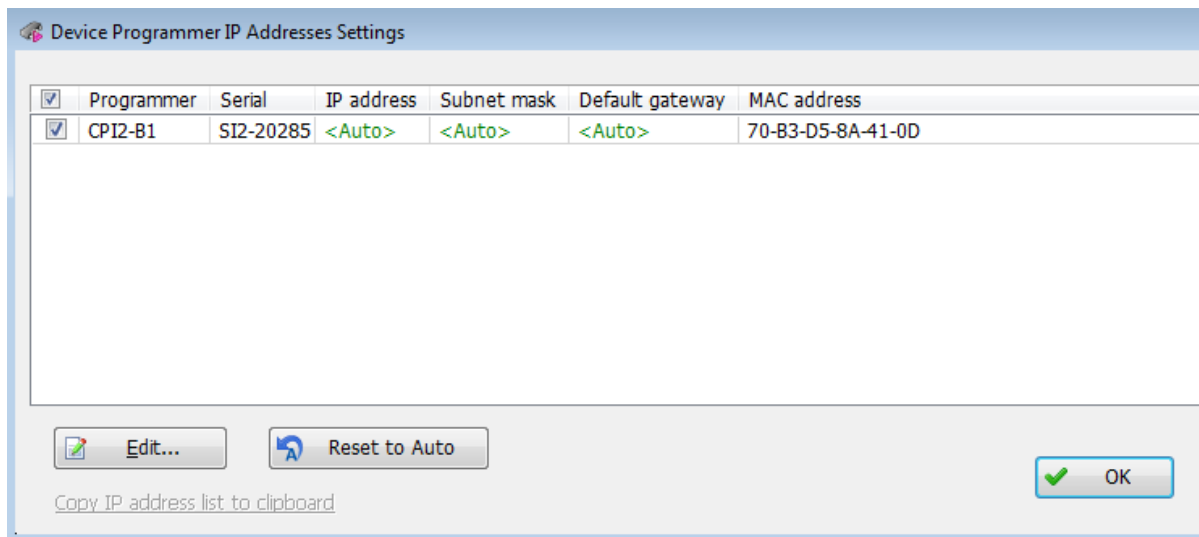
Signature string	Check this box to log signature string read from the device.
Checksum	Check this box to log checksum value read from the device.
Buffer name	Check this box to log buffer name.
Programming address	Check this box to log ranges of device locations that have been programmed.
Programming options	Check this box to log all programming options.
Log File Format	A Pair of radio 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 Sata Caching, Standalone... Dialog

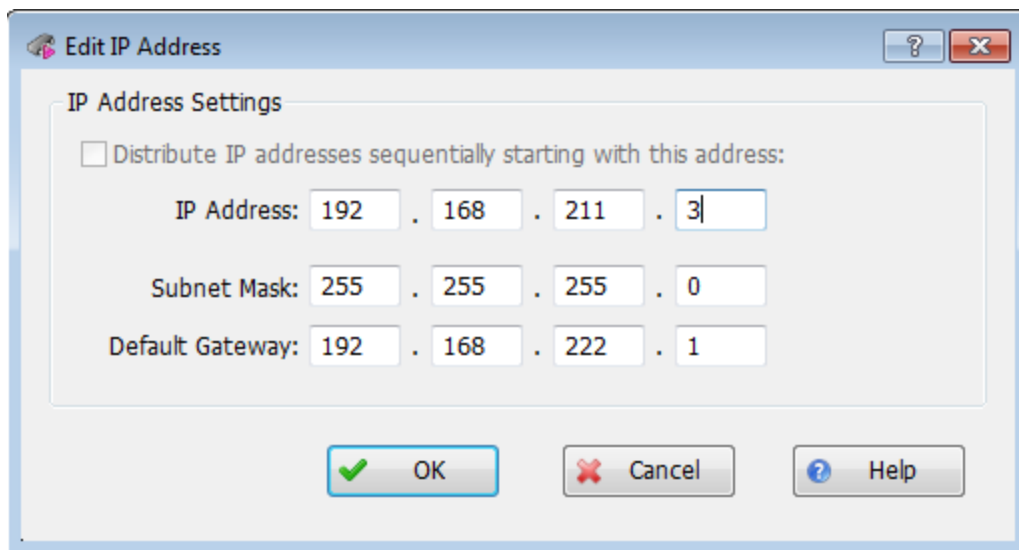
This topic refers to the settings of **Standalone Operation Mode**. Read the entire [chapter](#)¹³².

3.2.3.4.5 IP Address Setting Dialog

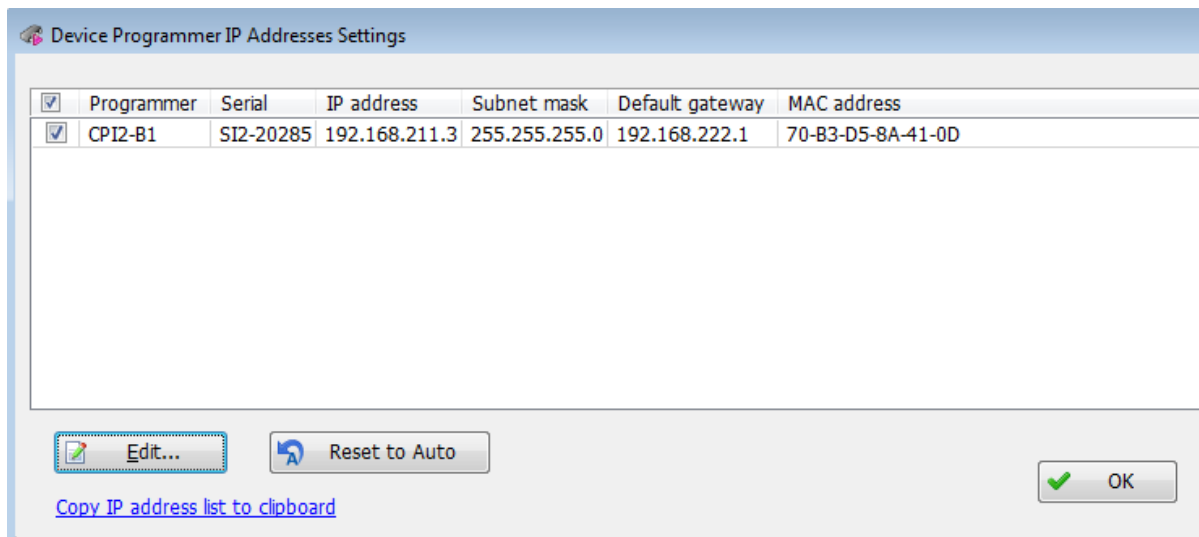
If a single CPI2-B1 programmer is controlled via Ethernet, a DHCP server assigns local IP address to this programmer. To set a static IP address a CPI2-B1 programmer should be **connected to a PC via USB**. To set the IP addresses invoke the **Configure > IP address settings...** menu. This will open the dialog below:



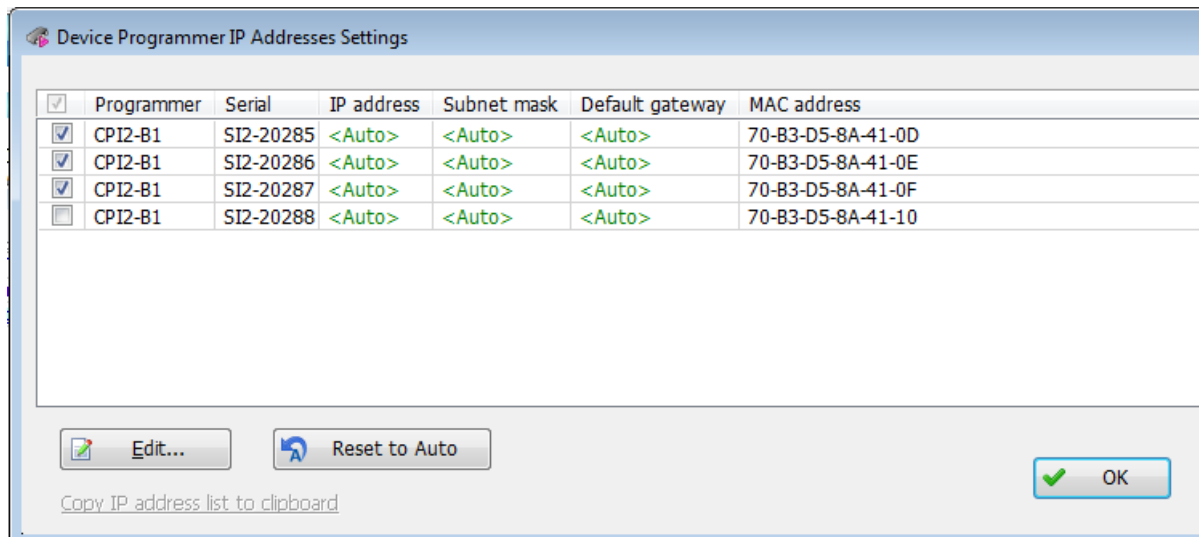
It displays a string with the CPI2-B1 with its serial number and MAC address. To assign static IP addresses click the **Edit** button. In the popped up **Edit IP Address** dialog type in the IP address value, subnet mask, default gateway and click **OK**.



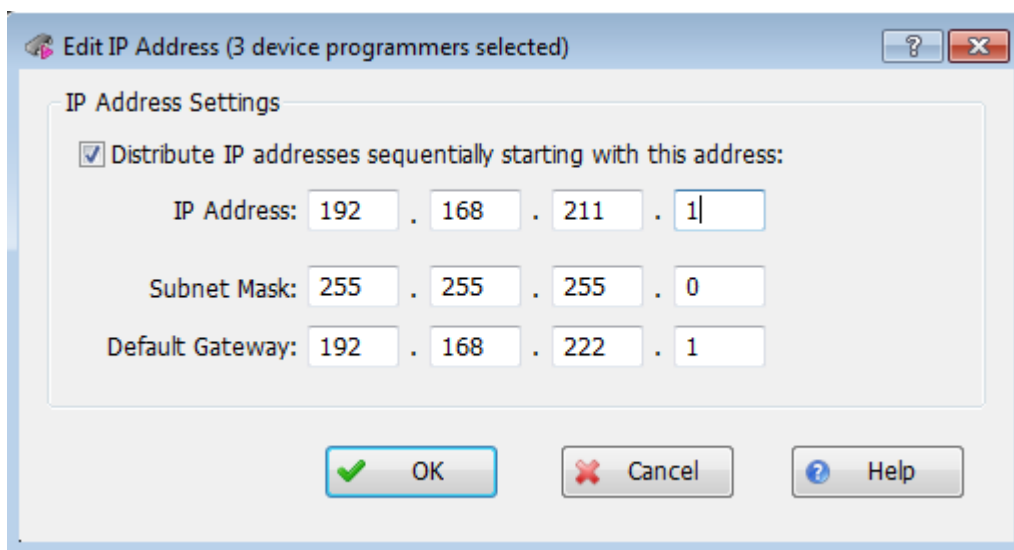
You will see a new IP address set:



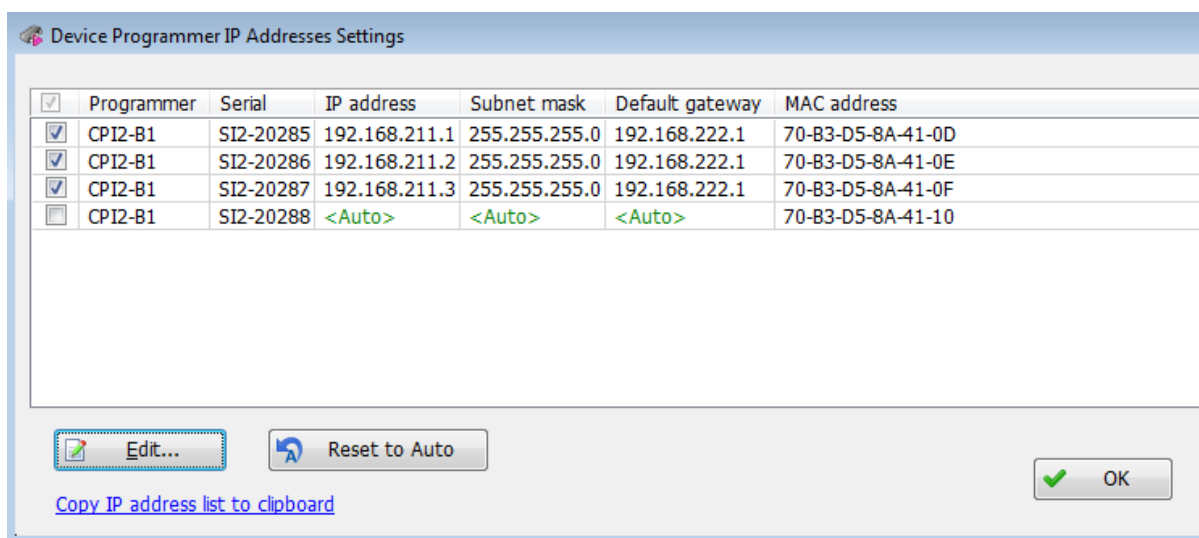
If you control multiple CPI2-B1 device programmers from one PC, you may want to set static IP addresses for all (or to a few) of them at once. Invoke the **Configure > IP address settings...** menu to open the dialog below. In the example below it lists four CPI2-B1 programmers with their serial numbers. Select those you would like to set IP addresses for (in the case below with serial numbers SI2-20285, SI2-20286 and SI2-20287).



Then click the **Edit** button to set the addresses for three selected programmers:



In the dialog check the **Distribute IP addresses...** box and enter the first IP address. Then click **OK** to complete settings. This will assign the specified address to the most top device programmer in the list; other selected programmers will be automatically assigned with IP addresses incremented by 1. See the result below illustrated setting static IP addresses for 3 of 4 CPI2-B1 device programmers:



ChipProg-02 set identical Subnet mask and Default gateway for all the programmer sites. After setting static IP addresses you can copy these settings to the clipboard and then to a file.

Important Note.

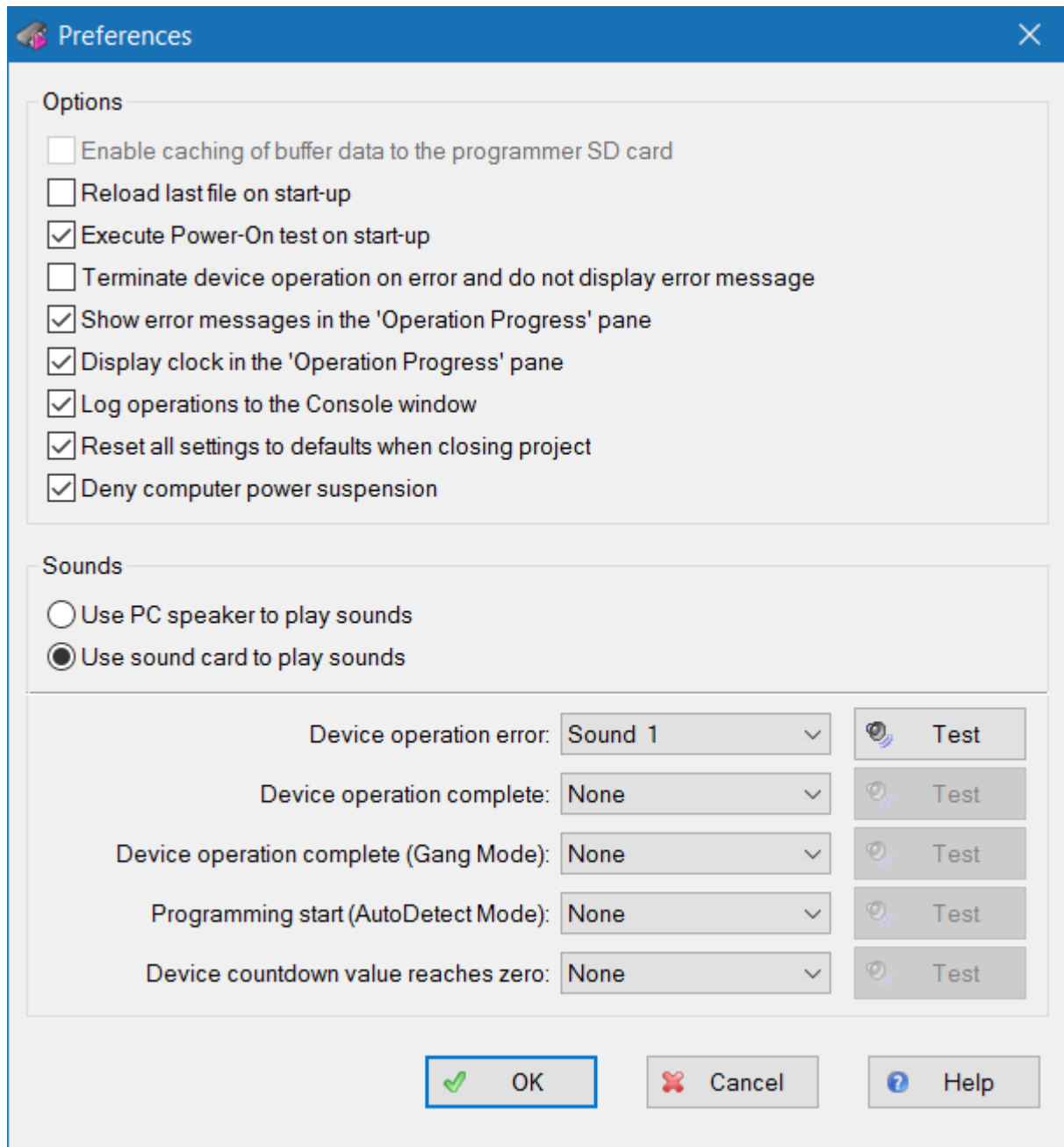
to complete setting static IP addresses before restarting the programmer with LAN control you must cycle the programmer power.

3.2.3.4.6 Simplified User Interface Editor

This topic refers to the settings of the **Simplified User Interface** (SUI). Read the entire [chapter](#) ¹¹³.

3.2.3.4.7 The Preferences Dialog

This dialog contains settings for miscellaneous options.



Preferences

Options

- ☐ Enable caching of buffer data to the programmer SD card
- ☐ Reload last file on start-up
- ☒ Execute Power-On test on start-up
- ☐ Terminate device operation on error and do not display error message
- ☒ Show error messages in the 'Operation Progress' pane
- ☒ Display clock in the 'Operation Progress' pane
- ☒ Log operations to the Console window
- ☒ Reset all settings to defaults when closing project
- ☒ Deny computer power suspension

Sounds

- ☐ Use PC speaker to play sounds
- ☒ Use sound card to play sounds

Device operation error:	Sound 1	Test
Device operation complete:	None	Test
Device operation complete (Gang Mode):	None	Test
Programming start (AutoDetect Mode):	None	Test
Device countdown value reaches zero:	None	Test


OK Cancel Help


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 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	<p>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.</p> <p>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.</p> <p>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.</p>
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.

3.2.3.4.8 The Environment Dialog

The Environment dialog includes the following tabs:

[Fonts](#)  tab,

[Colors](#)  tab,

[Mapping Hot Keys](#) ⁸¹ tab,

[Toolbar](#) ⁸² tab,

[Miscellaneous Settings](#) ⁸² tab.

3.2.3.4.8.1 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.

3.2.3.4.8.2 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	<p>Name of the color scheme. You can type a name or choose a recently used one from the list.</p> <p>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.</p>
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	<p>Opens the Color dialog if the Inherit Windows Color and Use Inverted Text/Background Color boxes are unchecked for this type of window.</p> <p>The Color dialog also opens with a double-click on a color in the Colors list.</p>
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	<p>To highlight syntax in the Source window you can specify additional font attributes - Bold and Italic.</p> <p>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.</p> <p>Sometimes this effect takes place with Fixedsys font. If you need to use Bold fonts, choose the Courier New font.</p>

3.2.3.4.8.3 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.

3.2.3.4.8.4 Toolbar

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


<u>Control</u>	<u>Description</u>
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 specified windows.
Toolbar Settings are the Same for Each Project/Desktop File	Applies current settings of this dialog to other projects or future opened files.

3.2.3.4.8.5 Messages

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

3.2.3.4.8.6 Miscellaneous Settings

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

<u>Control</u>	<u>Description</u>
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	Makes mouse double click equivalent to single click plus pressing OK button in dialogs.

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 ^[104] 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.

3.2.3.4.9 The Editor Options Dialog

The ChipProg-02 software includes a **built-in [Scripts Files](#)**^[176] editor. The **Editor Options dialog** provides access to editor settings and includes the following tabs.

[General Editor Settings](#)^[83] tab,

[Key Mapping](#)^[85] tab.

3.2.3.4.9.1 The General Tab

The **General** tab of the **Editor Options** dialog has settings for common options that apply to every [Source](#)^[186] 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 ^[187] .
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 ^[188] 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 ^[188] windows will have Auto Word/AutoWatch pane at their right, and <i>the automatic word completion function</i> 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 ^[189] 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](#)^[80] 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.

3.2.3.4.9.2 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 ^[86] 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 ^[184] 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](#)^[250] 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 KEY CMD. For more information about developing scripts, see [Script Files](#)¹⁷⁶.

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; others 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 commands of this menu can be launched by hot keys [F7] ...[F12].

Commands	Scripts	Window	Help
Blank Check			F8
Program			F9
Verify			F10
Read			F11
Erase			F7
Auto programming			F12
Self-Tests...			
Switch to Stand-Alone mode...			
Switch to Simplified User Interface...			
Local menu			Ctrl+F10, Ctrl+Enter
Calculator...			Shift+F4

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 ^[108] .
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 ^[132] .
Switch to Simplified User Interface		Hides a standard GUI and replaces it with a preset Simplified User Interface ^[113] .
Local menu		Opens local menu of the active window.
Calculator		Opens Calculator ^[88] dialog which performs calculator functions.

3.2.3.5.1 Calculator

The primary purpose of the embedded calculator is to evaluate [expressions](#)^[202] 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](#)^[176] 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 ^[178] dialog from which you can
	New Script Source	Create a new Script File text.
	Open Watches window	Opens the Watches ^[193] window.
	Add watch...	Adds watch to the Watches window .
	Editor window ^[186]	Opens a list of the commands to Compose a new, Open, Save, Save as, Print a script file. of the Editor ^[186] window.
	Text Edit ^[186]	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](#)^[178] 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](#) .


<u>Command</u>	<u>Description</u>
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 crashes 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 

Clicking 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 go beyond the set of CPI2-B1 default features. For example, the **CPI2-ACI** license enables use of the Phyton ChipProg-02 [Software Development Kit \(SDK\)](#)^[158], [On-the-Fly Control](#)^[126] utility and integration with NI [LabVIEW](#)^[172] software in addition to the default capabilities.

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 tied to the CPI2-B1 with a certain serial number as "Enabled" in green color. The licenses which optionally may be added at a later time are marked as "Not licensed" in grey color.

If you have purchased a new license or licenses Phyton sends you a binary file that enables specific capabilities. 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, select 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](#)^[52]:

- [Program manager](#)^[106]
- [Device and Algorithm Parameters' Editor](#)^[93]
- [Buffer](#)^[95]
- [Device Information](#)^[92]
- [Console](#)^[104]

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

- Editor
- Watches

3.2.4.1 The Device Information Window

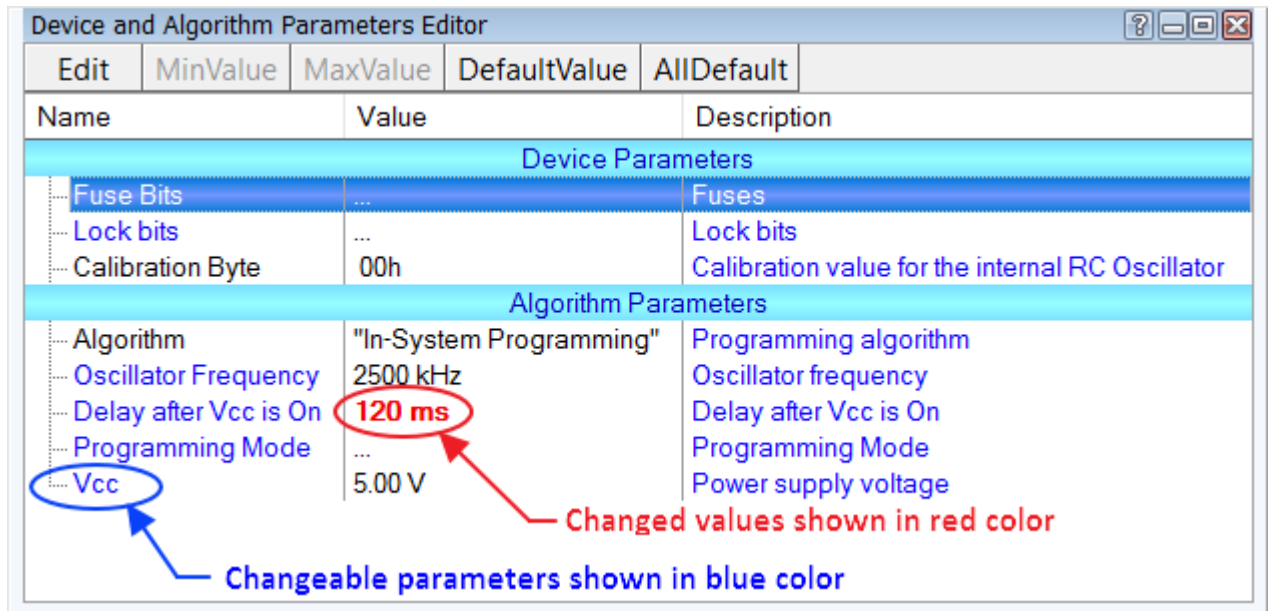
This window displays the type of selected target device and a link opening a connection diagram between the [TARGET](#)^[22] connector of CPI2-B1 and a selected target device (DUT).



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](#)^[86] command in the [Program Manager](#)^[106] window.



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

Device Parameters	<p>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.</p>
<p>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</p>	

Manager window as shown in the illustration.

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.

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.

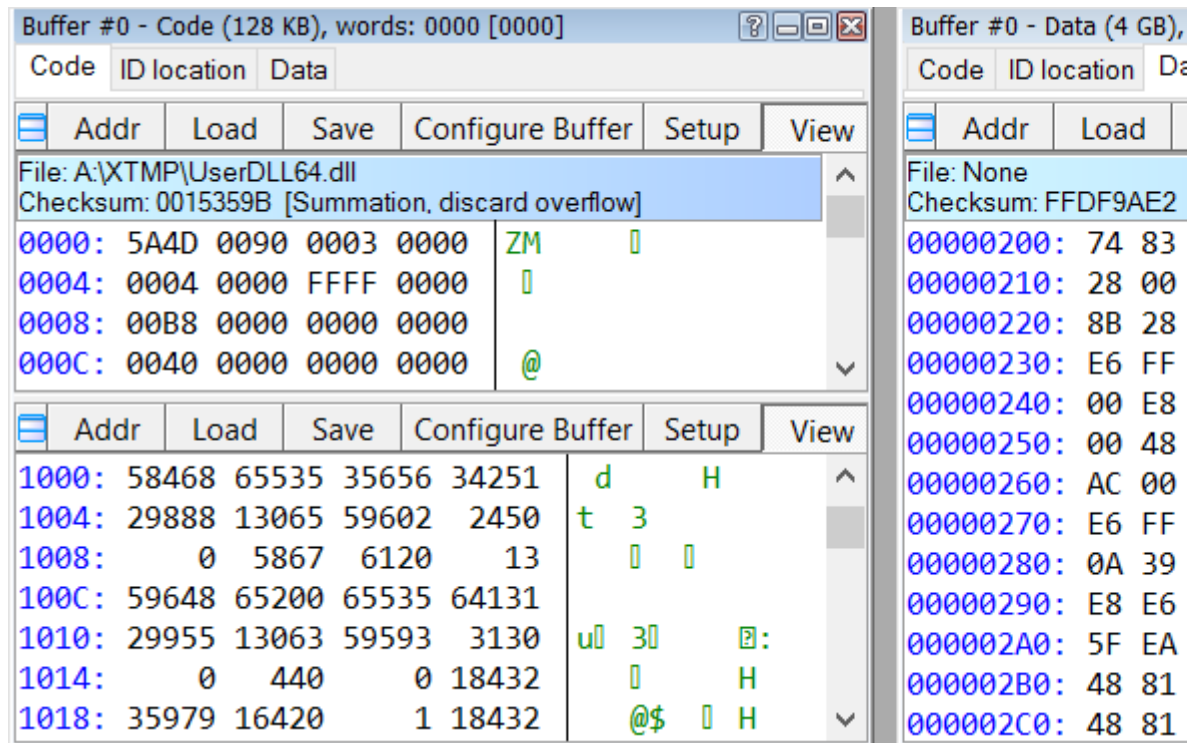
3.2.4.3 The Buffer Dump Window

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

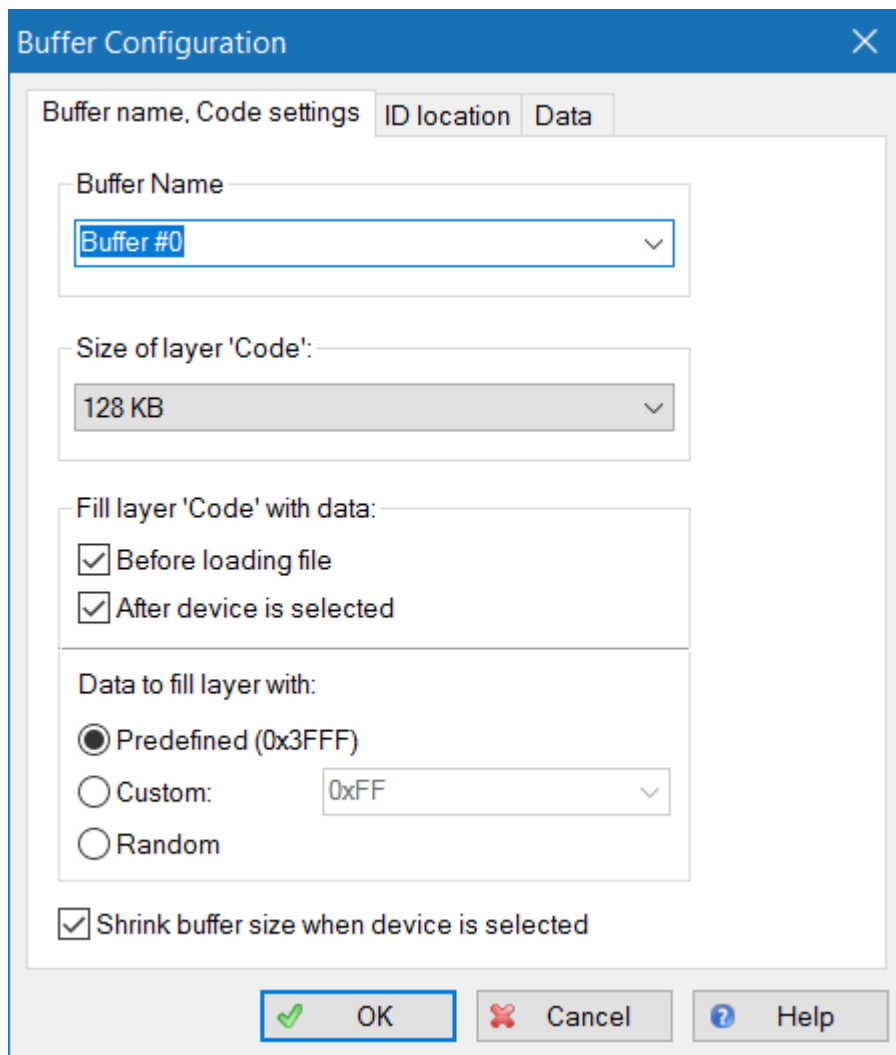
The context-sensitive menu brought up by a right mouse click is used to invoke context commands and dialogs of the **Buffer Dump** window. Most, but not all, local menu entries are duplicated by local toolbar buttons at the top of the window. Following are local menu and toolbar items:

Menu Item	Toolbar button	Description

New address...	Addr	Opens the Display from Address ^[100] dialog.
Load file to buffer...	Load	Opens the Load Window Dump ^[102] dialog.
Save data to file...	Save	Opens the Save Window Dump ^[104] dialog.
Configure buffer...	Configure buffer	Opens the Configuration Window Dump ^[97] dialog.
Window setup...	Setup	Opens the Window Dump Setup ^[98] 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 ^[100] dialog. This is only enabled when the View only, edit disabled is unchecked.
Operations with memory blocks	Block	Opens the Operations with Memory Blocks ^[100] 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.



The image shows a 'Buffer Configuration' dialog box with a blue title bar and a close button (X) in the top right corner. The dialog has three tabs: 'Buffer name, Code settings' (selected), 'ID location', and 'Data'. Under the 'Buffer name, Code settings' tab, there are several settings:

- Buffer Name:** A dropdown menu showing 'Buffer #0'.
- Size of layer 'Code':** A dropdown menu showing '128 KB'.
- Fill layer 'Code' with data:** Two checked checkboxes: 'Before loading file' and 'After device is selected'.
- Data to fill layer with:** Three radio buttons: 'Predefined (0x3FFF)' (selected), 'Custom: 0xFF' (with a dropdown arrow), and 'Random'.
- Shrink buffer size when device is selected:** A checked checkbox.

At the bottom of the dialog are three buttons: 'OK' (with a green checkmark icon), 'Cancel' (with a red X icon), and 'Help' (with a question mark icon).

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](#)⁹⁵ 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 display: 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](#) ⁹⁵ window.

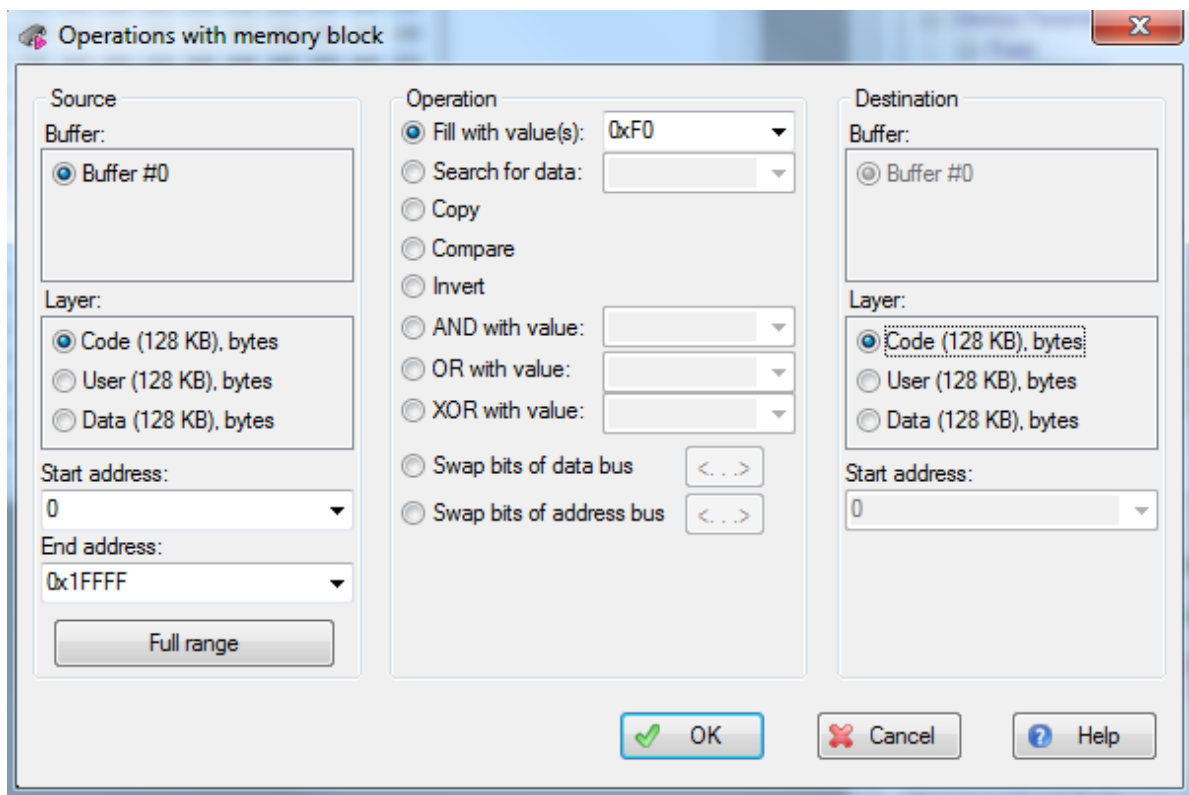
Element of dialog	Description
Type new address to display from:	Here you may enter any address within valid range.
History	Displays a list of previously entered addresses. You can pick one to set as starting address for the buffer dump.

3.2.4.3.4 The 'Modify Data' dialog

The dialog allows to edit data in the [Buffer Dump](#) ⁹⁵ window. The dialog can be invoked only when the **View** toolbar button is 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 will only 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.

<u>File Format</u>	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.
Angstrom SAV (*.sav)	This format is presented by Angstrom 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 ¹⁰³ 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**^[82] 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

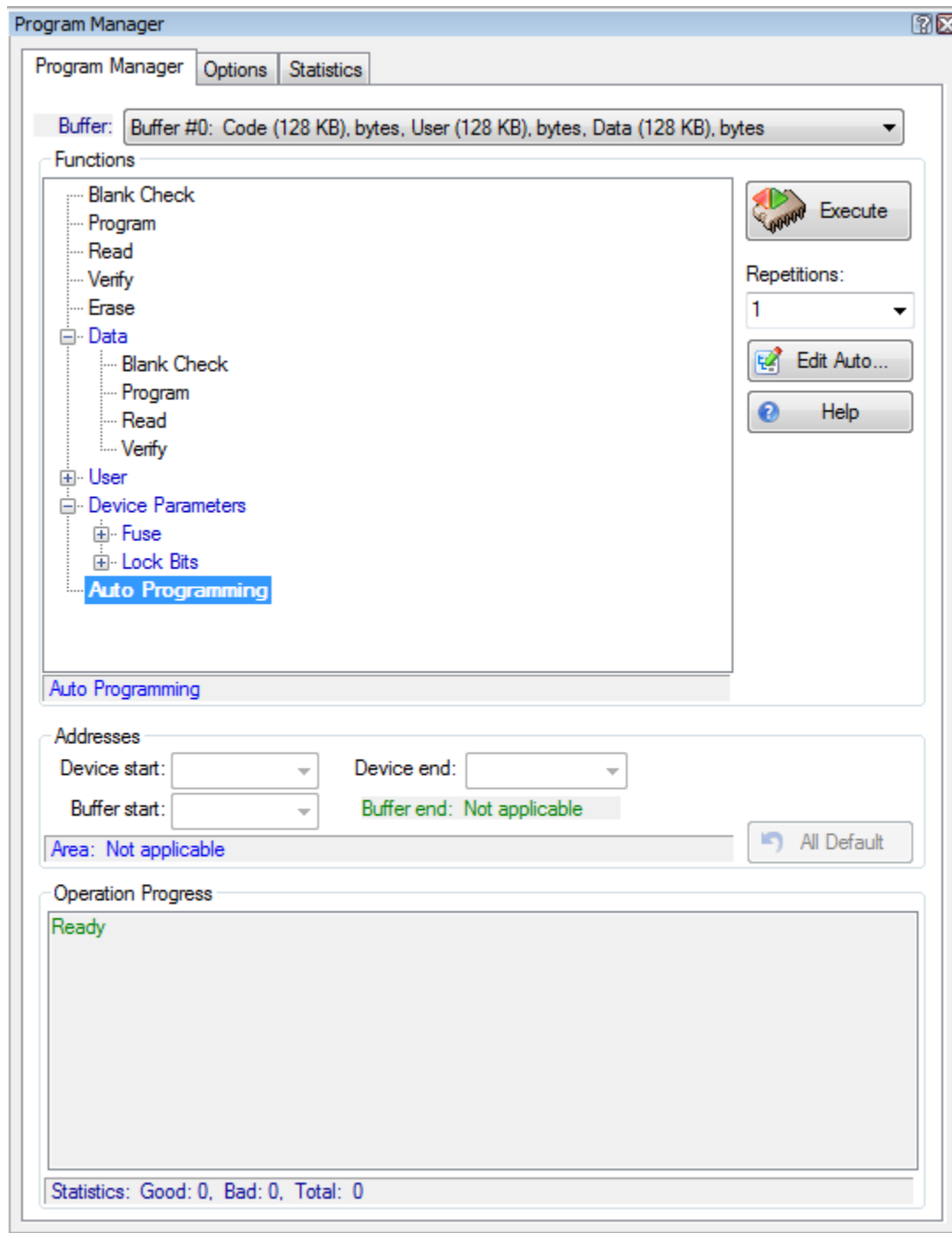
The **Program Manager window** is the primary screen object used by an operator to control the CPI2-B1 in the GUI mode. While some windows can be closed during programming operation, the Program Manager is supposed to be always open and visible. The window includes three tabs:

[The Program Manager tab](#)^[107] - by default this tab is open (see below)

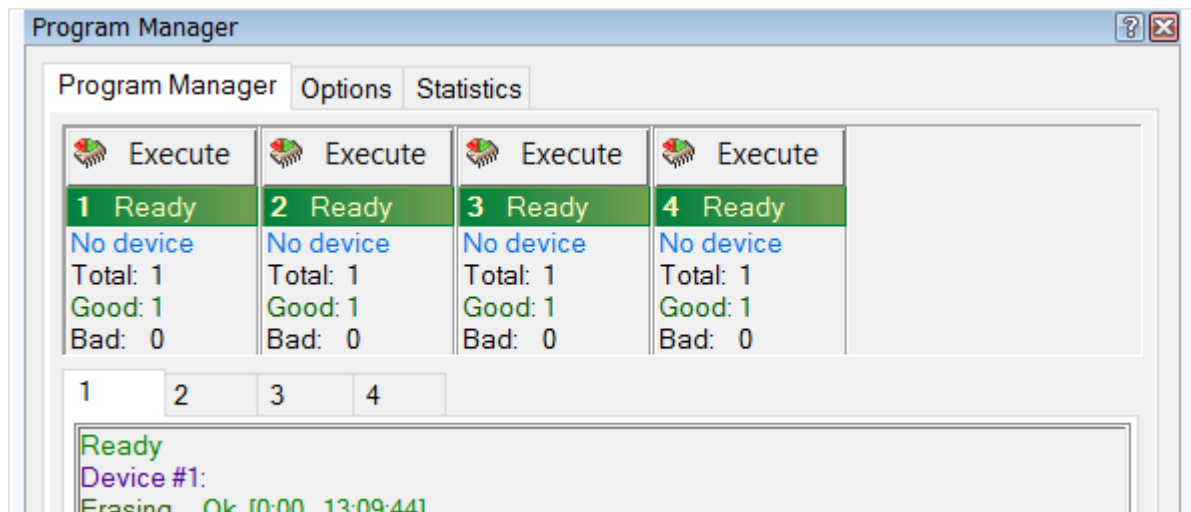
[The Options tab](#)^[109]

[The Statistics tab](#)^[111]

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](#)^[26] mode.



In the [Gang Programming](#)^[26] control mode the window's appearance is different. It displays as many tabs as many sites are united into the programming cluster of multiple CPI2-B1 programmers. Each tab has one **Execute** button, click on which initiates the [Auto Programming](#)^[108] command for a chosen site (device programmer). See below an example of the window for a cluster of four CPI2-B1 programmers driven in the [Gang Programming](#)^[26] 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 ^[108] 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 ¹⁰⁸ dialog.
Operation Progress	Displays progress bar and the status (OK, failed, etc.) of current operation.

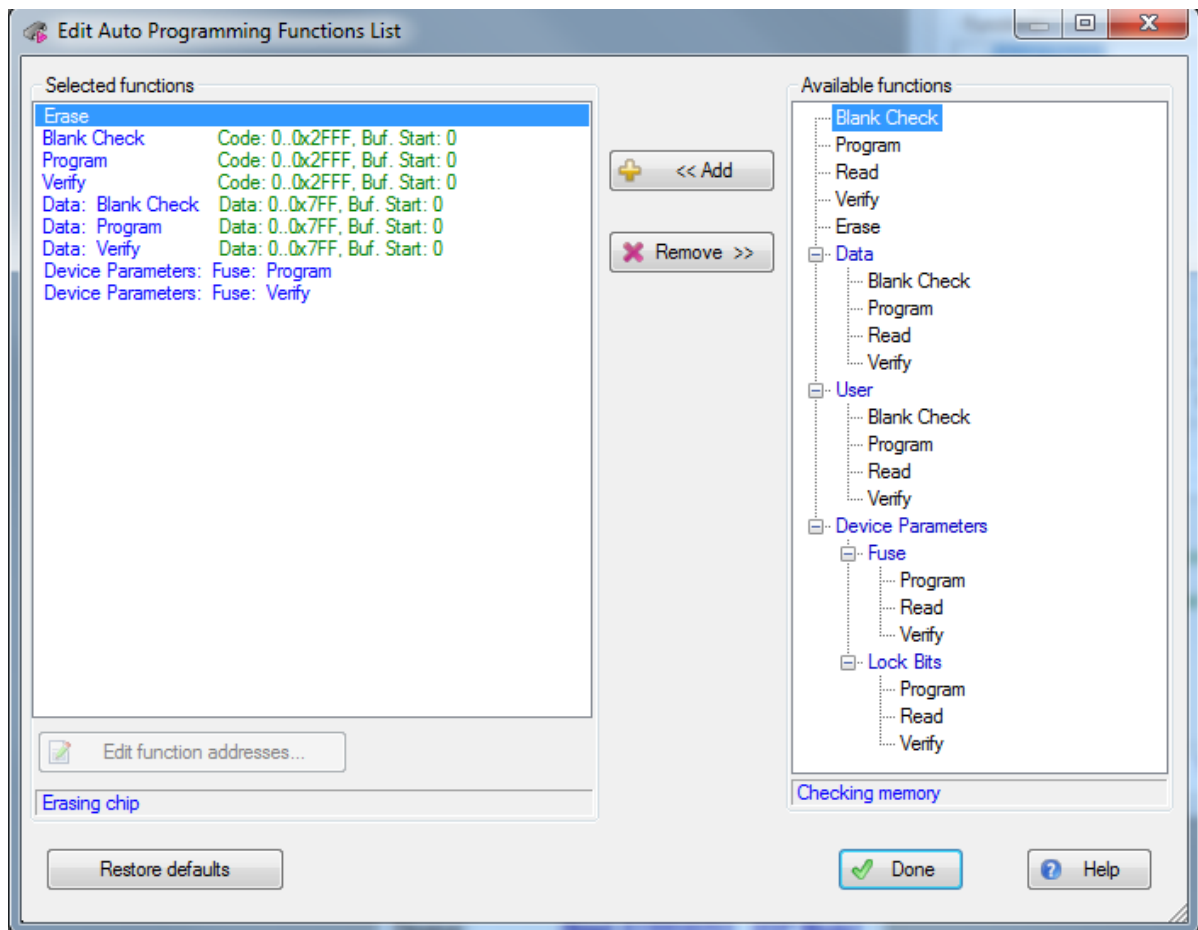
Besides generic functions such as **Blank Check, Read, Verify, Program, Auto Programming**, the **Functions** window often includes collapsed submenu of functions specific to the selected target device. When expanded it shows a list of commands for the parameters that can be set in the [Device and Algorithm Parameters](#)⁹³ 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^[24] connector.

3.2.4.5.2.1 Split data

Radio buttons in the **Split data** group of the [Option^{\[109\]}](#) 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.

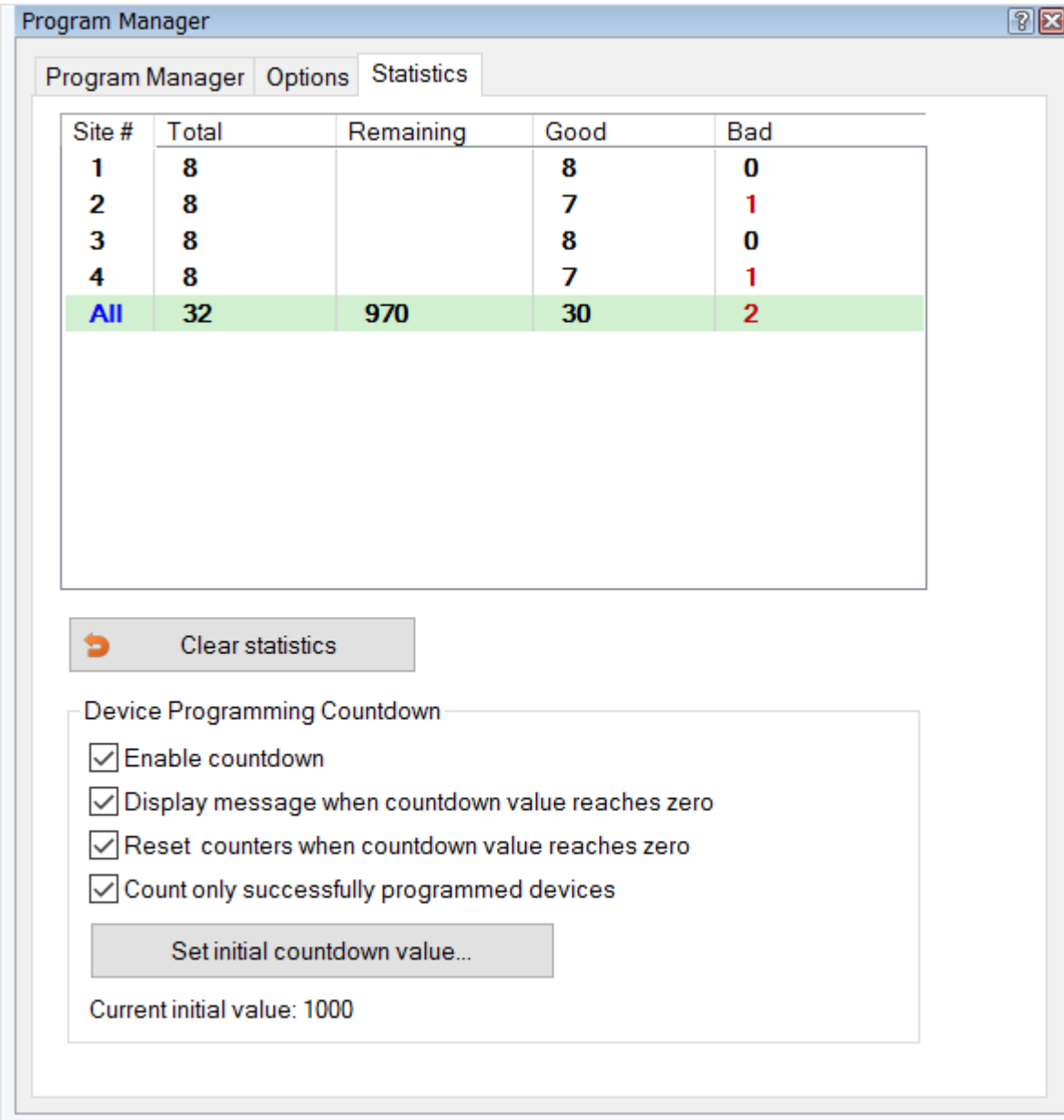
3.2.4.5.3 The Statistics tab

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](#)^[108] 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 ^[108] ; 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 ¹⁰⁸ .

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**).



The screenshot shows the 'Program Manager' window with the 'Statistics' tab selected. It displays a table with programming session statistics for four sites and a summary row. Below the table is a 'Clear statistics' button and a 'Device Programming Countdown' section with four checked options and a 'Set initial countdown value...' button. The current initial value is 1000.

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

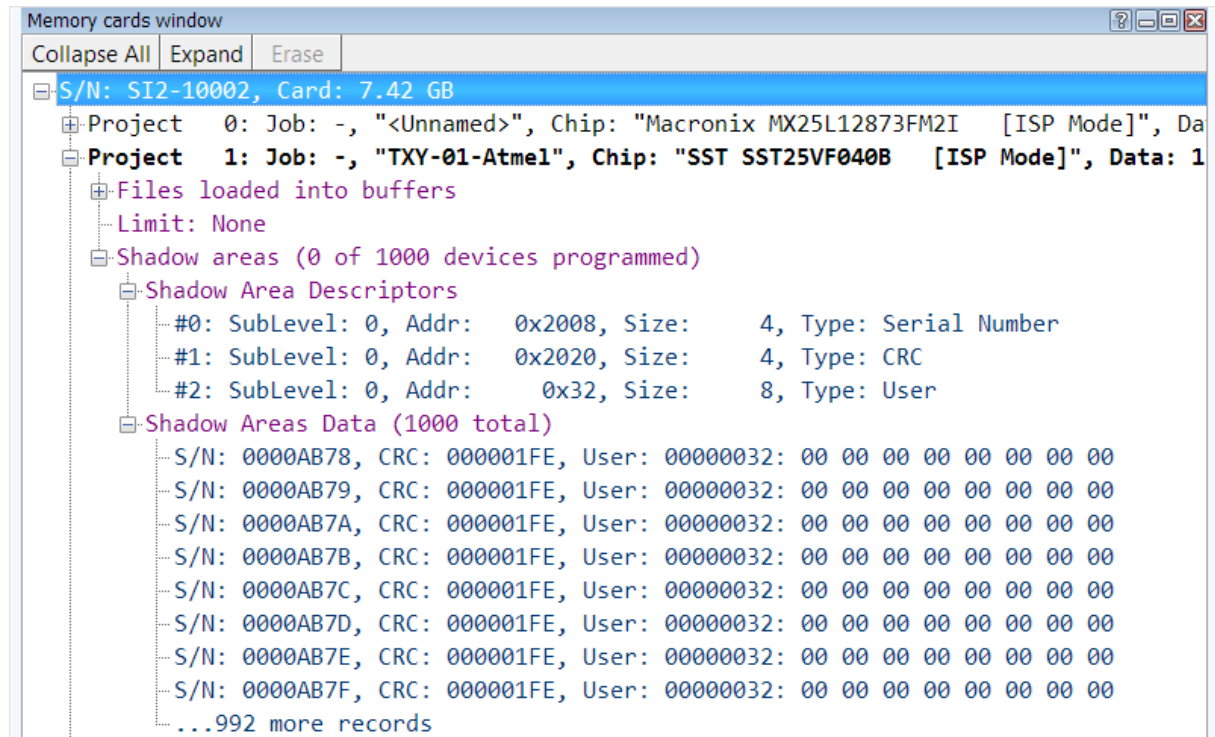
- ☒ Enable countdown
- ☒ Display message when countdown value reaches zero
- ☒ Reset counters when countdown value reaches zero
- ☒ Count only successfully programmed devices

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"](#)^[52] -> **"Memory Card Window."**



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.

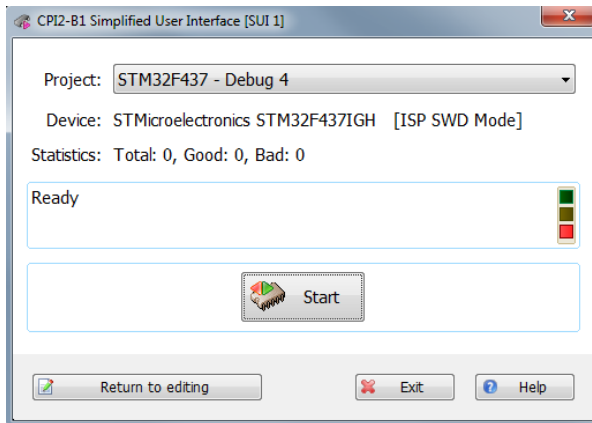
- [\(Script\) Editor](#)^[186] window
- [Watches](#)^[182] window
- [User](#)^[180] window
- [I/O Stream](#)^[180] window

These windows cannot be opened from the [View menu](#)^[52]. They may only be opened when you work with scripts. Operations with these windows are described in the [Scripts Files](#)^[176] chapter.

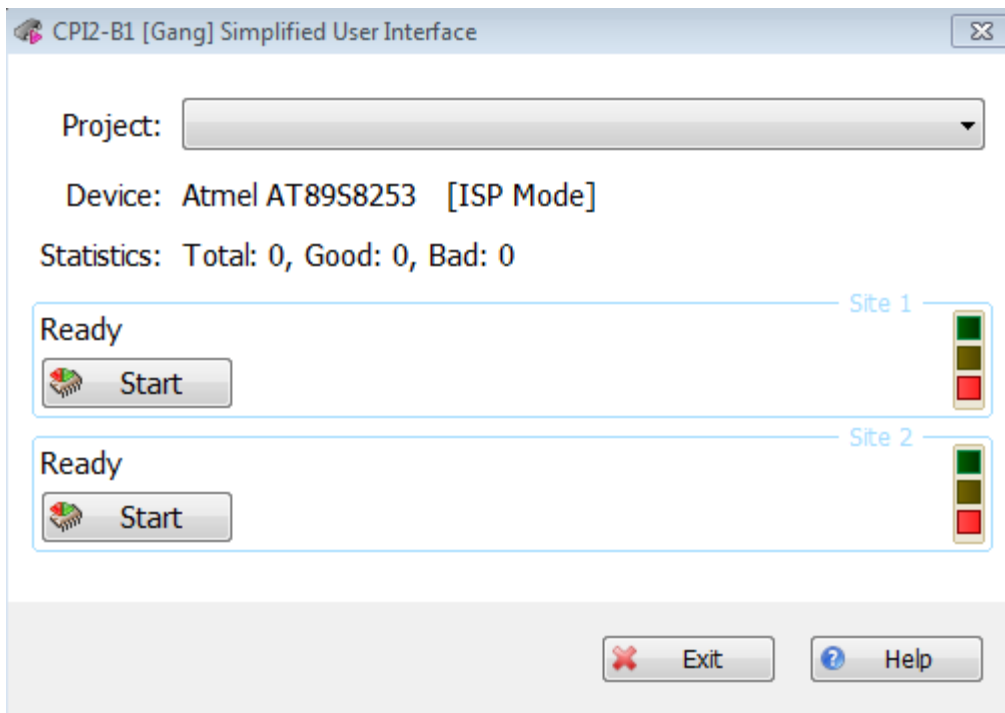
3.3 Simplified User Interface

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](#)^[106] command). To prevent casual CPI2-B1 mismanagement and to simplify routine operations, the ChipProg-02 enables switching the CPI2-B1

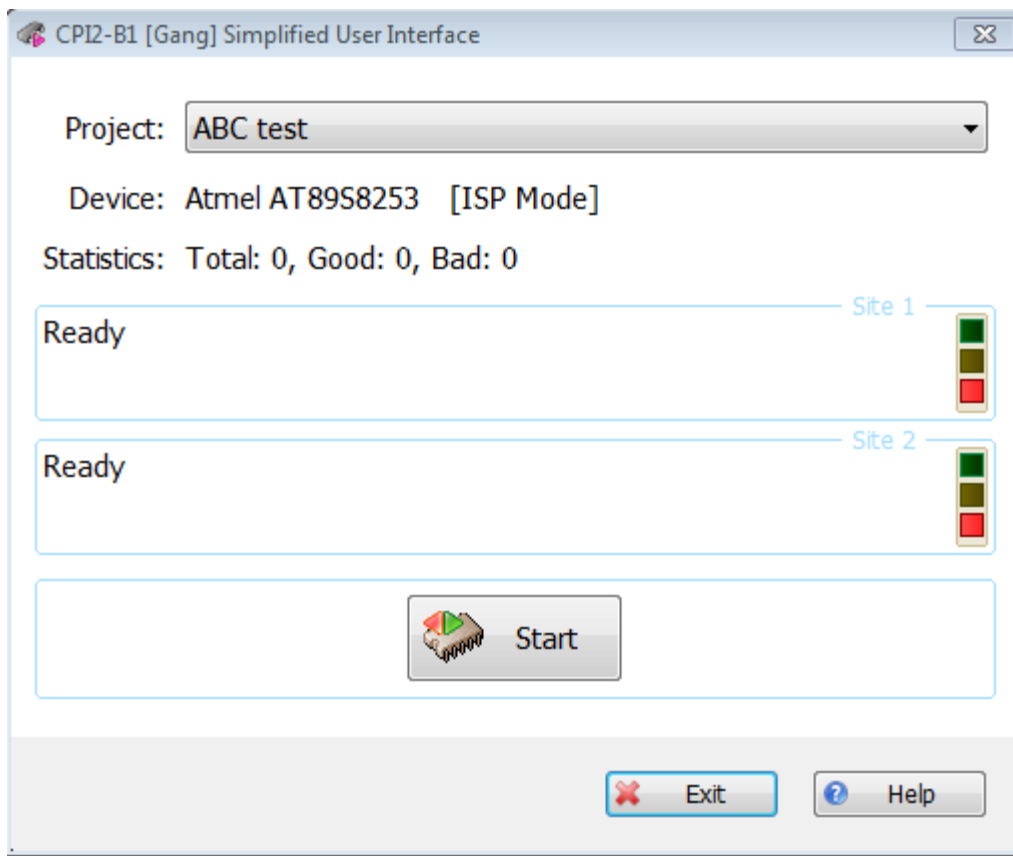
[graphical user interface](#)^[48] from the default mode to the **Simplified User Interface** mode (**SUI**). In this mode, operator can see a very simple PC screen with very limited information: a single **Start** button and three virtual LEDs that indicate CPI2-B1 status: Good, Busy or Error. Or, for the [Gang Programming](#)^[26] 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](#)^[26] 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 start 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](#); ^[47]
- can be started by executing [Auto Programming](#) ^[108] command, only.

A typical use scenario consists of two steps: [Preparation](#) ^[116] and [Use](#) ^[120].

1. [Preparation](#) ^[116]. An engineer or a technician (hereafter a supervisor) configures the programming session using the default CPI2-B1 [graphical user interface](#) ^[48] and saves the session [project](#) ^[47]. Project file can be stored at any location on PC hard drive. To [launch](#) ^[120] 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](#) ^[86] the user interface to SUI mode for use of the CPI2-B1 by a less skilled operator.
2. [Use](#) ^[120]. 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](#) ^[116]

[Operations with Simplified User Interface](#) ^[120]

3.3.1 Settings of Simplified User Interface

A session project contains information on device type, file name, [serialization](#)^[63] parameters, check sum, list of the functions included in the [Auto Programming](#)^[108] batch and other options, including the SUI [windows and controls configurations](#)^[52]. The SUI interface [settings](#)^[116] 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](#)^[57] menu - select target device.
- [Configuration](#)^[57] menu - set up a [buffer](#)^[97].
- [Configuration](#)^[57] menu - set options for device [serialization](#)^[69], writing [check sum](#)^[69] and [signatures](#)^[70], and [log file](#)^[73] controls.
- [Device and Algorithm Parameters Editor](#)^[93] window - specify the options different from default for a chosen device.
- [Program Manager](#)^[105] window > [Program Manager](#)^[107] tab > [Edit Auto](#)^[108] dialog - configure [Auto Programming](#)^[108] batch of functions.
- [Program Manager](#)^[105] window > [Program Manager](#)^[107] tab - set programming options.
- [Program Manager](#)^[105] window > [Statistics](#)^[111] 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.

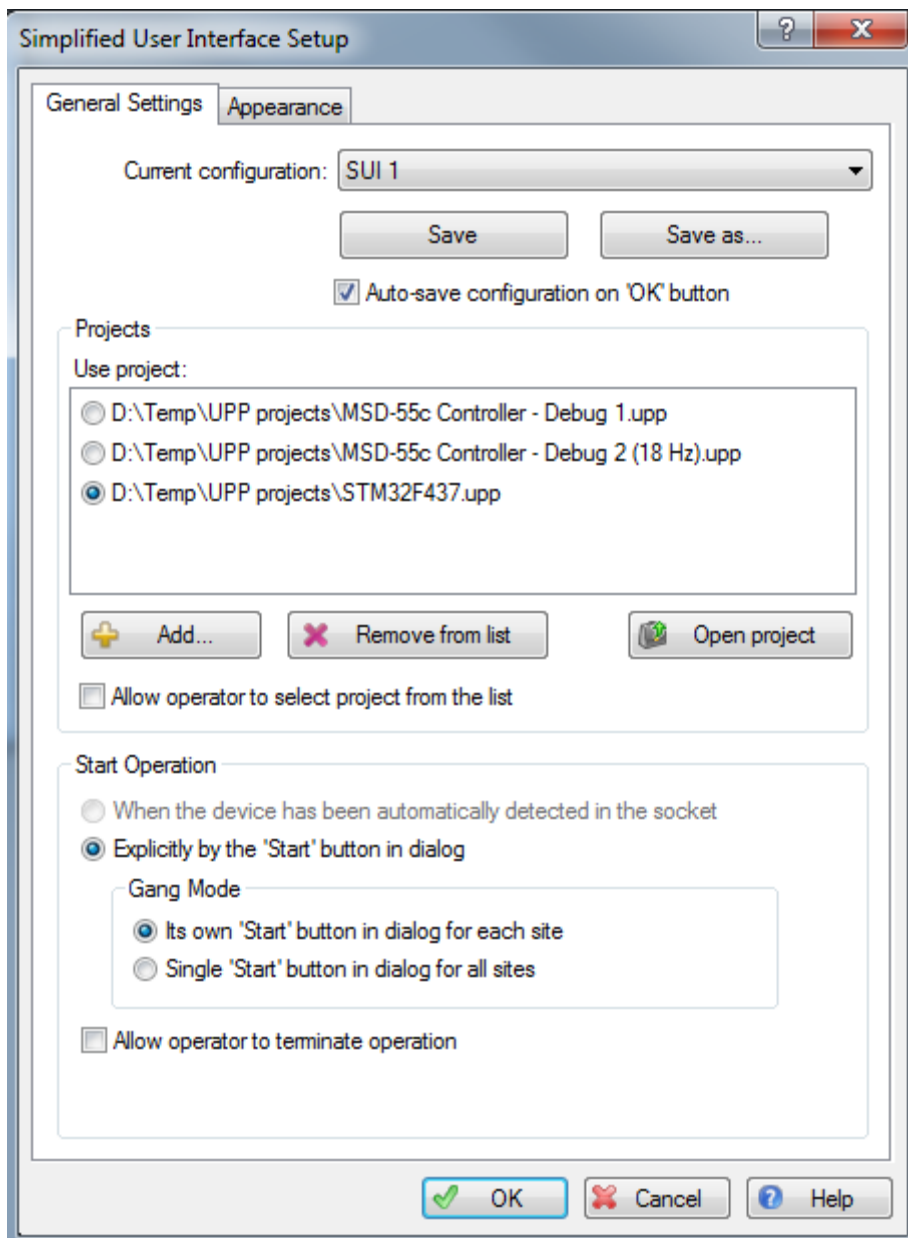
Once the above settings are done, create the project. In the menu select **Project > New**. In the [Project Options](#)^[53] dialog enter project name, file name, format, and other information. Click **OK** button to save the project to disk.

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](#)^[57] 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 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.

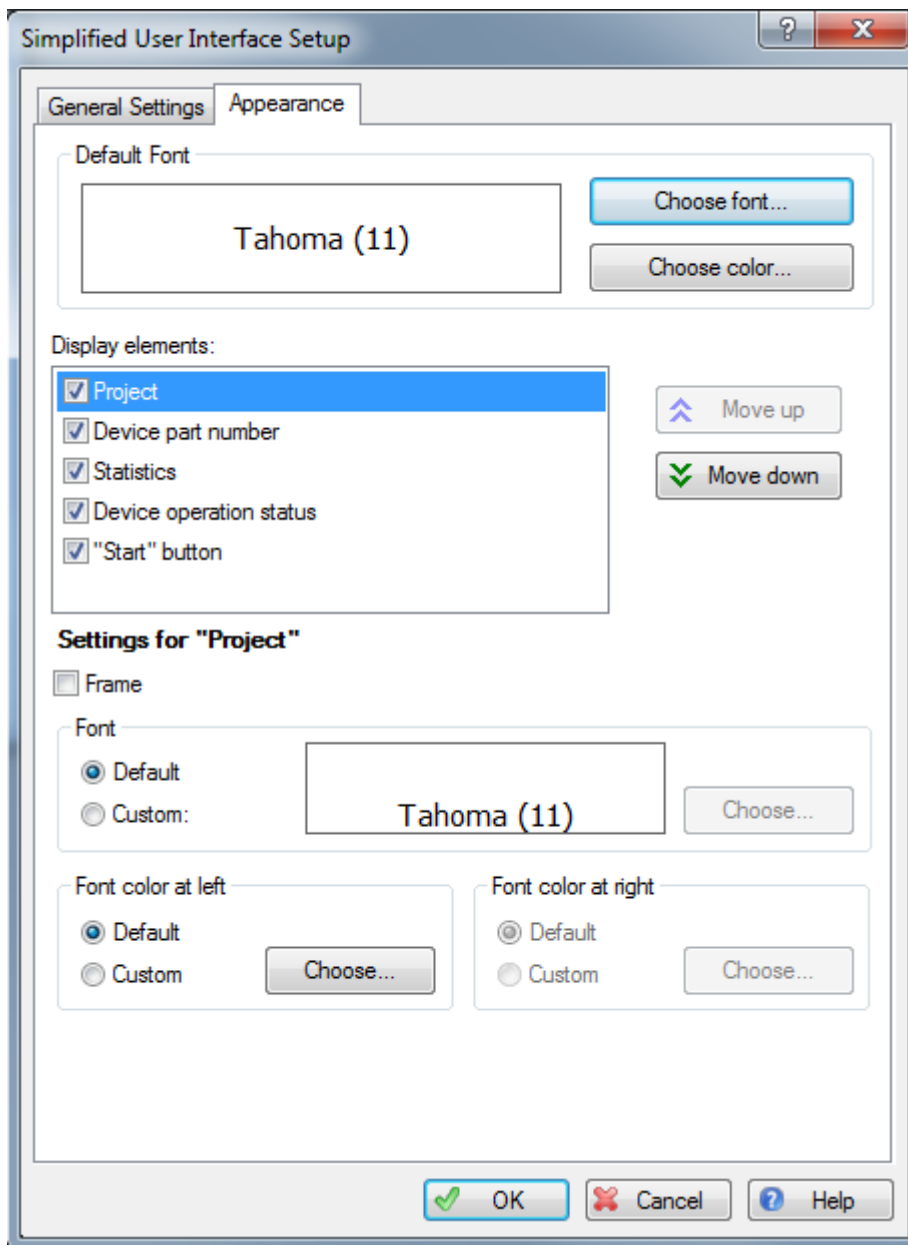
The only batch command that can be launched in SUI mode is [Auto Programming](#)^[108]. This command is executed either by pressing the physical button on the CPI2-B1 unit or by clicking the '**Start**' button in the **SUI** window.

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](#)^[69] and [check sum](#)^[69] 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](#)^[113] open the [Command](#)^[86] menu, and double click the **Switch to Simplified User Interface..** line.

To launch the ChipProg-02 with the [Simplified User Interface](#)^[113] (or in the **Simplified Mode**) use the **/Y<configuration name>** option key in [Command line](#)^[120] 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](#)^[57] > [Preferences](#)^[78] 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 the **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](#)^[47] 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](#)^[120] table.

Option names, project names, and the application executable name are **NOT** case sensitive, 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](#)^[120]

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>	<p>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^[197] (gang) mode (see below).</p> <p>The serial number can be found on the bottom of programmer case or by using the Help > About...^[90] 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.</p> <p>For example, the option -NSI2-10012 specifies that all other command line options apply to the programmer with serial number SI2-10012 only.</p>
-GANG	<p>This option launches multiple CPI2-B1 device programmers in gang-programming^[197] (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.</p> <p>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:</p>
-GANG:<number of sites>	<p>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 may be omitted.</p>
-GANG#<list of serial numbers>	<p>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.</p> <p>For example, if the -GANG#SI2-10014;SI2-10022 is specified, the application waits for establishing connections with two device programmers with serial</p>

Option	Description
	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>	<p>This option specifies an individual IP address or a range of multiple IP addresses to be pinged by a computer while it tries to connect CPI2-B1 device 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.</p> <p>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:</p> <p>-ETH#192.168.1.32 - connect a device programmer with the 192.168.1.32 static IP address.</p> <p>-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.</p> <p>-ETH#192.168.1.{16-128} - scan IP addresses in a range of 192.168.1.16 to 192.168.1.128.</p> <p>-ETH#192.168.1.* - scan IP addresses in a full range of 192.168.1.1 to 192.168.1.254.</p>

Option	Description
	<p>-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.</p> <p>-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.</p>
-C"<manufacturer^device>"	<p>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.</p> <p>For example: -C"NXP^MC9S08DV32MLF [ISP Mode]".</p> <p>Note. The use of the -C option is less beneficial than using projects^[47]. Projects provide much more flexible and effective control of device programming. Use of projects is highly recommended, especially for mass production, to create, configure, and save as many projects as needed and use them with command line.</p>
-L<file name>	<p>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^[103] from the file name extension. If actual file format differs from the one listed in the file format^[103] list use the -F option to explicitly specify file format (see below).</p>
-F<file format>	<p>This option sets format of the file specified by the -L<file name> option. The <file format> must be one of the following letters:</p> <ul style="list-style-type: none"> 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 <p>For example, -FH option loads file in the HEX format, which contains starting address in CPI2-B1 buffer.</p>

Option	Description
	<p>If binary format (B) is specified by the -F option, it should be followed by a destination starting address in the format used in C language. For example: the option -FB0xFF04 loads binary file and places data starting at the address 0xFF04 in the buffer.</p> <p>In the absence of -L<file name> the -F<file format> option is ignored.</p>
-A[buffer number]	<p>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.</p> <p>The -A option is only meaningful if a project name or an -L<file name> option is also specified on the same command line.</p>
-I	<p>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.</p>
-I1	<p>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.</p>
-I2	<p>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.</p>
-M	<p>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</p>

Option	Description
	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 ⁵² 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 ⁵² 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 ⁵² 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.

3.4.2 Command Line Option Files

Command line options can be specified directly or by command line option files - **response files**. Instead of specifying a command line option it is possible to put a character **@** following by a name of the file that includes the command line options. This character **@** following by a file name can be inserted in the command line anywhere. The ChipProg-02 reads the option file and inserts its content into the command line. For example, specifying the command line as:

UProgNT2.exe -G1 @C:\Files\Programmer.txt

where the **C:\Files\Programmer.txt** file includes the following lines:

```
-LF:\ARM\IAR\CPP\Debug\Exe\Test.hex
-FHEX
-A
-I2
```

is equivalent to specifying the command line:

UProgNT2.exe -G1 -LF:\ARM\IARCPP\Debug\Exe\Test.hex -FHEX -A -I2

Each line in a response file may include one or more options. Blank lines, lines beginning with the a semicolon (;) or double slash (//) characters are treated as comments and ignored them. For example, see the **C:\Files\Programmer.txt** file with added comments:

```
; ----- Load file to buffer F:\ARM\IARCPP\Debug\Exe\Test.hex
-LF:\ARM\IARCPP\Debug\Exe\Test.hex

; ----- Specify the HEX file format
-FHEX

; ----- Conduct Auto Programming
-A

; ----- Hide GUI. Copy error messages to clipboard.
-I2
```

A command line may include several response files. For example:

UProgNT2.exe @F:\Config1 @C:\Files\Programmer.txt

It is also allowed to include one response file into another - then a command line interpreter will extract all the options of both response files.

3.5 On-the-Fly Control Interface

The **On-the-Fly Control** interface is very similar to [command line](#)^[120] 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](#)^[47], execute [script](#)^[176], 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](#).^[172]

The **On-the-Fly Control utility** is an alternative to a more advanced [Application Control Interface](#)^[158] ([DLL control](#)^[158]); 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](#)^[131]. 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](#)^[104] window and, optionally, to [log file](#)^[73] 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](#)^[127] 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](#)^[132].

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.
-G+][<log file name and path>]	Send the ChipProg-02 Console ^[104] 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.

Option	Description
	<p>Examples:</p> <ul style="list-style-type: none"> -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 ^[104] window before it terminates.
-L< ChipProg-02 executable file name and command line options>	<p>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).</p> <p>Example: -L"UProgNT2.exe /g1"</p>
-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	<p>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.</p> <p>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).</p>
-M[=<timeout in seconds>]	<p>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 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.</p> <p>If the -M option is not accompanied by a [=<timeout in seconds>] parameter, On-the-Fly Control program will wait for the programmer</p>

Option	Description
	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>	<p>Load the specified project¹⁷⁶ 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.).</p> <p>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.</p> <p>Example: -P"C:\Prog\Projects\Antenna-01 Test.upp"</p>
-S<script file>	<p>Run the specified script¹⁷⁶. 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.</p> <p>Example: -S"D:\Prog Scripts\Checksum.cmd"</p>
-NWS	Do not wait for completion of the script ¹⁷⁶ specified by the -S option.
-F<function list>	<p>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 (*).</p> <p>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.</p> <p>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.</p>

Option	Description
	<p>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.</p> <p>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) .</p> <p>Examples:</p> <ul style="list-style-type: none"> -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>]	<p>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").</p> <p>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%.</p> <p>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.</p> <p>Examples:</p> <ul style="list-style-type: none"> -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

Option	Description
	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](#)^[104] 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 target device.
2	Invalid option or parameter on command line ^[120] .
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 another 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 Standalone Operation Mode

CPI2-B1 device programmers can be operated in the **standalone mode** that does not require a computer for driving device programmers. The major difference between the computer and standalone control modes is a physical location of the memory which stores the data to be programmed into target devices. These data includes:

- Target device type;

- Static data - usually the same code, which should be replicated inside of a series devices belonging to the same type;
- Dynamically changing data, unique for each device in the series: [serial numbers](#)^[63], [signatures](#)^[70], date stamps, etc.
- User-specified [Device and Algorithm Parameters](#)^[93].
- Factory programmed serial number of the CPI2-B1 units.

While the programmer is under computer control, all the data above form an image (or several images) *physically* located in the computer RAM. In case of the standalone control mode, these images are *physically* stored on an SD card inside of a CPI2-B1 .

An SD card is a kind of flash memory media that connects directly to master MCU inside the programmer. Storing data on this media **inside of device programmer** enables much faster streaming the data into a target device. Moreover, even if the programmer is controlled by a PC, utilizing the benefits of faster streaming images from SD cards to target devices allows to speeding up the mass programming. To store the data image on the SD card for both standalone and faster PC control modes, the data should be first configured in the [GUI](#)^[48] mode and then [cached](#)^[134] onto SD cards. Capacity of SD card used in the CPI2-B1 device programmer may vary from 8 to 64 GB - Phyton has the rights to use the cards of any capacity without prior notice.

Preparing the data above for the standalone control **unavoidably requires use of [projects](#)**^[47]. A user should conduct the following steps:

1. In the [GUI](#)^[48] mode create a project and store it on a computer as a file with the .UPP extension. The project should include all the data and parameters above - target device type, static and dynamically changing data, etc.
2. Connect the device specified in the project to a device programmer;
3. Enable data [caching](#)^[134].
4. Program *one* device using the [Auto Programming](#)^[108] command.
5. Assign a number for the created [Standalone Job](#)^[136].

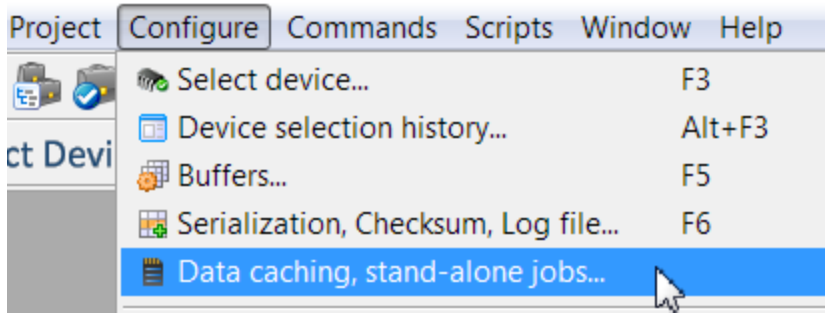
Upon completion of the steps above the programmer creates a replica of the project above on the SD card. A project replicated on the SD card is hereafter called as a [Standalone Job](#)^[136]. The programmer enables to create and to store as many as 256 independent standalone jobs, which can be launched in the GUI by the job number. Only 32 of them can be assigned for launching by applying a 5-bit code to the JOB_SEL0...JOB_SEL4 inputs on the connector CONTROL. After the job is selected by the JOB_SEL code, it launches by applying the START signal to the CONTROL [connector](#)^[24] or by pressing the Start button on the top panel .

4.1 Preparing Standalone Mode Jobs

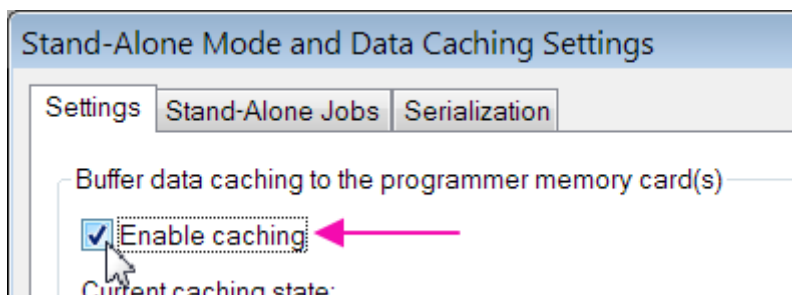
Preparing of use a CPI2-B1 device programmers in the Standalone Mode (SA) includes the following steps:

- Enabling data caching;
- Creating projects to be run in standalone mode;
- Converting these projects into standalone jobs by caching data on the embedded SD card;
- Assigning numbers to prepared standalone jobs, so they can be called by a certain number by the Start signal or from the GUI;

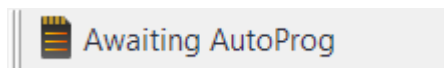
Open the [Configure](#)^[57] > **Data caching, Standalone jobs...** menu:



This will open the **Standalone Mode and Data caching Settings** dialog. Open the **Settings** tab and check the **Enable caching** check box.



This is the first step. The green text in this dialog indicates a real data caching status. This status is also indicated by the icon



located in the top right corner of the main window, at the right of the button **Auto** on main toolbar.

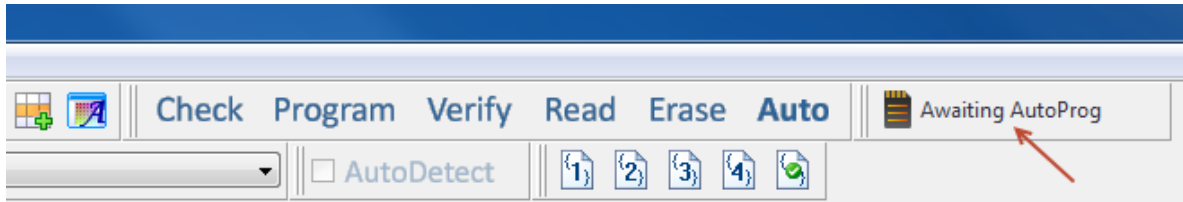
4.1.1 Data Caching

Data Caching is a process of copying data to be programmed into the target device onto the SD card inside of the CPI2-B1 device programmer. Then the programmer streams the data stored on the SD card to the target device instead of streaming them from the computer RAM that greatly speeds up all the programming operations.

This mechanism of fast programming can be used in two ways: a) in true standalone mode when the programmer is disconnected from a PC and is controlled by electrical signals from a fixture or ATE and b) when it is driven from the ChipProg-02 GUI. Taking data for device programming from the SD card allows to speed up device programming in the computer control mode as well as in the true standalone mode.

If you plan to control the programmer by electrical signals from your fixture or ATE in the true standalone mode, caching data to the SD card is a mandatory. But, if you plan to control the programmer from the ChipProg-02 GUI, data caching makes sense in case of programming devices with relatively large flash memory, only. Otherwise, the time spent of the data caching procedure will not be paid off by saving time on the faster device flashing.

Data Caching function is off by default. The Data caching status is displayed in a very right position of the ChipProg-02 main window toolbar:



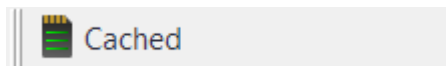
The following Data Caching statuses can be observed when you operate in the standalone mode:

No SD card(s)	The SD card was not found by the programmer's hardware or it malfunctions. Data caching is not possible.
Caching is off	Data caching is turned off by an operator.
Awaiting AutoProg	The programmer is ready to cash data. To perform caching, start Auto Programming ^[108] operation on the target device.
Cached	Data caching was completed. Since then the programmer will take data cashed on the SD card, not from the buffer.
Cached, Job: #2	Data caching was completed. Project was assigned to standalone job with a specified number (#2 here) and is ready for launching.

To bring up settings for caching, standalone jobs, and serialization, click on the image of caching status (**Awaiting AutoProg**), or use the [Configure](#)^[57] > **Data caching, Standalone jobs...** menu command.

First, create a project that can be then convert to a standalone job. Use a universal mechanism of creating a project described in the [Project Options Dialog](#)^[53]. What is important is to name each project to simplify assigning a number for each standalone job and to easily navigate in the line of jobs in the Stand-Alone Jobs tab.

After storing your first project under a certain name (for example, RTX-12 (2016-11-21)) connect a target device specified in this project to the CPI2-B1 device programmer and click the **Auto** button on the main toolbar or click twice on the **Auto Programming** line in the **Project Manager** window. If the data caching was [enabled](#)^[133], the first run of the **Auto Programming** macro command ends with issuing a short warning "Accessing memory card(s)" and the icon



will appear in the right position in the main toolbar. Similarly, you can create and store on one SD card(s) as many standalone jobs as you need - up to 256. ChipProg-02 application automatically assigns numbers to each job from the #0 up to #255.

The ChipProg-02 program uses the following rules of assigning job numbers stored on the SD card:

- If there is an open project in the ChipProg-02 GUI, the program searches a standalone job with the same name on the SD card. If it finds such an SA job, the program updates it with the data and parameters stored in the project in the GUI. If the program cannot find the SA job with the same name, then the program assigns to this job the lowest number, not taken yet by an unnamed project.

If there is no such an unnamed project, the ChipProg-02 application assigns the lowest available number. If there are no free numbers issues an error message.

- If there is no open project in the ChipProg-02 GUI, then the cached data are considered as a "unnamed job". Then the program checks whether the SD card already stores another job with the same parameters, the caching procedure completes. Otherwise, the program assigns the "oldest" number earlier assigned an unnamed job. If there is no an available jobs to be assigned, the program issues an error message.

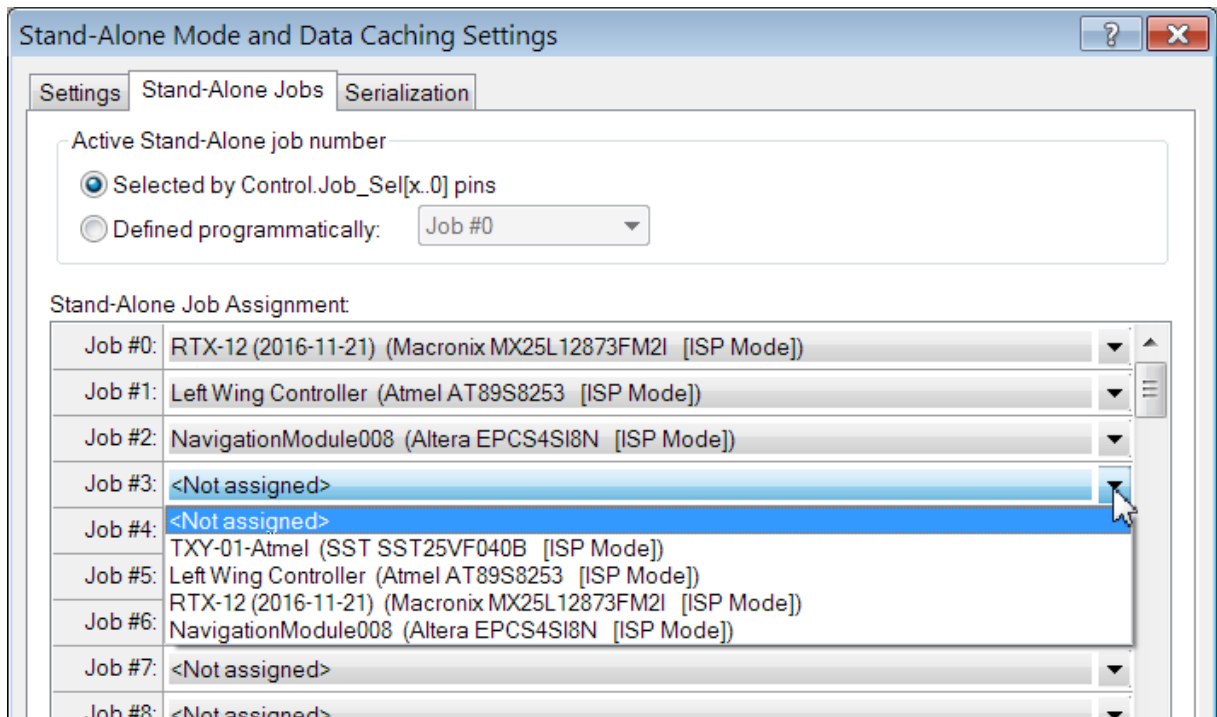
All created SA jobs are visible in the **Stand-Alone Jobs** tab of the dialog. First 32 SA jobs addressed by the 5-bit JOB_SEL0...JOB_SEL4 selector can be launched by applying the START signal to the CONTROL [connector](#)^[24] or by pressing the Start button on the top panel..

4.1.2 Standalone Jobs

All SA jobs, created by caching data to SD card(s), are visible in the **Stand-Alone Jobs** tab of the **Standalone Mode and Data caching Settings** dialog. Any SA job can be launched either by electrical signals applied to the [CONTROL](#)^[24] connector (below displayed as "**Selected by Control.Job_Sel[x...0]**") or from the ChipProg-02 GUI (displayed in the tab as "**Defined programmatically**").

First 32 SA jobs addressed by the 5-bit JOB_SEL0...JOB_SEL4 selector can be launched by applying the START signal to the CONTROL [connector](#)^[24] or by pressing the **Start** button on the top panel..

In the **Stand-Alone Jobs** tab picture below you can see how to assign SA job numbers. The jobs are displayed here in the ascending numerical order: from 0 to 255. Click the ↓ (arrow down) sign at the job line to open the list of cached SA jobs, pick the project name to assign the job number. Only a named project can be associated with an autonomous job. Each project can only be associated with a single job.

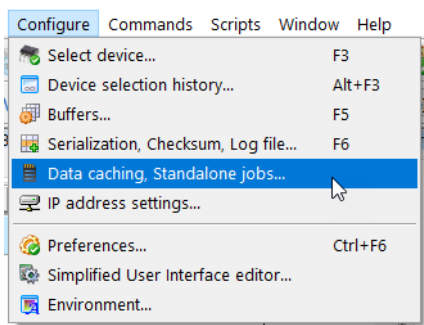


Two radio buttons "**Selected by Control.Job_Sel[x...0]**") and "**Defined programmatically**" enables to choose a method of the standalone launching. Clicking the **OK** button at the bottom of the dialog window fixes the method of launching standalone jobs: by ATE signals or from the GUI.

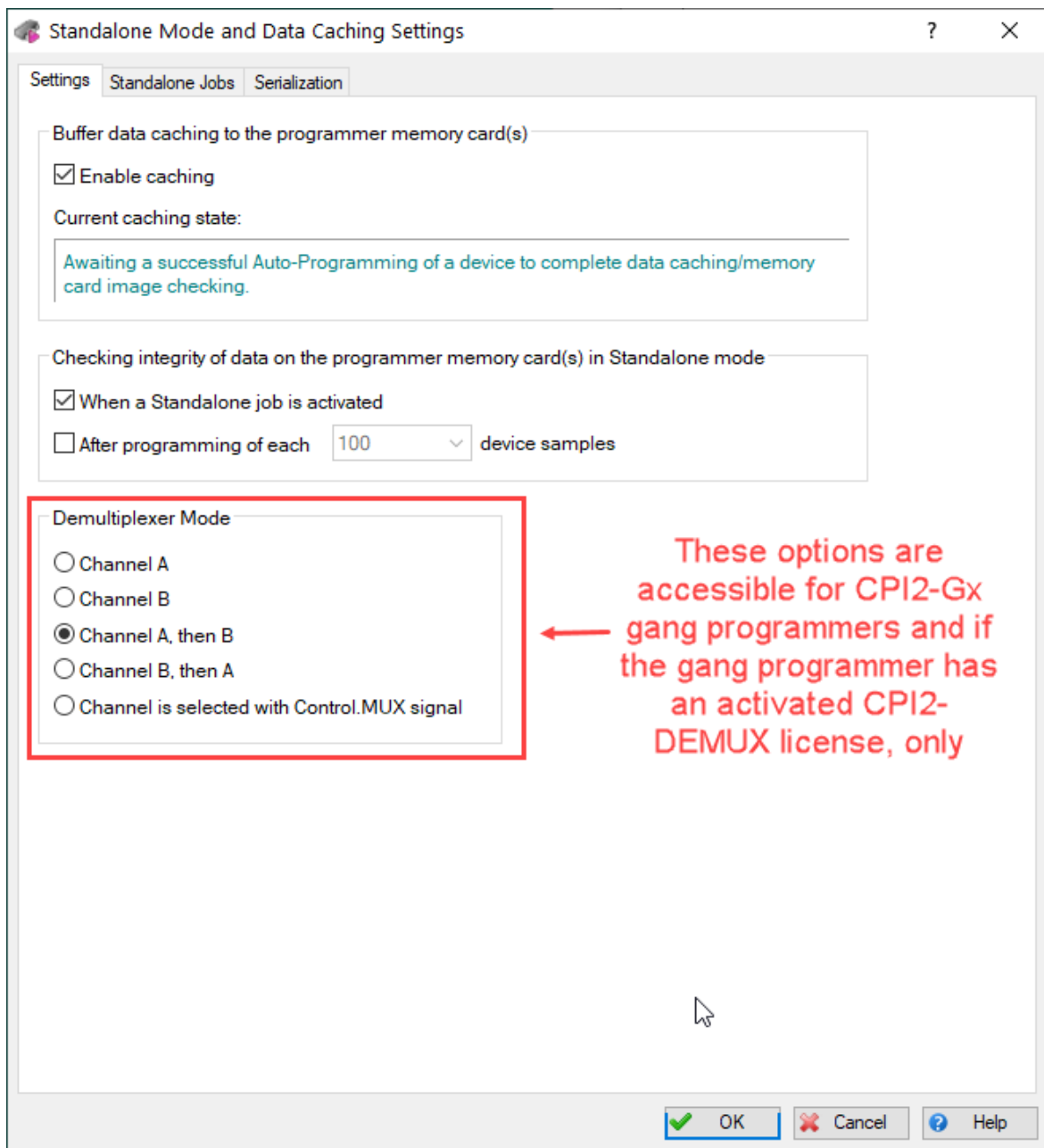
After assigning a number in the **Stand-Alone Jobs** tab a project becomes a standalone job. This job physically locates on the SD card, it has a unique number and can be launched by this number either by the ATE signals or by a mouse click from the GUI. However, any SA job can be updated by adding dynamically changing data ([Serialization](#)^[138]) and a [limitation](#)^[141] of the devices to be programmed that is described in the following chapters.

4.1.3 Standalone mode settings

To setup the standalone mode options open the **Configure > Data caching, Standalone jobs...** dialog:



The dialog enables to set all possible standalone mode options:

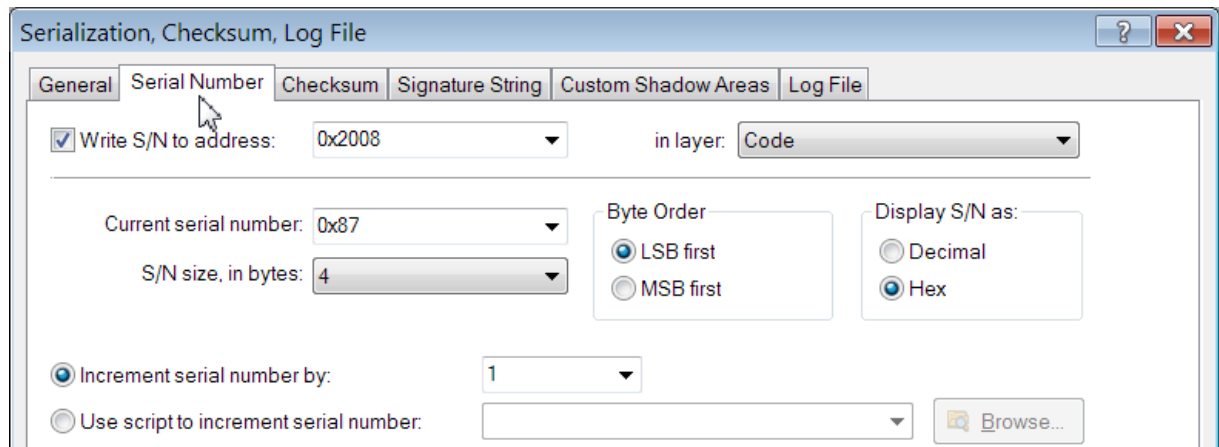


If you operate with the CPI2-B1 device programmer in the Standalone mode the **Enable caching** box must be checked. When it is unchecked, the programmer can be operated from the [GUI](#)^[48], only.

4.1.4 Device serialization

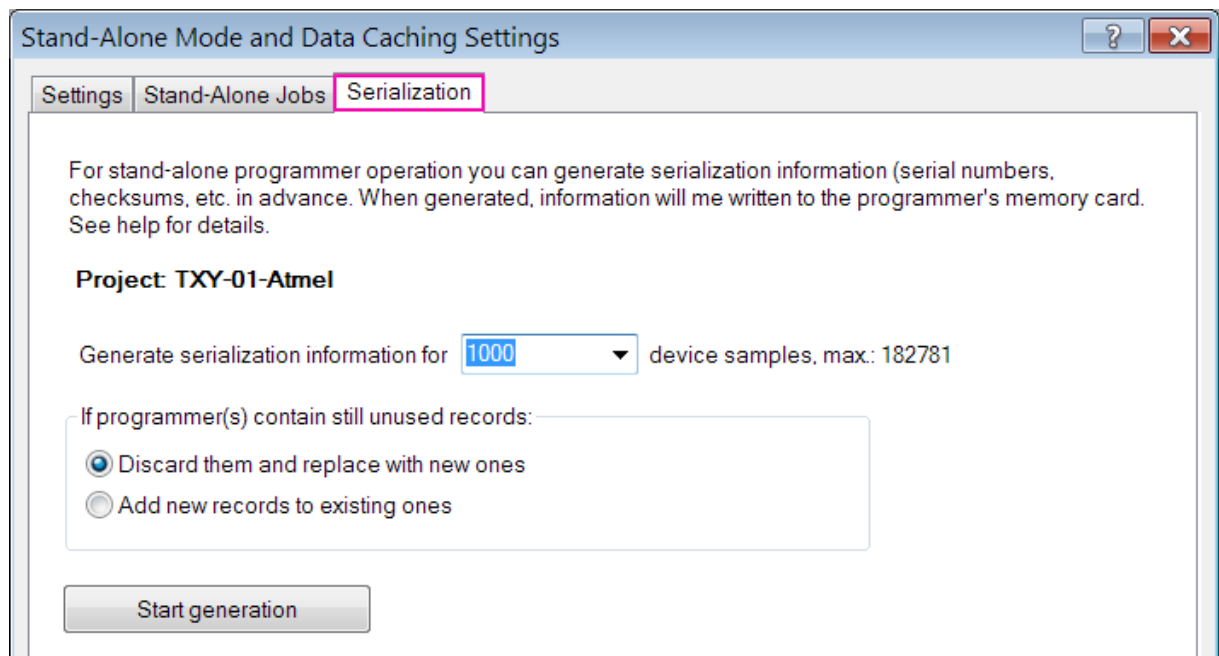
Very often the image to be written into the target device is comprised of the static data, common for all the devices to be programmed in one session, and the data unique for each device in this series. Usually such data represent unique [serial numbers](#)^[69], [checksums](#)^[69], [signatures](#)^[70] and custom data stored in the [custom shadow memory](#)^[71] areas. Such dynamically changed data blend with static data before physical writing the image into the target device. The ChipProg-02 enables to program complex images in the standalone mode as well in the computer controlled mode.

Dynamically changed data mentioned above should be prepared in the project by means of the [serial numbers](#)^[69], [checksums](#)^[69], [signatures](#)^[70] and [custom shadow memory](#)^[71] dialogs below.



In context of the standalone programming, preparing all dynamically changes data is defined here in one term: **"Serialization"**. Each SA job has its own serialization settings. These settings must be done *before* generating serialization information for standalone mode. See the picture 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 it is shown below.



Serialization information is stored in a fixed part of the SD card memory. The maximum number of target devices is the project specific. For example, in the picture above the maximal number of the devices to be programmed is 182781.

When operating in standalone mode, the 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 written into devices, the ChipProg-02 terminates the programming process and issues an error message. To continue programming process, additional serialization information should be generated.

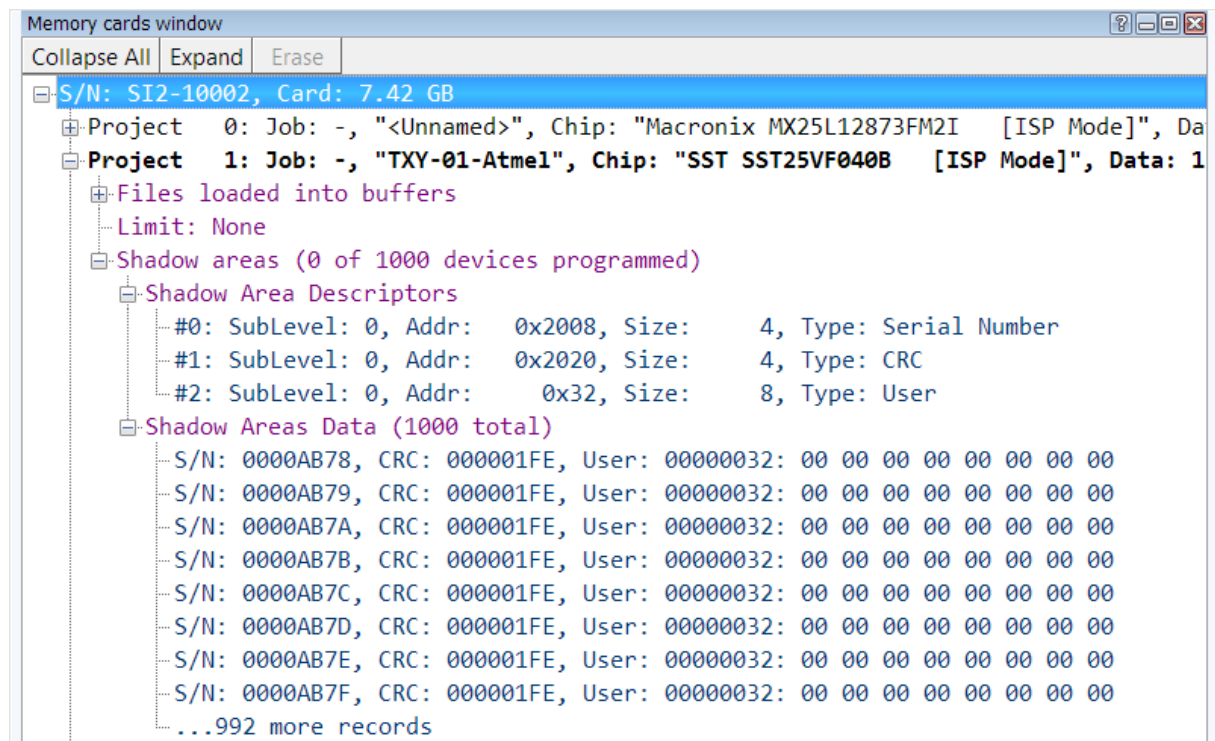
The dialog above enables two alternative options of how to handle unused records if they remain in the programmer - either **to discard** them and replace with new ones or **to add new** records to remaining unused. In the last case, the added serial number will continue to carry out the number of numbers.

Since it is impossible to predict a capacity of free memory on the SD card that can be assigned for the serialization information, the **Serialization** records can be generated by a new data caching, only.

Note! To perform a new generation click the "Start Generation" button in the dialog. Clicking the "OK" button at the bottom of the dialog does not start generation of the device serialization in the standalone mode.

If multiple CPI2-B1 device programmers run in the [gang](#)^[197] mode, serialization records are equally distributed among SD cards in all CPI2-B1 device programmers joint in the gang cluster .

Current serialization information can be viewed in the [Memory cards window](#)^[142] (see an example below). In this window serialization records are called "shadow areas" (which they actually are).



Limitations of Serialization in Standalone Mode

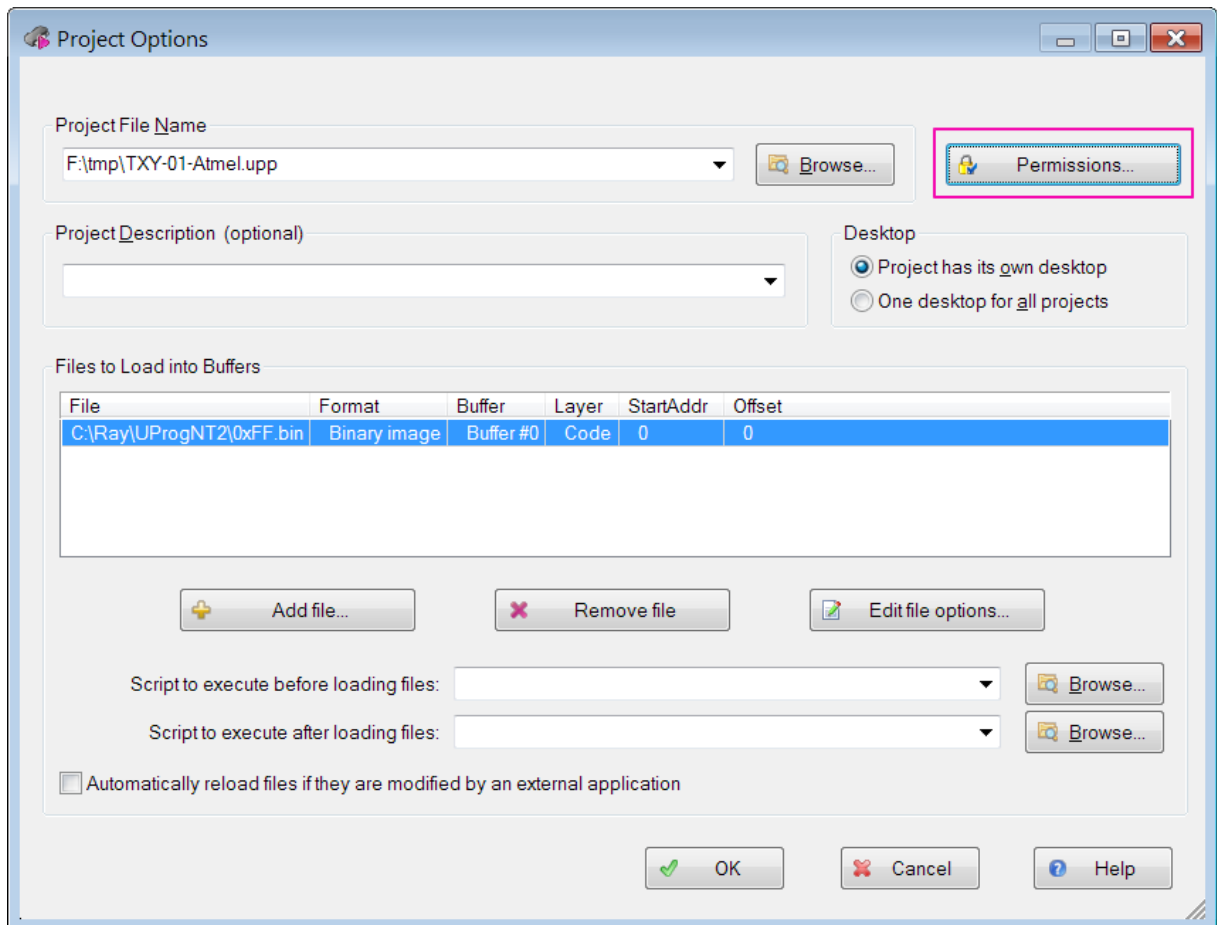
Besides a necessity to remember to add in serialization records in time, the following limitations should be kept in mind:

- If programming of a target device causes an error, serialization record is still used up, in spite of the application program settings. In such case, serial numbers of target devices will not remain consecutive, they will include gaps.

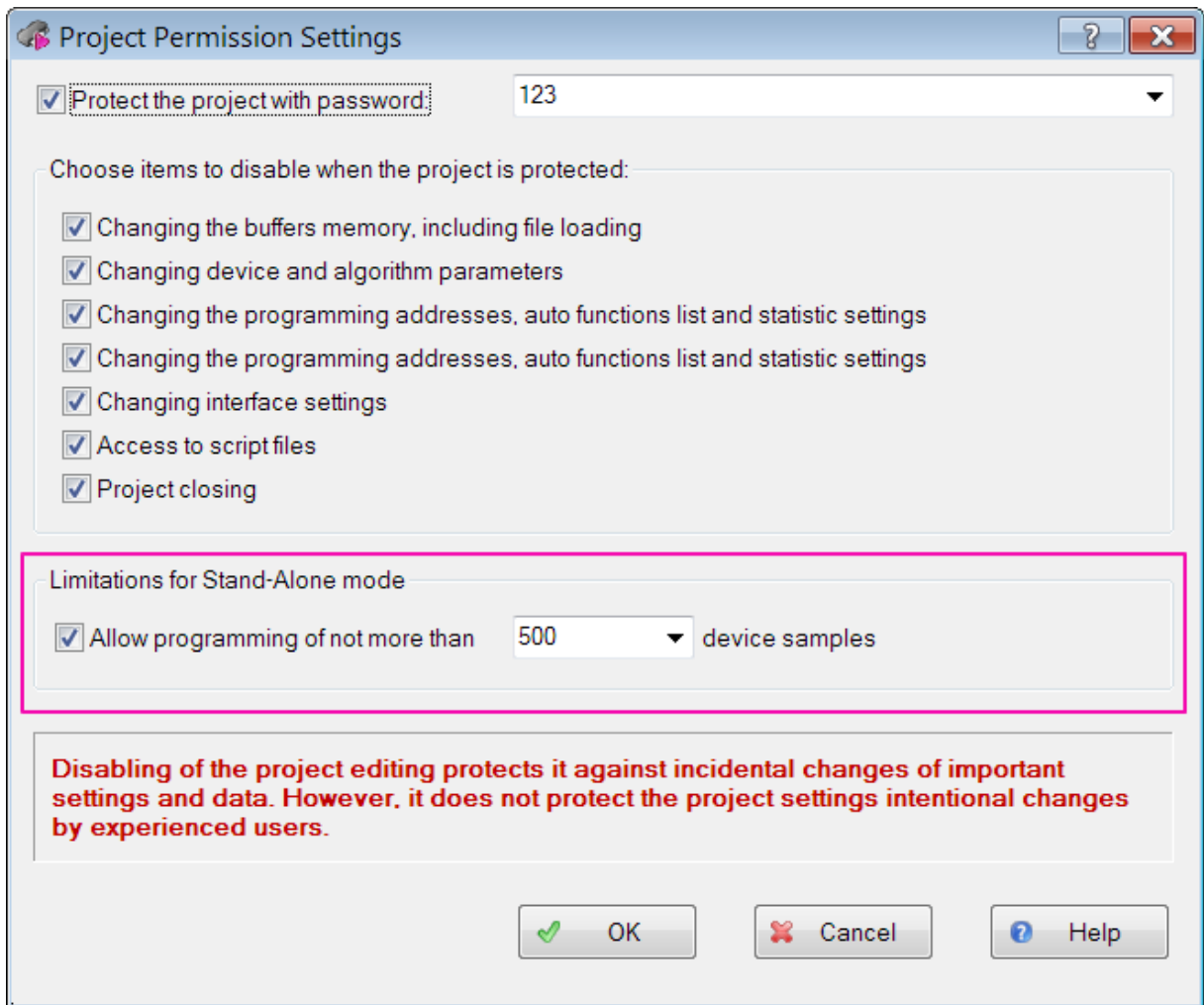
- If you use [scripts](#)¹⁷⁶ for generating serial numbers, checksums, and other dynamically changing data take in account the difference of launching the scripts in ChipProg-02 application. In the GUI control mode scripts launch immediately *before* programming of a next target device. However, when generating records for standalone mode, scripts launch immediately *after* generation of a next record. If the 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.

4.1.5 Permissions and setting limits

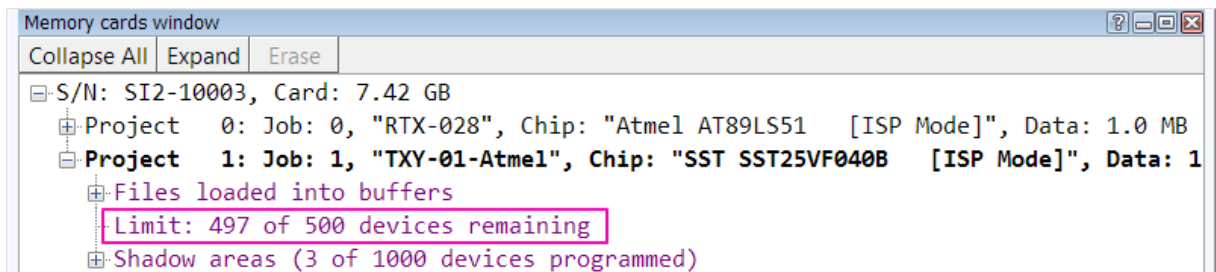
A CPI2-B1 user is able to set the number of target devices to be programmed in standalone mode. Before setting the limit this function should be permitted through the [Project Option](#)⁵³ dialog. Open the dialog, browse the project file (.UPP file) and click the button [Permissions](#)⁷⁸.



This brings up the **Project Permission Settings** dialog, in which you can specify the number of devices to be programmed. To enable this setting, you must check the **Protect the project with password** box and specify a password.



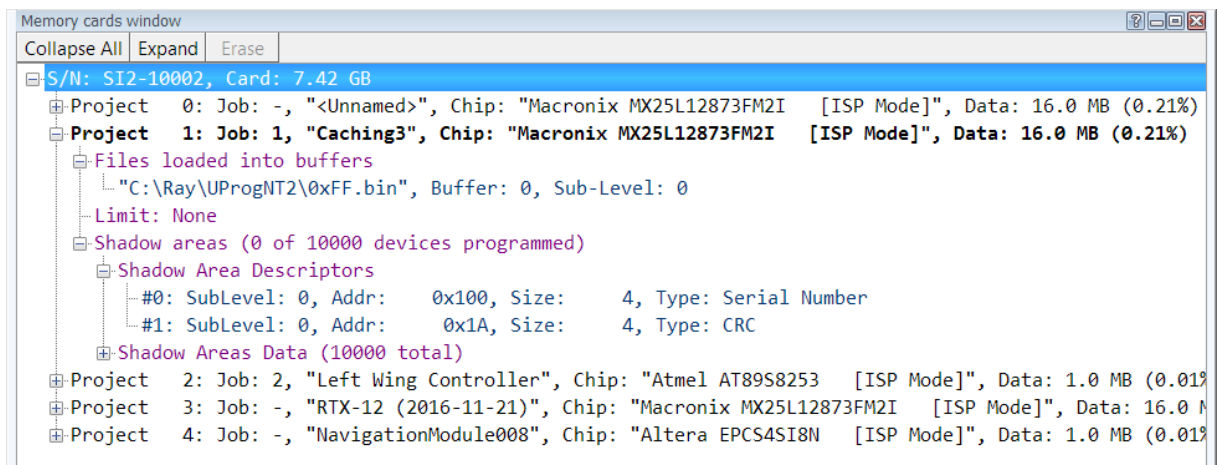
A current state of the device counter can be monitored in the [Memory cards](#)¹⁴² window. See an example below.



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.

4.1.6 SD card window

Memory cards (or SD cards) window can be used to examine information stored on the card, as shown on the figure below. Use the [View](#)⁵² menu to open this window.



During subsequent programming operations, the programmer uses buffer layers data from SD card. The ChipProg-02 application tracks changes in the settings that may cause modification of data stored in the SD card. If necessary the program launches data re-caching. This may be triggered by the following changes:

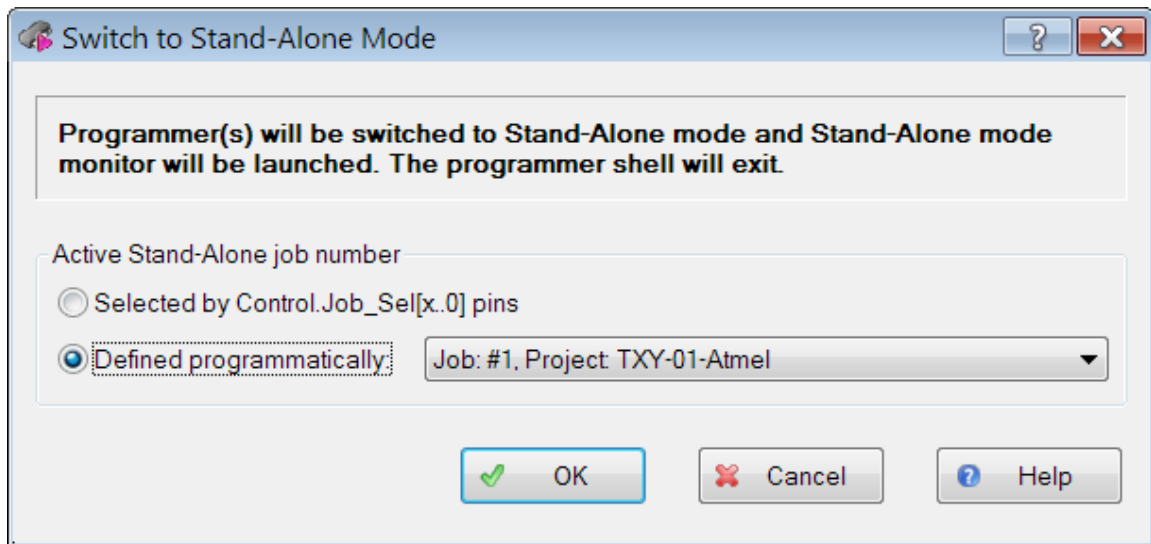
- Writing data into the memory buffer - manually or by reading a file or by a script or communication via the [ACI](#)^[158];
- Modification of the target device settings;
- Modification of [serialization](#)^[138] settings;
- Modification of the [Auto Programming](#)^[108] parameters.

4.2 Switching to Standalone Mode

After powering-up, a CPI2-B1 device programmer keeps staying in the idle mode until it will be launched either in the computer controlled mode from the CPI2-B1 [startup](#)^[20] dialog or in the [standalone](#)^[132] mode. In turn, launching the programmer in the standalone mode can be done either programmatically, or by applying electrical signals to appropriate pins on the connector [CONTROL](#)^[24].

*Launching the CPI2-B1 in the SA mode **programmatically** can be done in two ways:*

- From the ChipProg-02 GUI menu **Commands -> Standalone Mode**. This will open the **Switch to Standalone Mode** dialog below. In this dialog you can specify a method of selecting SA jobs - by the signals applied to the connector [CONTROL](#)^[24] or programmatically from the drop down Job menu in this dialog.



- By clicking the **Start Standalone Mode Monitor** button in the CPI2-B1 [startup](#)^[20] dialog. Or just by calling the **SAMonitor.EXE** executable that locates in the same folder where the ChipProg-02 was installed. This will open the [Standalone Mode Monitor](#)^[145] window (read the next chapter).

Launching the CPI2-B1 in the SA mode by applying electrical signals from ATE can be done by one the way below

- By applying a logical 1 signal to the SAMODE pin of the connector [CONTROL](#)^[24] right after powering the CPI2-B1 unit, while it remains in the idle mode.
- By applying and holding for at least 2 sec logical 0 signal to the START pin of connector [CONTROL](#)^[24].
- By pressing and holding for at least 2 seconds the START button on the programmer top panel..

Once CPI2-B1 switches to standalone (SA) mode, the green (GOOD) and red (ERROR) LEDs start blinking. These LEDs will keep blinking until the programmer is switched back to computer controlled mode. When the CPI2-B1 remains running in the SA mode, a SA job can be launched by either one of the signals above.

ChipProg-02 software allows real-time monitoring of activity device programmers driven by this software by a special utility - the [Standalone Mode Monitor](#)^[145]. This monitor window displays status(es) of the device programmer(s) along with a current Standalone Job number, device counters, statistics of failures, and other useful information.

To interact with ATE or test fixture CPI2-B1 device programmers running in the standalone mode output three status signals onto appropriate pins of the connector [CONTROL](#)^[24]: BUSY, GOOD and ERROR. These signals - log. 0 means active - indicate statuses of the device programming operations:

- BUSY=log.0 while the operation lasts, then returns to the log.1 state,
- GOOD=log.0 in case the device was programmed and successfully verified and stays low until a new programming cycle starts;
- ERROR= log.0 in case of failure.

These signals, outputted to the connector CONTROL, are duplicated by, respectfully, yellow, green and red LEDs on the top panel of the CPI2-B1 units.

If the programming 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, not programmatically. For this purpose, the [CONTROL](#)^[24] connector contains five pins (Job_Sel [4..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, the Job #0 will be automatically selected..

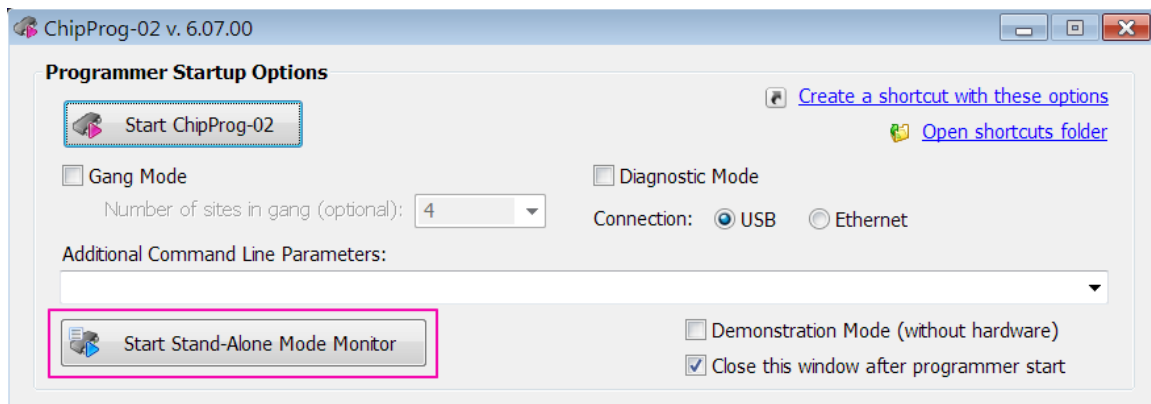
If multiple CPI2-B1 device programmers run in the [gang](#)^[197] mode, the ChipProg-02 program takes care of synchronizing Standalone Mode Jobs on all single device programmers in the gang cluster.

4.3 Standalone Mode Monitor

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

The application can be launched in the following two ways:

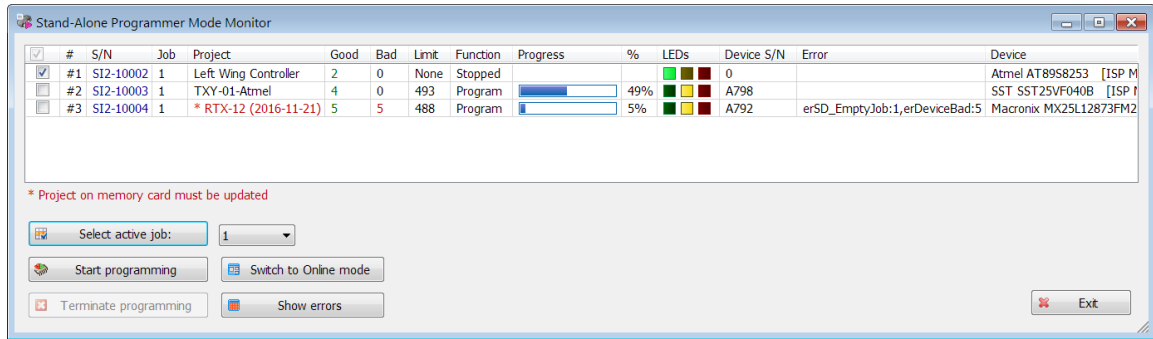
- By clicking the **Start Standalone Mode Monitor** button in the in the CPI2-B1 [startup](#)^[20] dialog below. Or just by calling the **SAMonitor.EXE** executable that locates in the same folder where the ChipProg-02 was installed.



- From the menu "**Commands**" -> "**Switch to standalone mode** in the GUI mode.

Being launched in one way or another, the **Standalone Mode Monitor** switches all the programmers, which it is able to communicate with to, into the [standalone mode](#)^[132] (**SA mode**) unless these units are already running in the SA mode. The Monitor can "see" only those programmers which are not being used at the moment by the ChipProg-02 application; this is because at any given time a programmer cannot be be under control under more than one application. On the other hand, the ChipProg-02 application does not "see" the programmers already running in the SA mode.

The **Standalone Mode Monitor** does not disturb running the launched programmers; it does not slow them down. The monitor displays their current state, only. See below an example of the **Standalone Mode Monitor** window for a gang cluster comprised of three CPI2-B1 device programmers with serial numbers SI2-10002, SI2-10003 and SI2-10004.



Where:

#	The programmer number in the list.
S/N	Serial number of the device programmer.
Job	Order number of the active SA job.
Project	Name of the project associated with the SA job (it specifies the data being written into the target device).
Good	Counter of successfully programmed devices. This counter resets to zero when the programmer is powered off or when a new job is selected.
Bad	Counter of devices programmed with errors.
Limit	Number of devices remaining before achieving the limit defined in the project settings. Limit counters are preserved upon powering off the programmer.
Function	Name of the currently performed programming function.
Progress	Indicator of the function execution process.
%	Percent completeness of the function.
LEDs	LEDs that indicate the programmer status.
Device S/N	Current target device serial number, if it was defined in the standalone mode serialization settings.
Error	Error codes following the error counts. Programmer keeps up to 8 types of errors.
Device	Target device as it was selected in the project.

All buttons in the dialog above are exclusively applicable to the CPI2-B1 device programmers marked in the check boxes in the very left column of the Monitor window.

Select Active Job button: For selected device programmers, set active SA job number in the field at the right of the button. Setting the SA job number by itself does not activate the job - only clicking this button does activate it. If the selected job was not associated with any project, then an attempt to start programming aborts with an error **SD_EmptyJob**. The **Select Active Job** button is accessible if all selected device programmers are stopped, only.

Start Programming button: Start device programming on selected CPI2-B1 device programmers that are currently in the stopped state.

Terminate Programming button: Abort target device operations on all currently selected CPI2-B1 device programmers. Completing of this command can be delayed for a while.

Switch to Online Mode button: Here **Online** mode means the computer controlled mode. Clicking this button immediately switches selected device programmers from the SA mode into the GUI computer-controlled mode. This could be used for restarting the **Standalone Mode Monitor** and to make the programmers running in the SA mode visible for the GUI. The problem is that the ChipProg-02 GUI does not "see" the programmers running in the SA mode. Once the programmers are switched into the online (computer controlled) mode, the **Monitor** is no longer able to communicate with them. For refreshing communications between the GUI and the **Standalone Mode Monitor** it should be restarted. Then the monitor re-establish communications with the device programmers.

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

If a project name is displayed in **red characters**, this indicates that the project data were written by an older version of the ChipProg-02 software and must be refreshed. In many cases this is crucial because updating the ChipProg-02 version automatically causes updating the CPI2-B1 firmware that include device programming drivers. If you see the project name displayed by **red characters** you must cycle the power, launch the ChipProg-02 in the GUI mode, open the project, make sure the [data caching](#)^[134] is enabled, connect a board with a target device selected in the project to the programmer and launch the [Auto Programming](#)^[108] command once to re-cash the project data on the programmer's SD card.

4.4 Example of Setting Up Standalone Mode

This example lists all operations necessary for setting up a standalone mode.

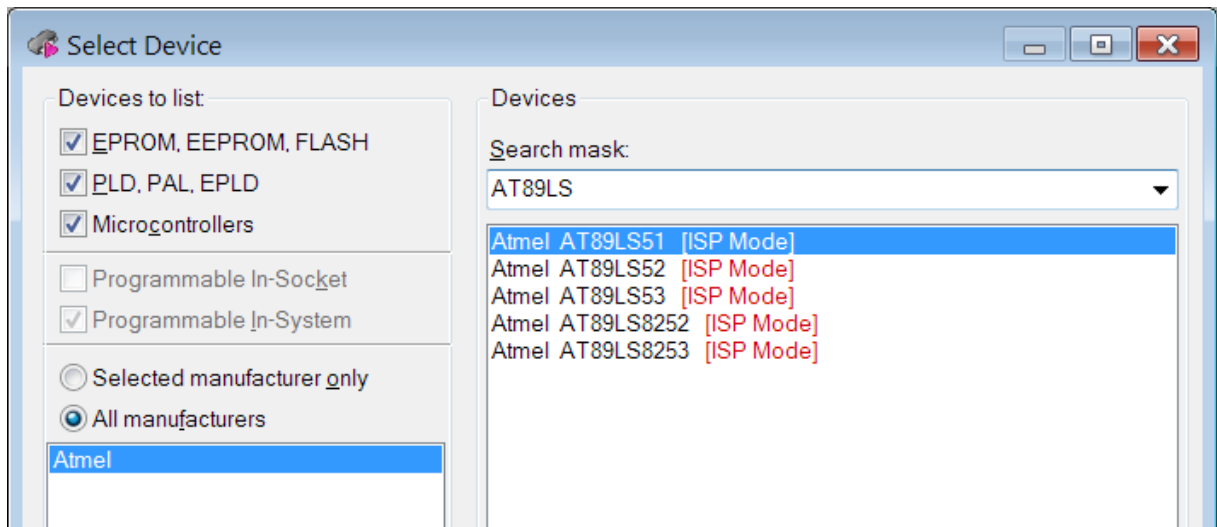
- Target device: Microchip/Atmel AT89LS51 [ISP Mode].
- File C:\Work\Monitors\RTX-028.hex (in standard hex format) has to be loaded into the ChipProg-02 memory buffer.
- A 32-bit serial number has to be written into each target device at address 0x200. Serial numbers are increased by 1 for each device.

Connect a **CPI2-B1** device programmer to a computer via a USB cable, launch the ChipProg-02 software and launch the programmer in the GUI mode.

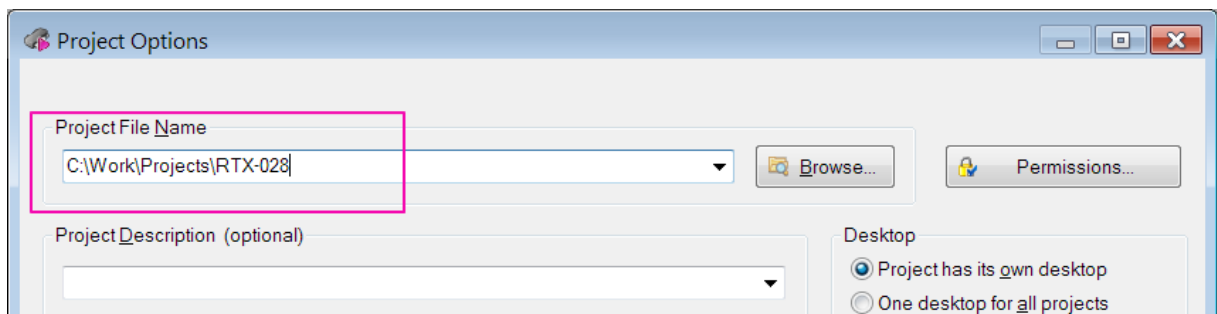
Click on "Select device" button:



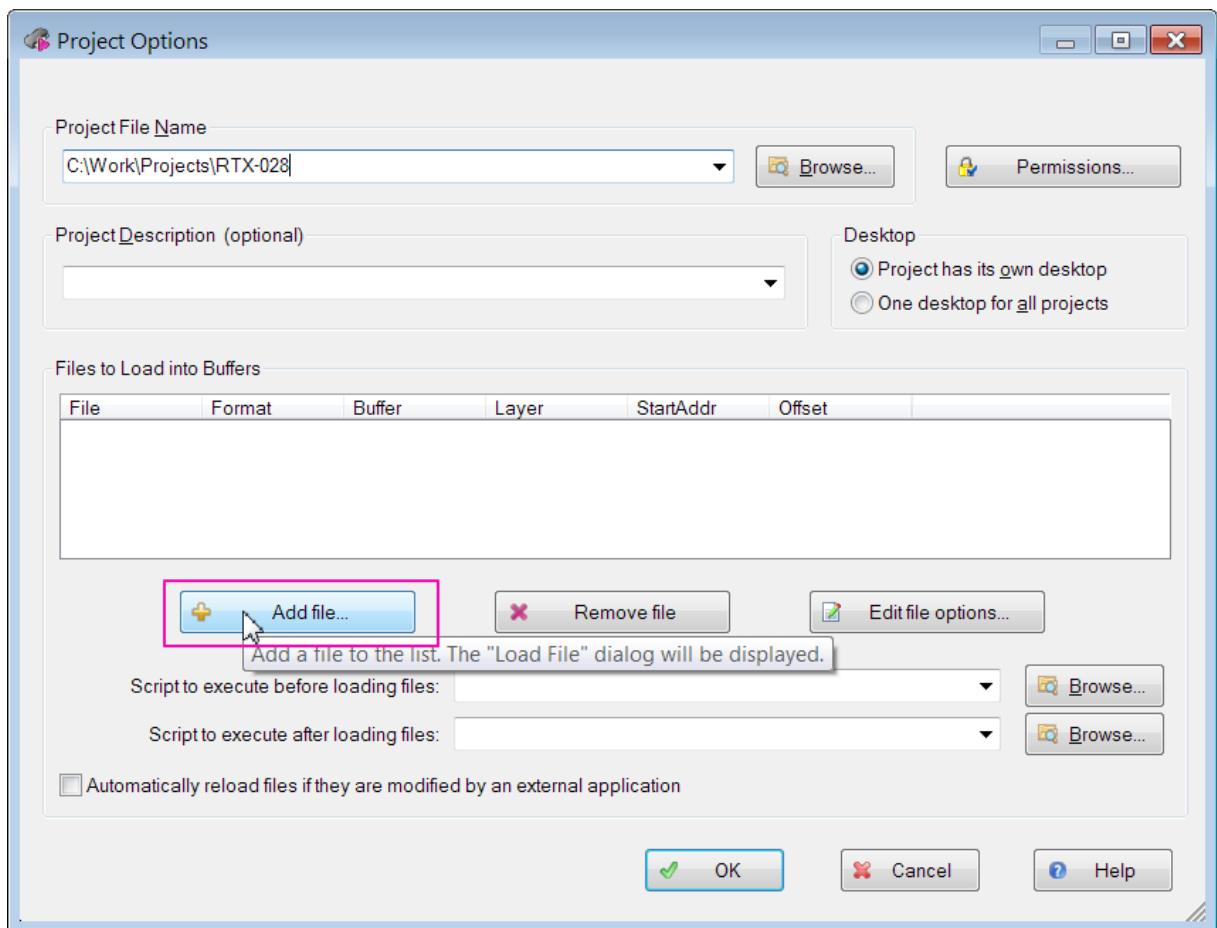
Select device type Atmel AT89LS51 [ISP Mode]:



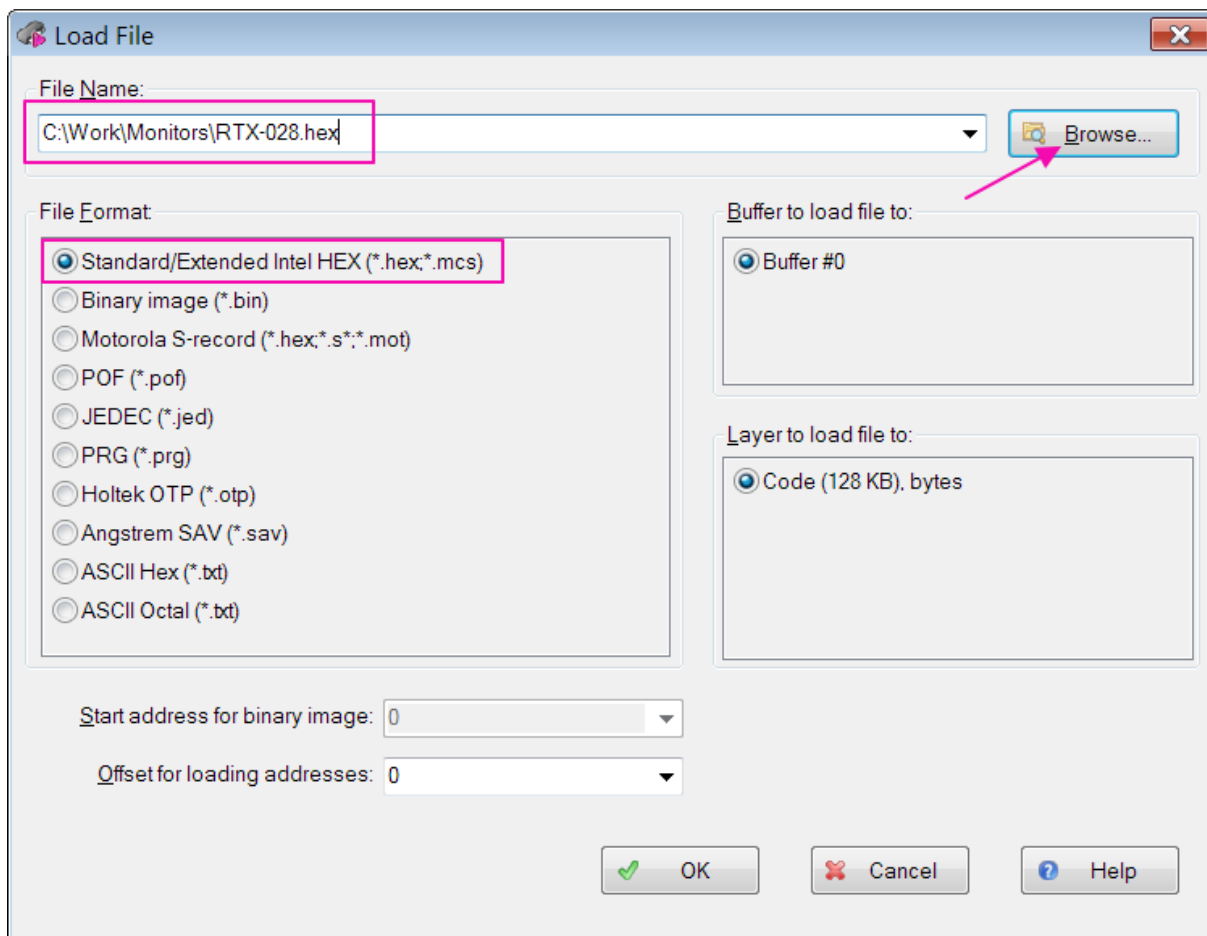
Open the menu **Project -> Create New**:



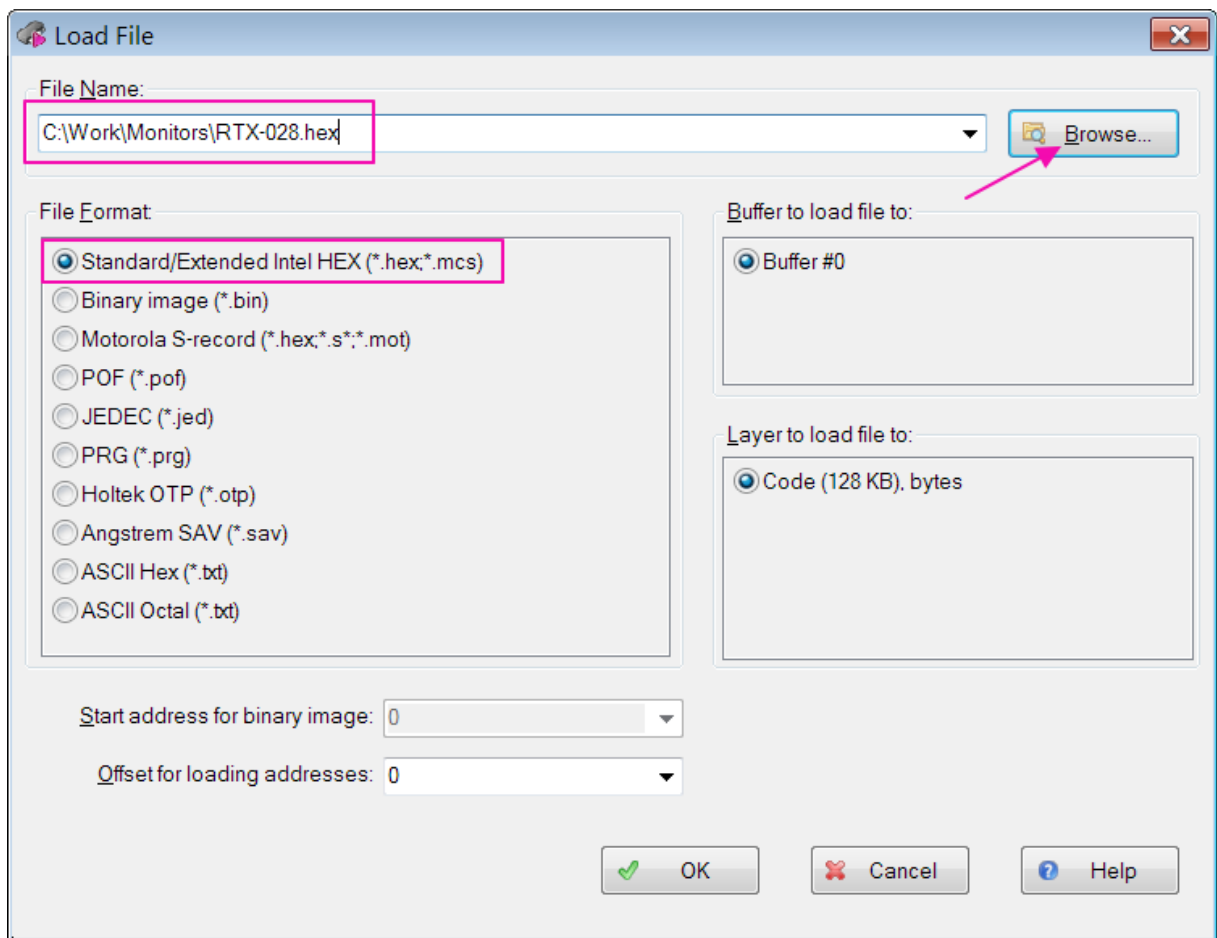
This brings up project creation dialog. In the field "**Project file name**" enter the name of the project file. Alternatively, click on **Browse** button and select folder and file using standard Windows dialog:



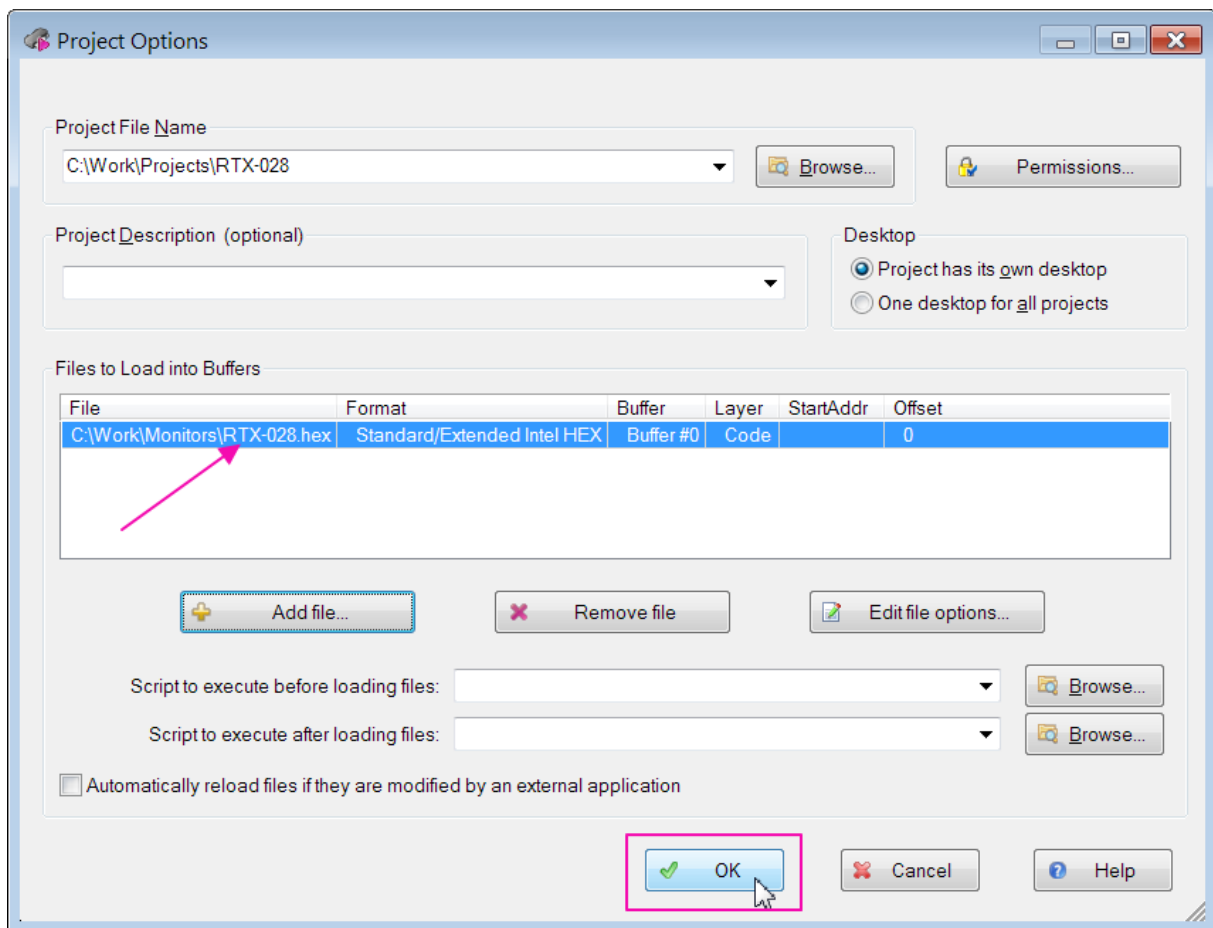
Select file C:\Work\Monitors\RTX-028.hex to be loaded:



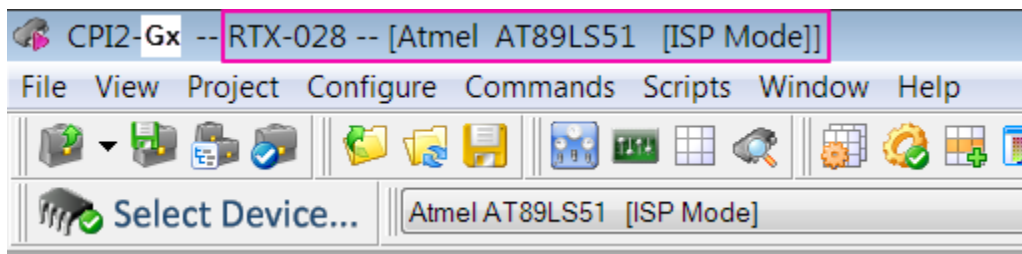
In file selection dialog enter C:\Work\Monitors\RTX-028.hex, or use **Browse** button. Select "Standard/Extended Intel HEX":



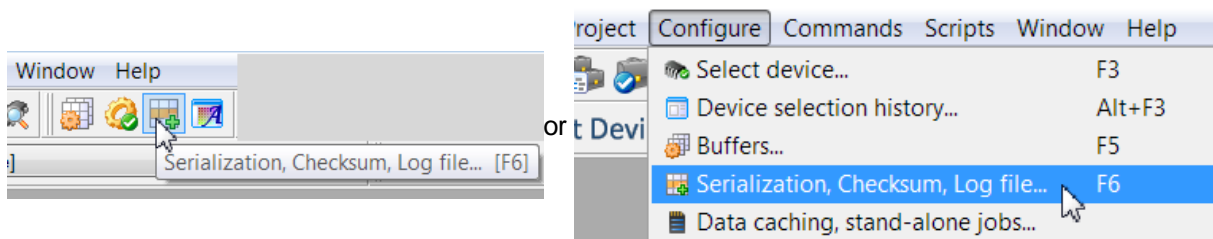
Confirm file selection by clicking OK, and the settings dialog will show the name of selected file. Confirm project settings by clicking OK; the project will be saved as the file **C:\Work\Projects\RTX-028.upp**. If the folder **:Work\Projects** does not exist, the program will prompt you to create it.



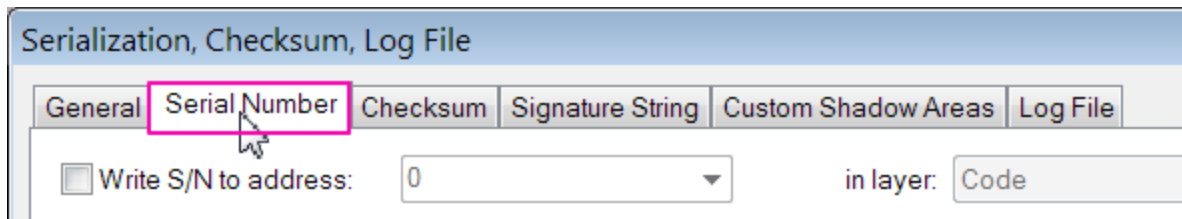
Now we are working with a project, as shown in the window title:



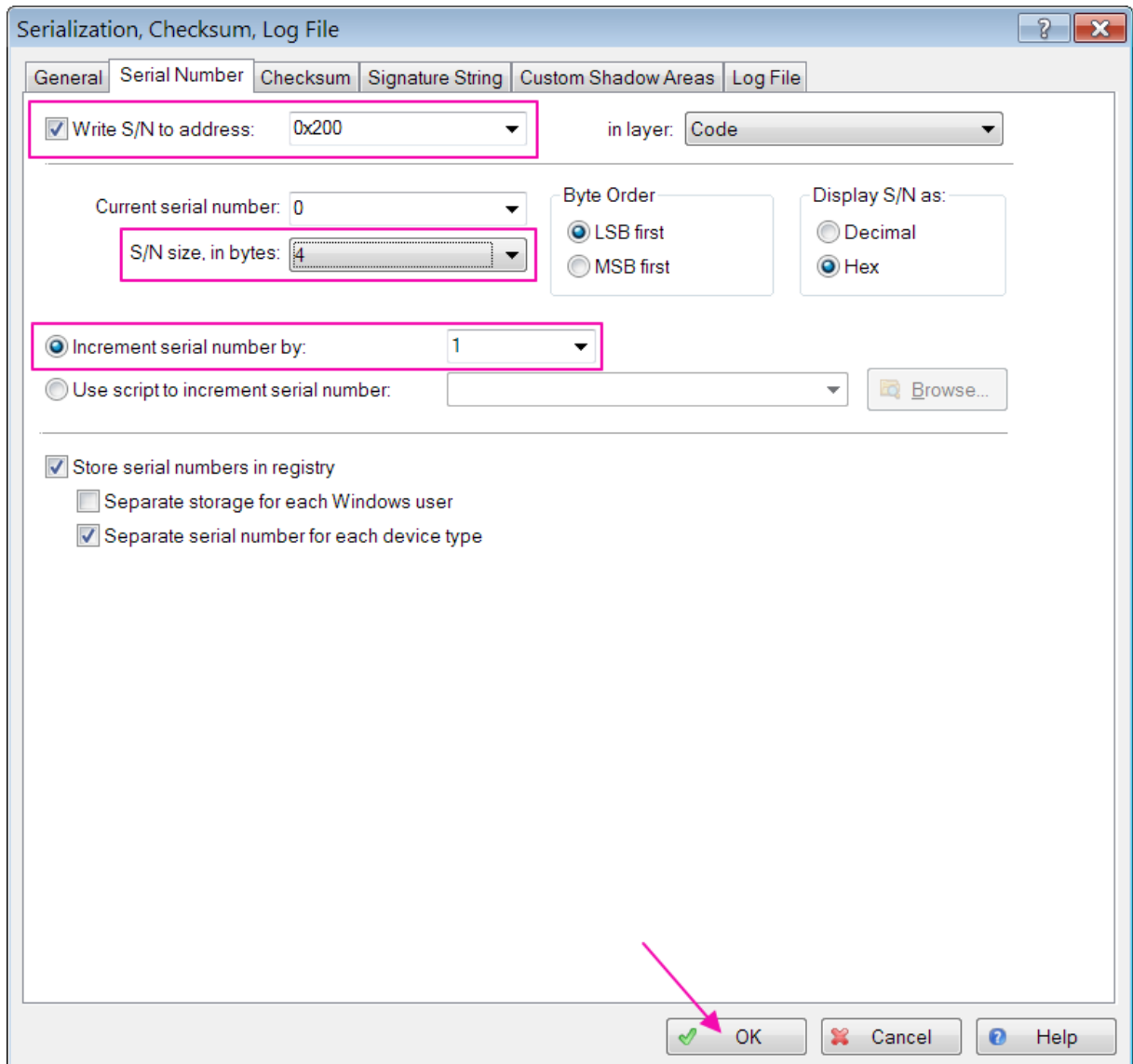
Now we need to set parameters of serial numbers written to each target device. To do this, open serialization settings:



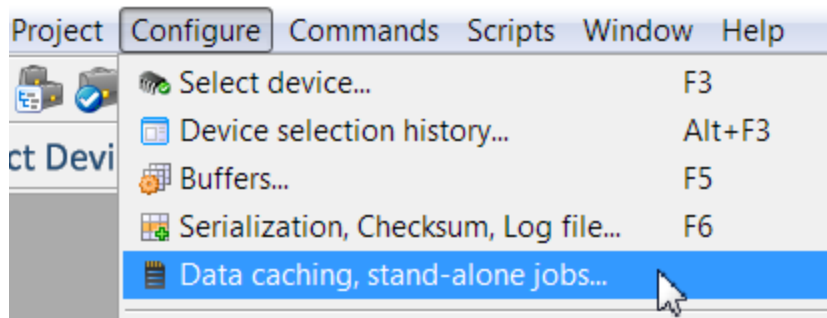
In the appeared dialog select the **"Serial Number"** tab:



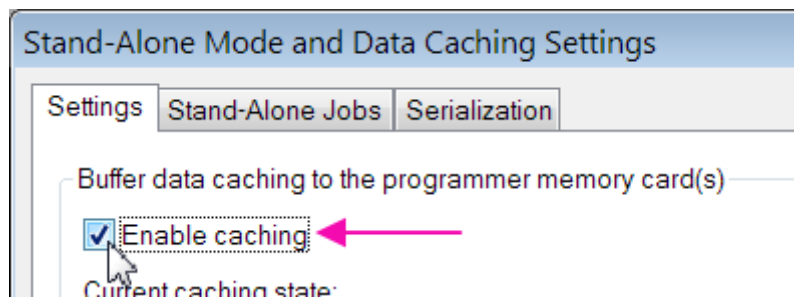
Check off the **"Write S/N address"** box and enter 0x200 into the address field. Set serial number size equal to 4 bytes, set increment to 1, then click **OK**:



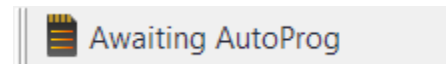
We are almost done setting project options but now we need to turn on the data caching; to do this, run menu command **Configure -> Data Caching...**:



This brings up a dialog for serialization parameters. Check off the **Enable Caching** box, then click **OK**:



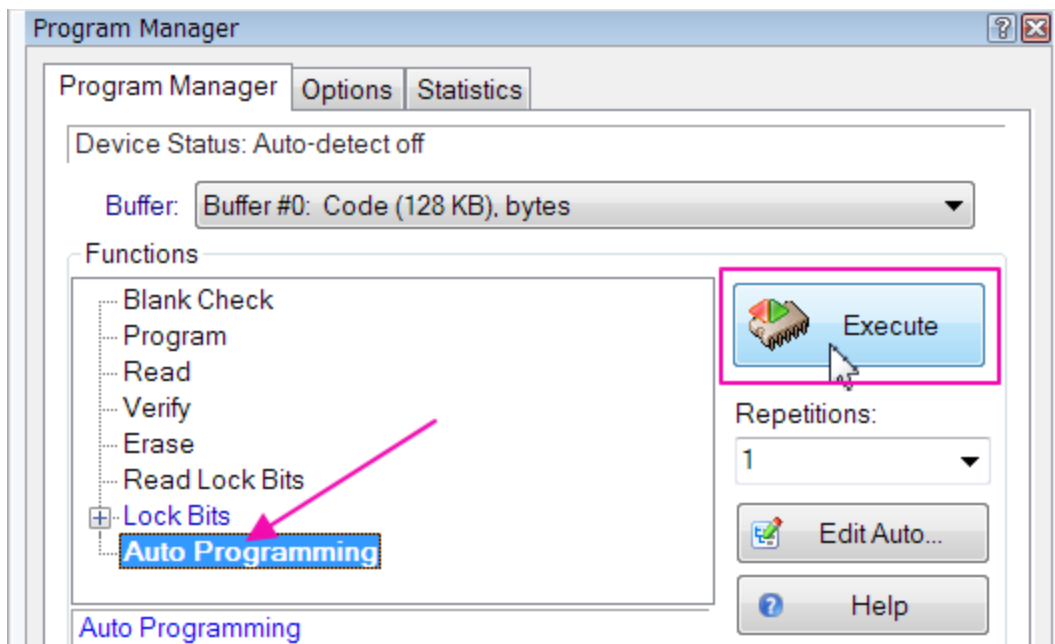
Data caching status now looks like this:



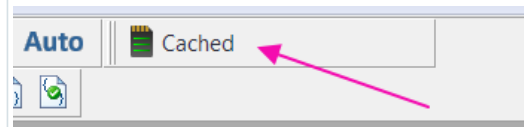
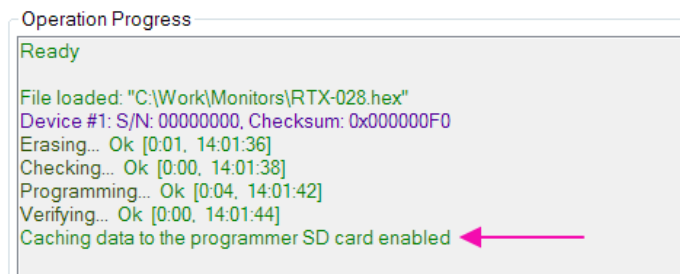
Since projects are not saved automatically in the ChipProg-02 application, you not must save the project by clicking the Save project icon on the main toolbar:



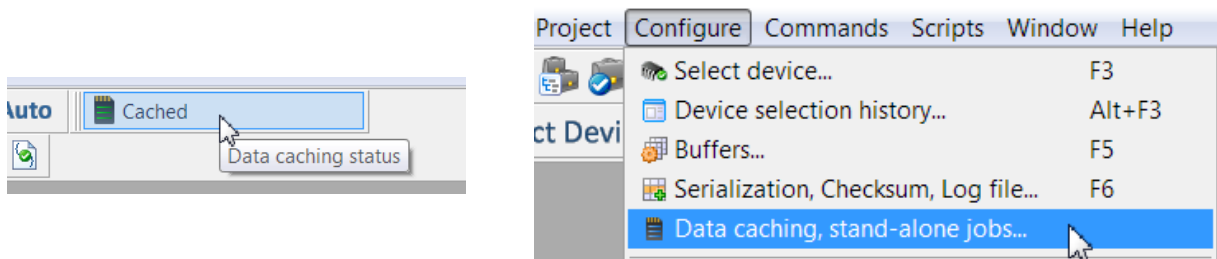
Preserving the connection diagram for the chosen AT89LS51 device connect it to the connector [TARGET²²](#) on your **CPI2-B1** device programmer and launch the **Auto Programming** command in the **Program Manager** window:



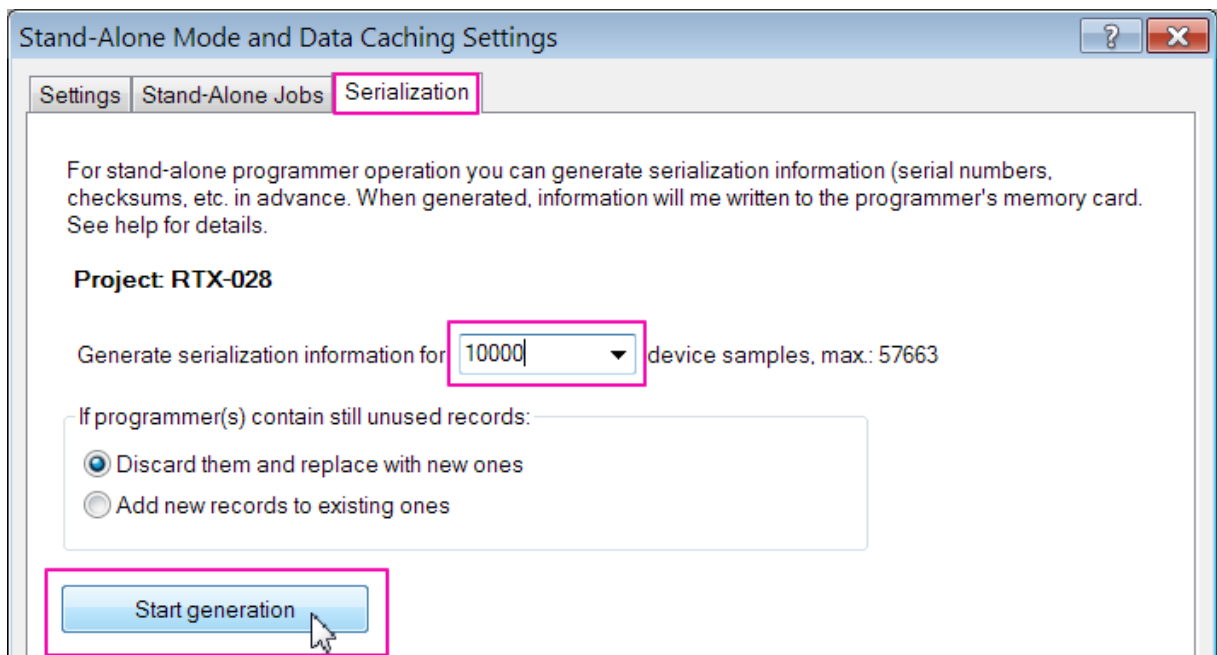
If the Auto Programming operation has completed successfully, the history field will display a line saying **"Caching data to the programmer SD card enabled"**; caching status will read **"Cached."**



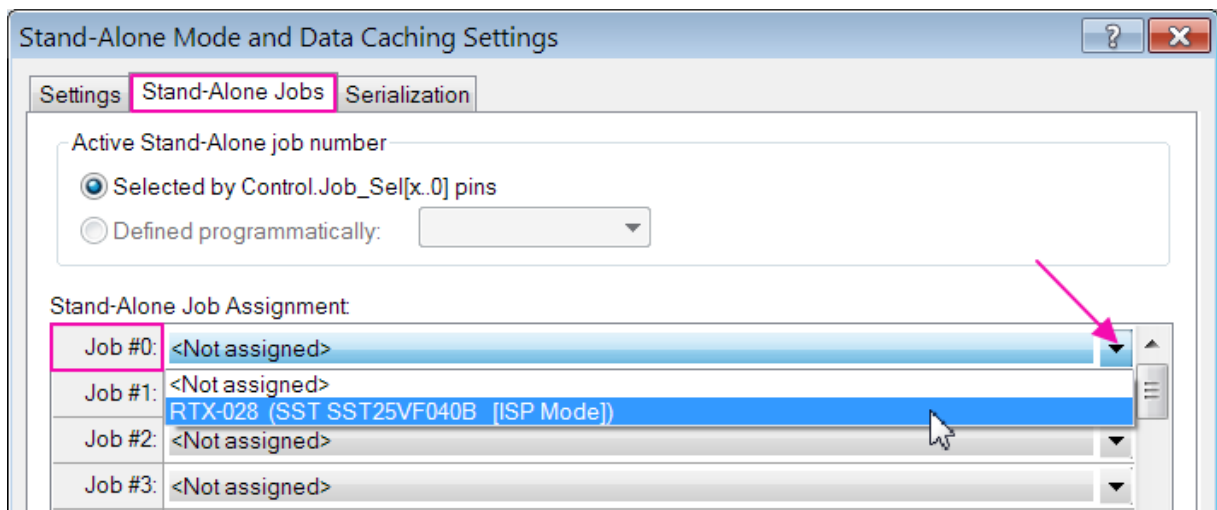
Now we need to generate the serial number information for writing serial numbers into target devices. To do this, either click on the caching status field or select menu **Configure -> Data Caching...**:



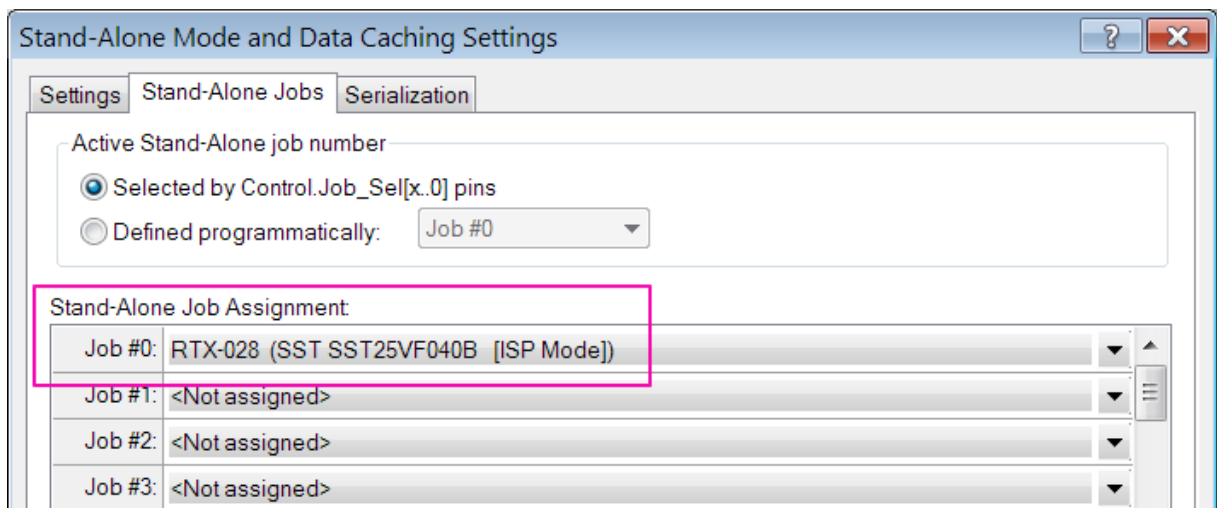
This brings up standalone mode options dialog. Select the **Serialization** tab and specify amount of 10000 devices to generate serial numbers for. Then click **Start Generation** button:



Assign our project to a standalone job #0 by selecting "**Standalone Jobs**" tab and selecting project RTX-08 for job #0:



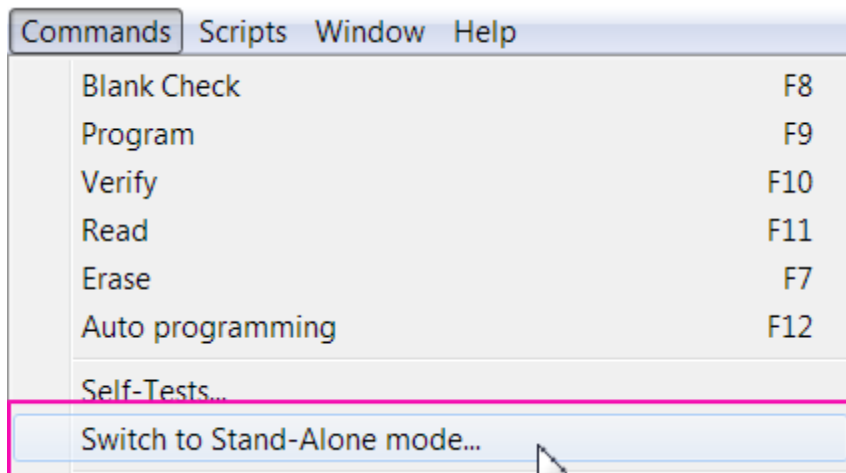
The dialog now looks like this:



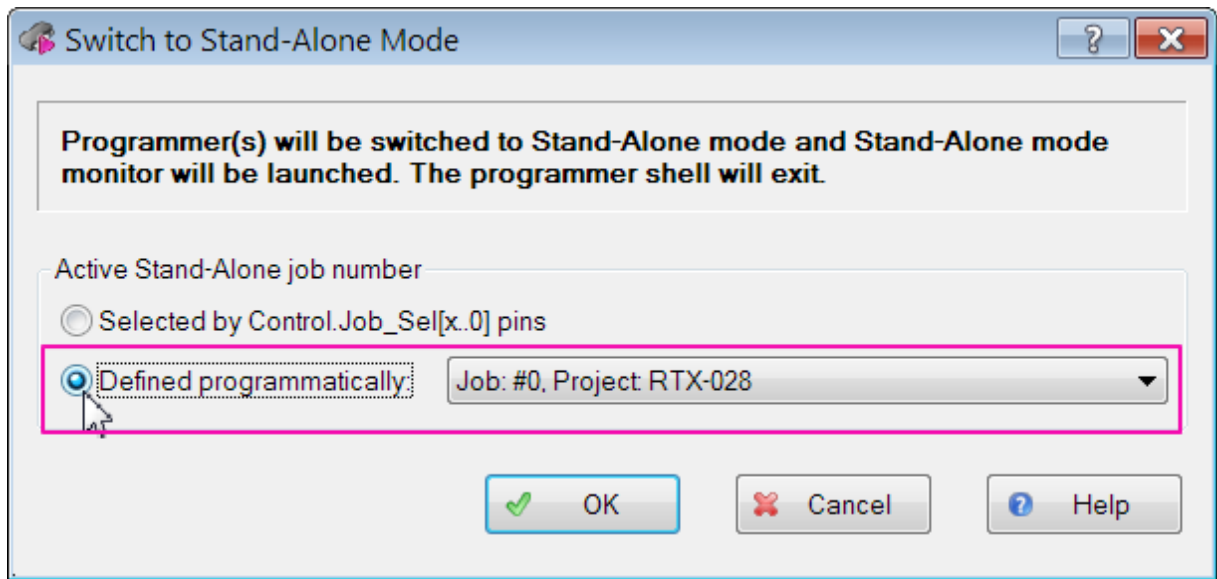
Confirm settings by clicking "OK" at the bottom.

This completes preparation of the standalone job associated with the project RTX-028. Contents of the 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.

The simplest way to switch the programmer into standalone mode is to call the menu command **Commands -> Switch to Standalone Mode:**



This brings up the dialog below allowing you to select the job #0 to be activated for execution from the GUI:



5 Software Development Kit (SDK)

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

5.1 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](#)^[160], 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.

5.2 Using ACI

To control a CPI2-B1 programmer, user program calls functions in the ACI.DLL. When user program calls the [ACI Launch\(\)](#)^[371] 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 dialogs; 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.

When launching a programmer by means of the [ACI Launch\(\)](#)^[371] function, ACI creates internal object called **connection** that identifies a launched programmer or multiple programmers working in the [Gang-programming](#)^[197] mode.

ChipProg-02 enables launching multiple CPI2-B1 device programmers and control each of them individually. The [ACI_SetConnection](#)^[378] function is used to select a particular **connection** to work with. Once a connection is selected, all further calls to ACI functions will be applicable to this **connection** use that connection (i.e. they all will affect only the selected device programmer). If there is only one programmer, the connection is selected automatically.

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**, but not six connections.

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 starts 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.

5.3 ACI Functions

This section provides an overview of Application Control Interface functions. Detailed description of each function can be found in the [ACI Functions](#) ³⁶⁵ 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 and control connections with device programmer(s)			
ACI_Launch ^[37]	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 ^[38]
ACI_Exit ^[37]	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 control session via ACI.	NA	NA
ACI_SetConnection ^[37]	Specifies a current device programmer(s) connection. Use this function when you control a number of device programmers by means of multiple calls of the ACI_Launch ^[37] function.	NA	ACI_Connection_Params ^[38]
ACI_GetConnection ^[36]	Allows getting the identifier of a current device programmer connection.	NA	ACI_Connection_Params ^[38]
ACI_ConnectionStatus ^[36]	Checks and returns a current connection status.	NA	NA
2. ACI functions that configure the programmer or get its current configuration			
ACI_LoadConfigFile ^[37]	Loads the programmer configuration parameters from the host computer to the programmer.	NA	ACI_Config_Params ^[38]
ACI_SaveConfigFile ^[37]	Saves the programmer's current configuration parameters to the host computer.	NA	ACI_Config_Params ^[38]
3. ACI functions that get the target device properties or set them			
ACI_GetDevice ^[36]	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 ^[58]	ACI_Device_Params ^[38]
ACI_SetDevice ^[37]	Sets the manufacturer's name and the part number of the device to be programmed in the programmer.	Select Device ^[58]	ACI_Device_Params ^[38]
4. ACI functions that get current parameters of the buffers and layers or configure them			

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
ACI_GetLayer ^[370]	Gets the parameters of a specified memory buffer and layer from the programmer to the host computer.	Buffer Dump ^[95]	ACI_Layer_Params ^[388]
ACI_CreateBuffer ^[367]	Creates a memory buffer with specified parameters in the programmer.	Buffer Dump ^[95]	ACI_Buffer_Params ^[378]
ACI_ReallocBuffer ^[374]	Changes a size of the layer #0 in a specified memory buffer in the programmer.	Buffer Dump ^[95]	ACI_Buffer_Params ^[378]
5. ACI functions that read the content of the buffer layer or write into it			
ACI_ReadLayer ^[374]	Reads data from a specified memory buffer in the programmer to the host computer.	Buffer Dump ^[95]	ACI_Memory_Params ^[388]
ACI_WriteLayer ^[378]	Writes data into a specified memory buffer of the host computer to the programmer memory buffer.	Buffer Dump ^[95]	ACI_Memory_Params ^[388]
ACI_FillLayer ^[368]	Fills a whole selected layer of a specified memory buffer with a specified data pattern.	Buffer Dump ^[95]	ACI_Memory_Params ^[388]
6. ACI functions that get programming parameters from the programmer or set them in the programmer			
ACI_GetProgrammingParams ^[371]	Gets current programming parameters from the programmer to the host computer.	Program Manager > Options ^[109]	ACI_Programming_Params ^[393]
ACI_SetProgrammingParams ^[371]	Sets programming parameters from the host computer to the programmer.	Program Manager > Options ^[109]	ACI_Programming_Params ^[393]
7. ACI functions that get device-specific programming options from the programmer or set them in the programmer			
ACI_GetProgOption ^[370]	Gets current programming options from the programmer to the host computer.	Device and Algorithm Parameters ^[93]	ACI_ProgOption_Params ^[389]
ACI_SetProgOption ^[370]	Sets programming options from the host computer to the programmer.	Device and Algorithm Parameters ^[93]	ACI_ProgOption_Params ^[389]
ACI_AllProgOptionsDefault ^[369]	Sets default programming options and programming algorithms in the programmer.	Device and Algorithm Parameters ^[93]	ACI_ProgOption_Params ^[389]
8. ACI functions that control programming operations			
ACI_ExecFunction ^[367]	Initiates a specified programming operation, keeping under control its successful completion or failure. It controls a single programmer.	Program Manager ^[107]	ACI_Function_Params ^[389]

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
ACI_StartFunction ^[378]	Initiates a specified programming operation and then does not check the operation result. It controls a single programmer.	Program Manager ^[107]	ACI_Function_Params ^[383]
ACI_GangStart ^[369]	Used to control multiple device programmers. Initiates auto programming in the gang (gang-programming ^[197]) mode.	Program Manager ^[107]	ACI_GangStart_Params ^[384]
ACI_GetStatus ^[371]	Gets a current programmer status information.	Program Manager ^[107]	ACI_PStatus_Params ^[395]
ACI_SerializationDialog ^[376]	This macro sends a command that opens Serialization dialog.	Serialization, Checksum, and Log Dialog ^[63]	NA
ACI_TerminateFunction ^[378]	Terminates a current programming operation.	Program Manager ^[107]	NA
ACI_GangTerminateFunction ^[369]	Terminates a current programming operation on a specified site of the gang programmer.	Program Manager ^[107]	ACI_GangTerminate_Params ^[385]
ACI_ErrorString ^[367]	Get the string describing the result of the last ACI function call	Program Manager ^[107]	NA
9. ACI functions that save files from the programmer and load projects or files to the programmer			
ACI_LoadProject ^[373]	Loads a specified project that must be previously prepared and saved manually in the programmer GUI.	Select Device ^[58] , Buffer Dump ^[95] , Device and Algorithm Parameters ^[93]	ACI_Project_Params ^[399]
ACI_FileSave ^[368]	Saves a specified file from a specified buffer's layer of the programmer into the instrumental computer.	Buffer Dump ^[95]	ACI_File_Params ^[381]
ACI_FileLoad ^[368]	Loads a specified file from the instrumental computer to a specified buffer's layer in the programmer.	Buffer Dump ^[95]	ACI_File_Params ^[381]
10. ACI functions that display programmer's windows and dialogs for setting up and debugging external programming sessions			
ACI_SettingsDialog ^[377]	Displays the programmer Preferences dialog.	Configure > Preferences ^[78]	NA
ACI_SelectDeviceDialog ^[375]	Displays the Select Device dialog.	Select Device ^[58]	NA
ACI_BuffersDialog ^[365]	Displays the memory buffers setting dialog.	Buffer Dump ^[95]	NA
ACI_LoadFileDialog ^[372]	Displays the file loading dialog.	Buffer Dump ^[95]	NA

Application Control Interface function name	Brief description	Associated windows and dialogs	Associated Application Control Interface structures
ACI_SaveFileDialog ^[374]	Displays the file saving dialog.	Buffer Dump ^[95]	NA

5.4 ACI Structures

This section provides an overview of the structures used in calls to [ACI functions](#)^[160]. Detailed description of each structure can be found in the [ACI Structures](#)^[378] reference section.

Structure	The ACI function that uses the structure
ACI_Launch_Params ^[385]	ACI_Launch ^[371]
ACI_Config_Params ^[380]	ACI_LoadConfigFile ^[371] , ACI_SaveConfigFile ^[374]
ACI_Device_Params ^[381]	ACI_GetDevice ^[369] , ACI_SetDevice ^[376] ,
ACI_Layer_Params ^[386]	ACI_GetLayer ^[370]
ACI_Buffer_Params ^[378]	ACI_CreateBuffer ^[367] , ACI_ReallocBuffer ^[374]
ACI_Memory_Params ^[388]	ACI_ReadLayer ^[374] , ACI_WriteLayer ^[378] , ACI_FillLayer ^[368]
ACI_Programming_Params ^[393]	ACI_SetProgrammingParams ^[377] , ACI_GetProgrammingParams ^[371]
ACI_ProgOption_Params ^[389]	ACI_GetProgOption ^[370] , ACI_SetProgOption ^[376]
ACI_Function_Params ^[383]	ACI_ExecFunction ^[367] , ACI_StartFunction ^[378]
ACI_PStatus_Params ^[395]	ACI_GetStatus ^[371]
ACI_File_Params ^[381]	ACI_FileLoad ^[368] , ACI_FileSave ^[368]
ACI_GangStart_Params ^[384]	ACI_GangStart ^[369] , ACI_GetStatus ^[371]
ACI_GangTerminate_Params ^[385]	ACI_GangTerminateFunction ^[369]

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.

5.5 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\(\)](#)^[37] 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\(&Status\)](#)^[371] 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\(&Status\)](#)^[371] function. After detecting correct and reliable placement, the program calculates the real size of the SST89V564RD device flash memory by executing the [ACI_ExecFunction](#)^[367] 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](#)

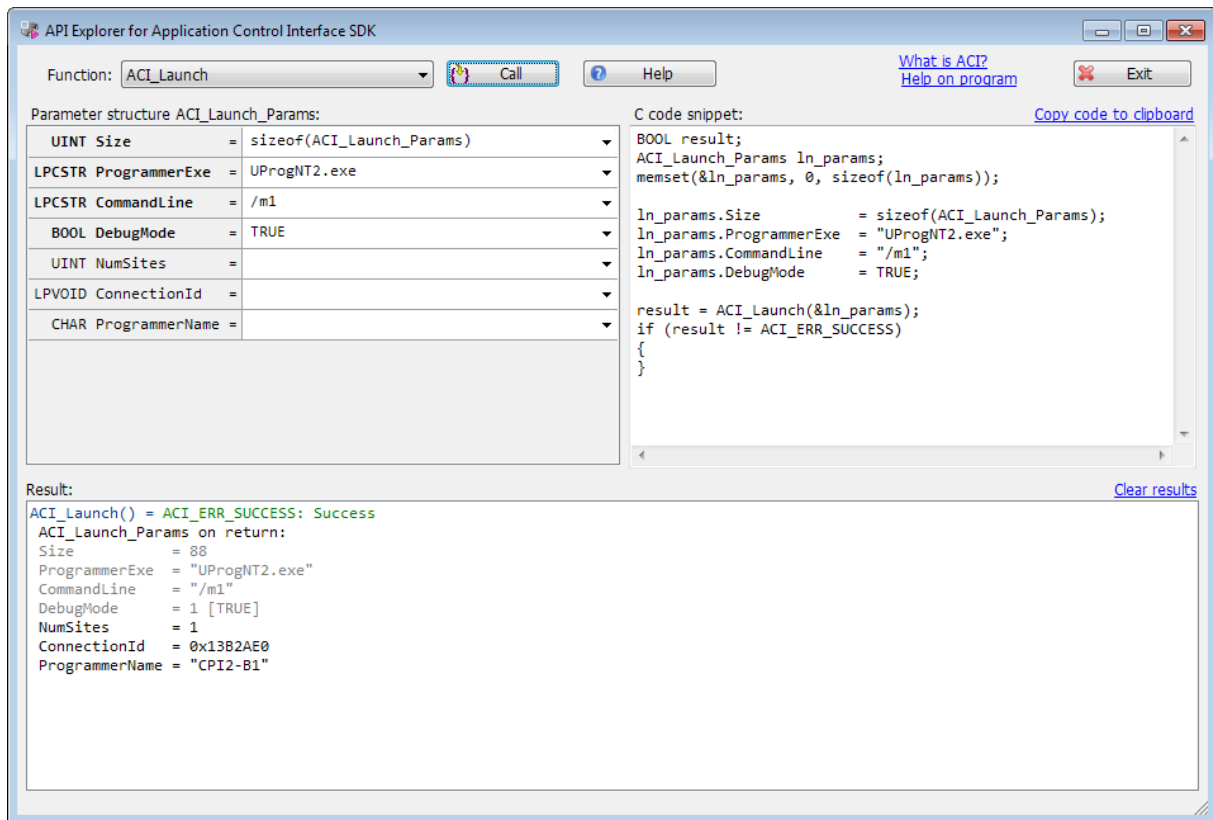
This example shows how to monitor the AutoProgramming procedure that takes a long time. Programming is launched by calling the [ACI_StartFunction](#)^[378]. Completion percentage of the operation is then checked by calling the [ACI_GetStatus](#)^[371] function. If the operation fails, the programmer 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](#)^[93] window. First, the program checks device insertion in the programmer's socket by calling the [ACI_GetStatus\(\)](#)^[371] function. After detecting correct and reliable insertion of the device, the program reads current set of options by calling the [ACI_GetProgOption\(\)](#)^[370] 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.

5.6 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; in 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 0xFFFF0.

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](#)^[374] 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 of 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.

6 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](#)^[120] table;
- Application Control Interface ([ACI](#)^[158]).

Each method is described in an appropriate section below.

The ChipProg-02 software includes a few [examples](#)^[173] of the **Virtual Instrument (.VI)** files.

6.1 LabVIEW Integration Using Command Line

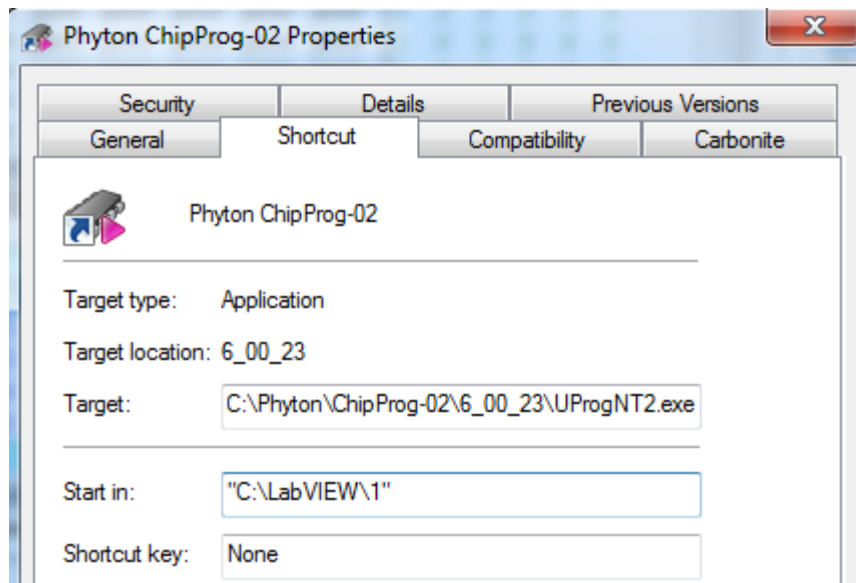
This is the most simple way to integrate ChipProg-02 with LabVIEW that involves two steps.

- 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:\LabView1**.

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\6_00_23\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:\LabView1** 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:\LabView1** 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](#)^[107] 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](#)^[109] tab; the number of chips to be programmed during a programming session and other options in the [Statistics](#)^[111] tab.

- Settings in the [Device and Algorithm Parameters Editor](#)^[93] 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...](#)^[63] 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](#)^[78] and [Environment](#)^[79] menus, such as color, fonts, sounds, etc.

Complete the definition of programming session by including appropriate [Command line options](#)^[120].

- 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](#)^[108] mode (key **/A**);
- Launching programmer in hidden mode, when the ChipProg-02 GUI is hidden (key **/I2**).

Notes:

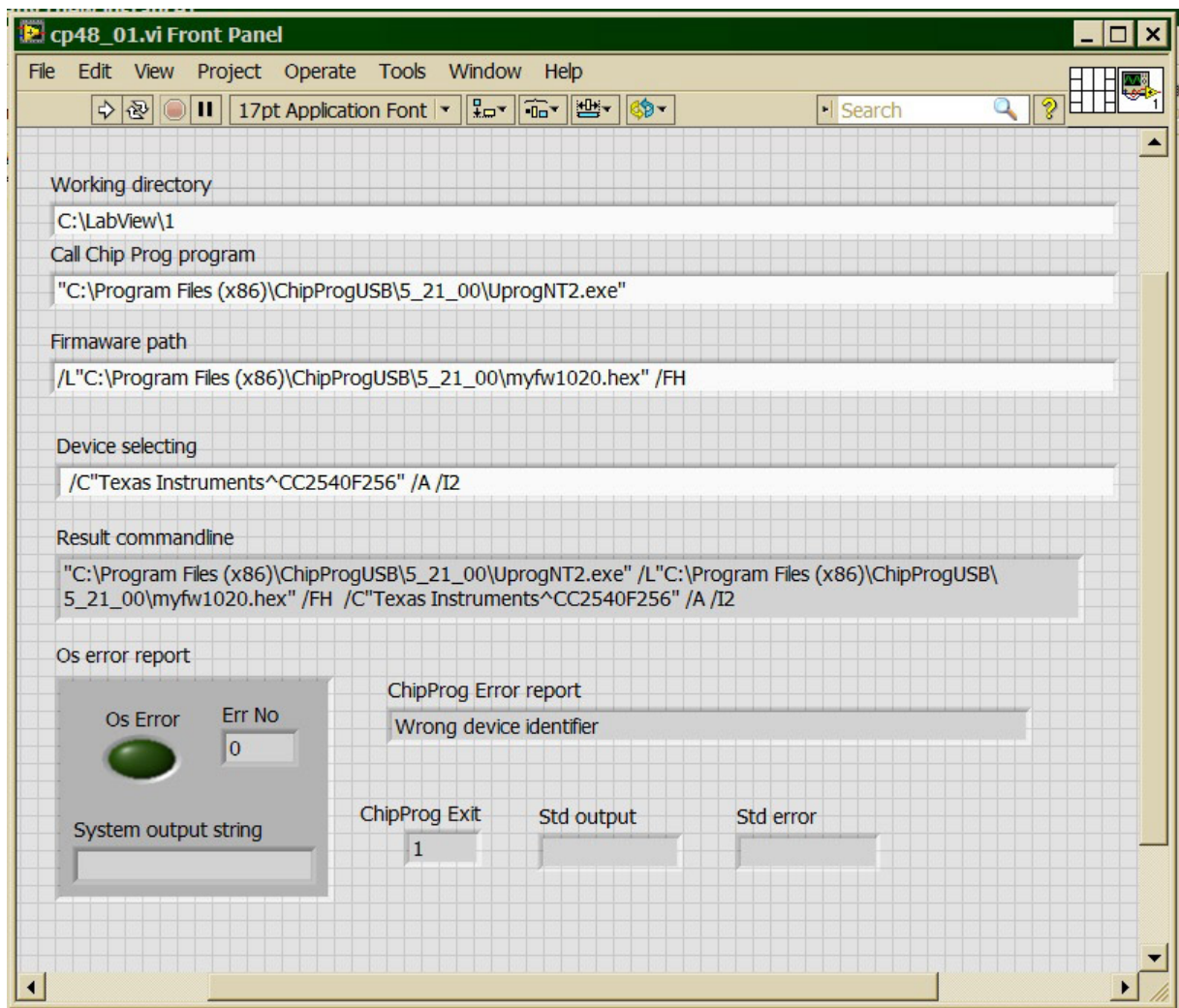
- Device specified by the **/C** key on command line must be the same as chosen in the ChipProg-02 user interface.
- Specifying **/I2** key on command line hides ChipProg-02 application main window, suppresses display of error messages but copies them to the Windows clipboard. If the session terminates 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:

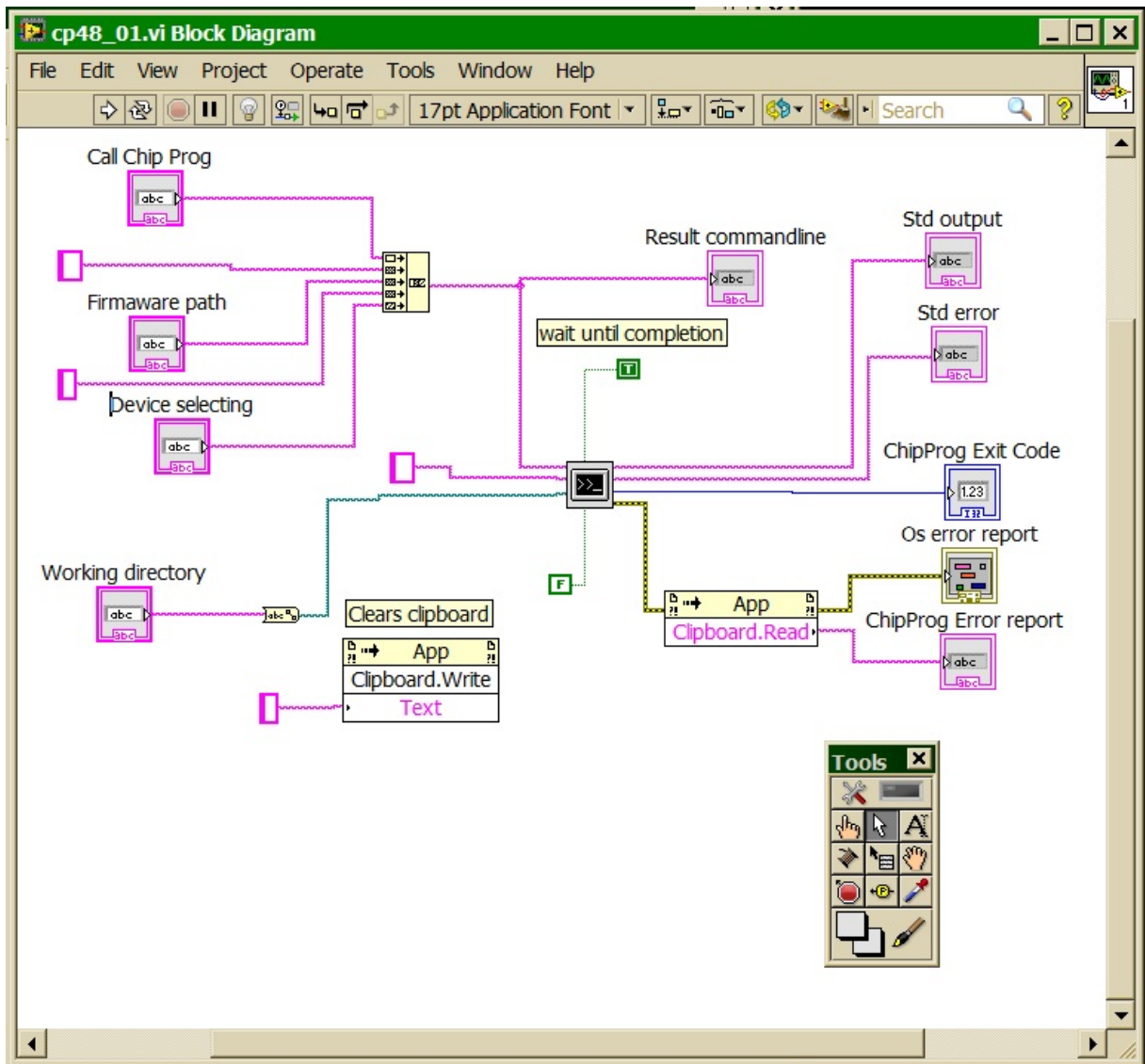
```
"C:\Program Files (x86)\ChipProg-02\6_00_21\UprogNT2.exe" /L"Program Files (x86)\ChipProg-02\6_00_21\myfw1020.hex" /FH /C"NXP^MK20N64VFT7 [ISP EzPort Mode]" /A/I2
```

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.

6.2 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](#) ^[173] 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.

See the VI file examples [here](#).

6.2.1 LabVIEW Integration Examples

The ChipProg-02 software includes a few examples of the Virtual Instrument files (VI files) that illustrate control of the CPI2-B1 programmers by the NI LabVIEW software. These examples are located in the folders:

- For the 32-bit LabVIEW version - C:\Phyton\ChipProg-02\x_xx_xx\LabVIEW\x86\Examples\
- For the 64-bit LabVIEW version - C:\Phyton\ChipProg-02\x_xx_xx\LabVIEW\x64\Examples\

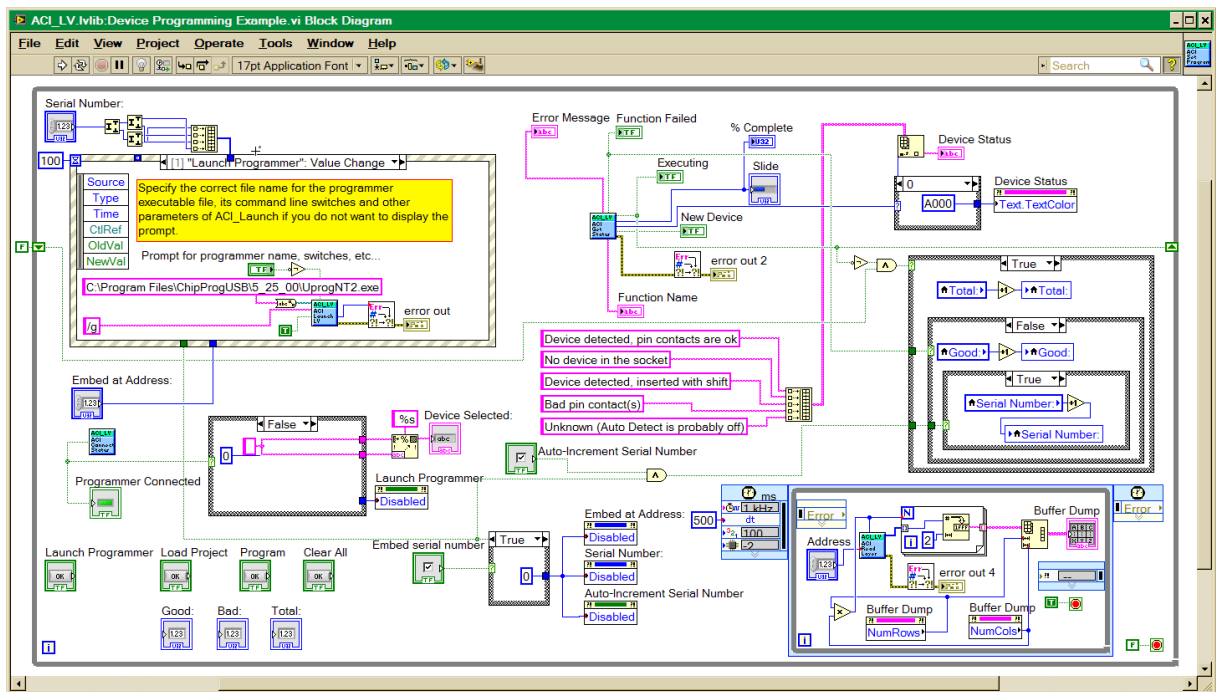
Currently, these folders contain three Virtual Instrument examples below but Phyton may add new examples further:

- **Device Programming Example.vi**
- [Programming Params Control Example.vi](#)
- [Gang_serial.vi](#)

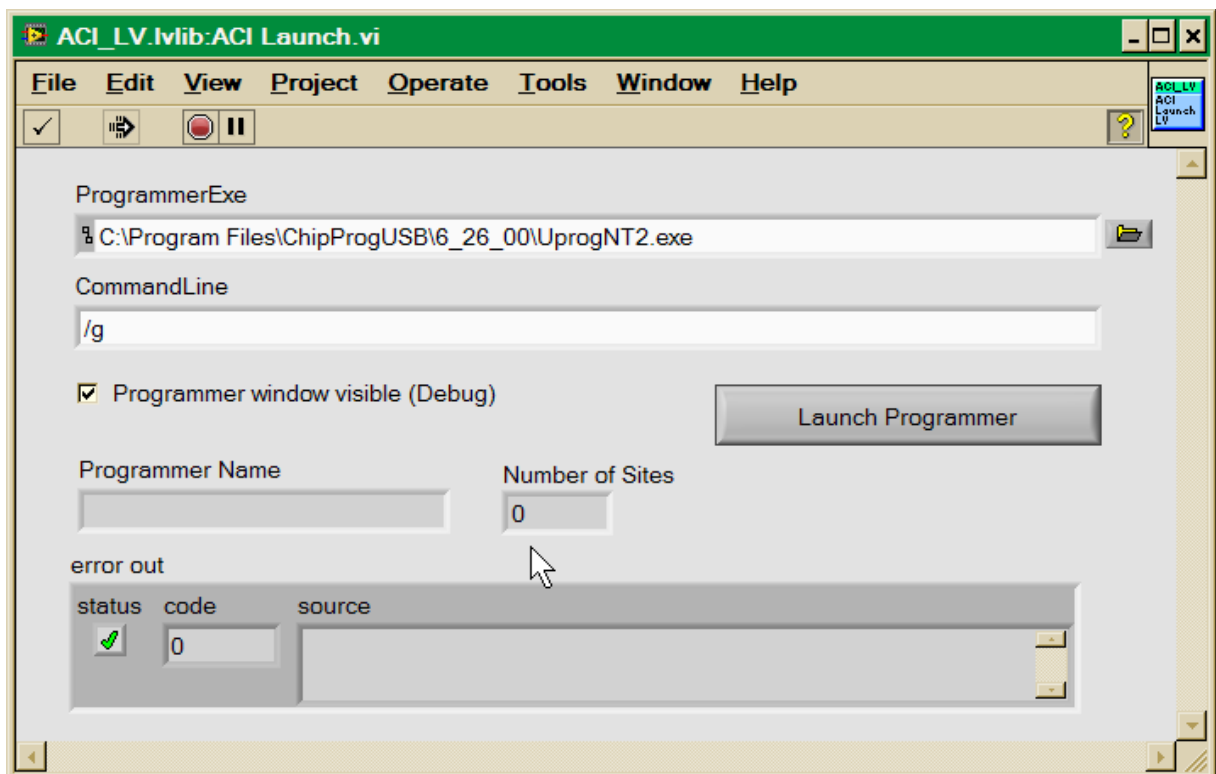
The **Device Programming Example.vi** demonstrates use of all major ACI functions, namely:

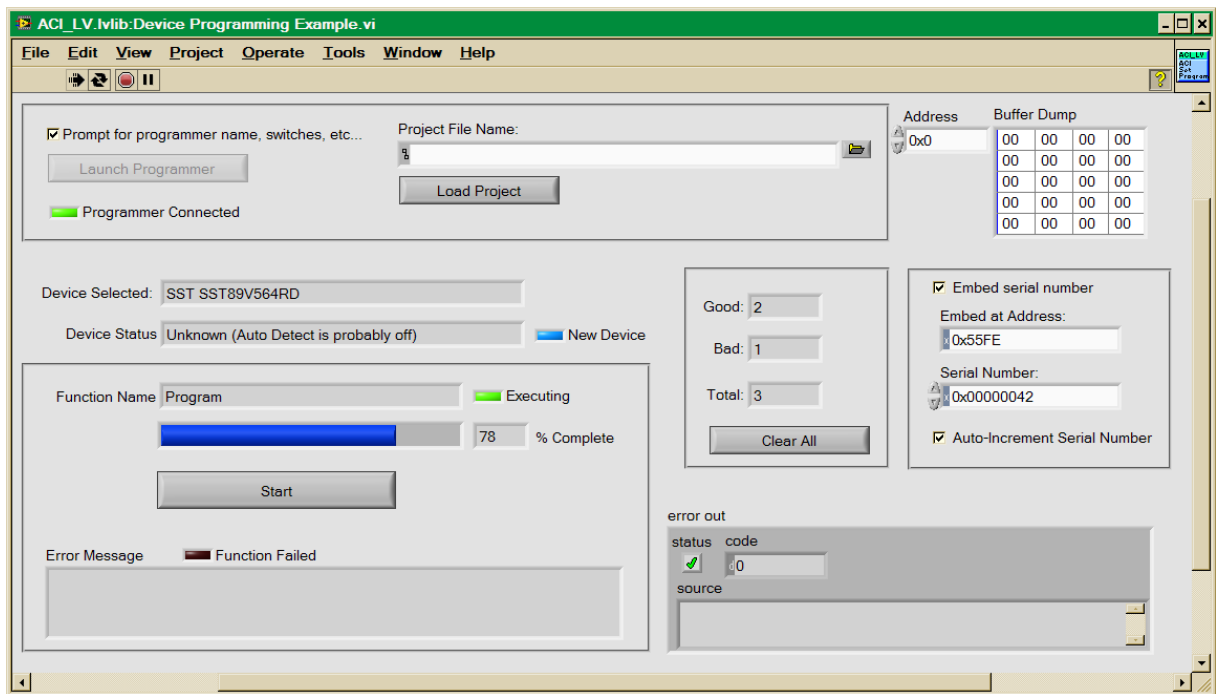
- launch a device programmer;
- load a project;
- display the device programmer buffer content in the GUI;
- display a chosen device in the GUI;
- display the device programmer socket's status (if a chosen programmer type supports this feature);
- write a serial number and increment it automatically 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 uncheck 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](#)¹⁰⁸ function. This process is illustrated in figures below.





The **Gang_serial.vi** example is a modification of the **Device Programming Example** described above and illustrates of how to operate with multiple CPI2-B1 device programmers running in the [gang programming](#)^[197] mode (or gang mode). The structure of the **Gang_serial** example is identical to the structure of the **Device Programming Example**.

To add a new site:

1. Copy all the contents of the last "case" structure containing the "Get status" function to a free space. Create a new "case" inside the structure and paste the copied data into it. Attach the data outputs in the same manner as in the previous "case" (it is necessary to copy and paste instead of duplicating the "case" entirely because copy/paste creates new variables required for the program to work. Duplicating the "case" will use old variables).
2. Create local variables from the newly created 'Total', 'good', 'bad' (Right Mouse Button (hereafter RMB) -> Create -> Local variable. Add them to the event structure of the "Clear all" event. Connect these variables just like others.
3. Create a local variable from the new 'Executing' element in the 'Read' mode. Include it into the iteration block via the logical 'OR' just like the other 'Executing' variables are.
4. Copy and paste the last (by number) 'Program Site' button. Duplicate the last (by number) "case" of the "Program site value change" in the event structure. As a condition set a change of the newly created button value.

When adding new [ACI Functions](#)^[365] make sure to set the correct site number in the appropriate variables. (Steps 1, 4 in particular).

The last thing to do is to arrange all new indicators on the front panel. Adding "site online" light is optional.

7 Scripting

7.1 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](#)^[180] and the [I/O Stream window](#)^[180]. Scripts can also create custom menus.

Scripts can send messages to [Console window](#)^[104] or to [User](#)^[180] 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](#)^[205]. The scripting subsystem supports many built-in functions, such as printf(), sin() and strcpy().

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

The **scripts** controls and associated dialogs and windows are concentrated under the [Script menu](#)^[88]. The major dialog that controls scripts is the [Script Files dialog](#)^[178].

How to write a script file

Script is similar to a in C language program. You can use the ChipProg-02 [built-in editor](#)^[184] 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](#)^[178] [dialog](#)^[178].

The Reference section contains detailed information about scripting.

7.1.1 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
}
```

7.2 The Startup Script

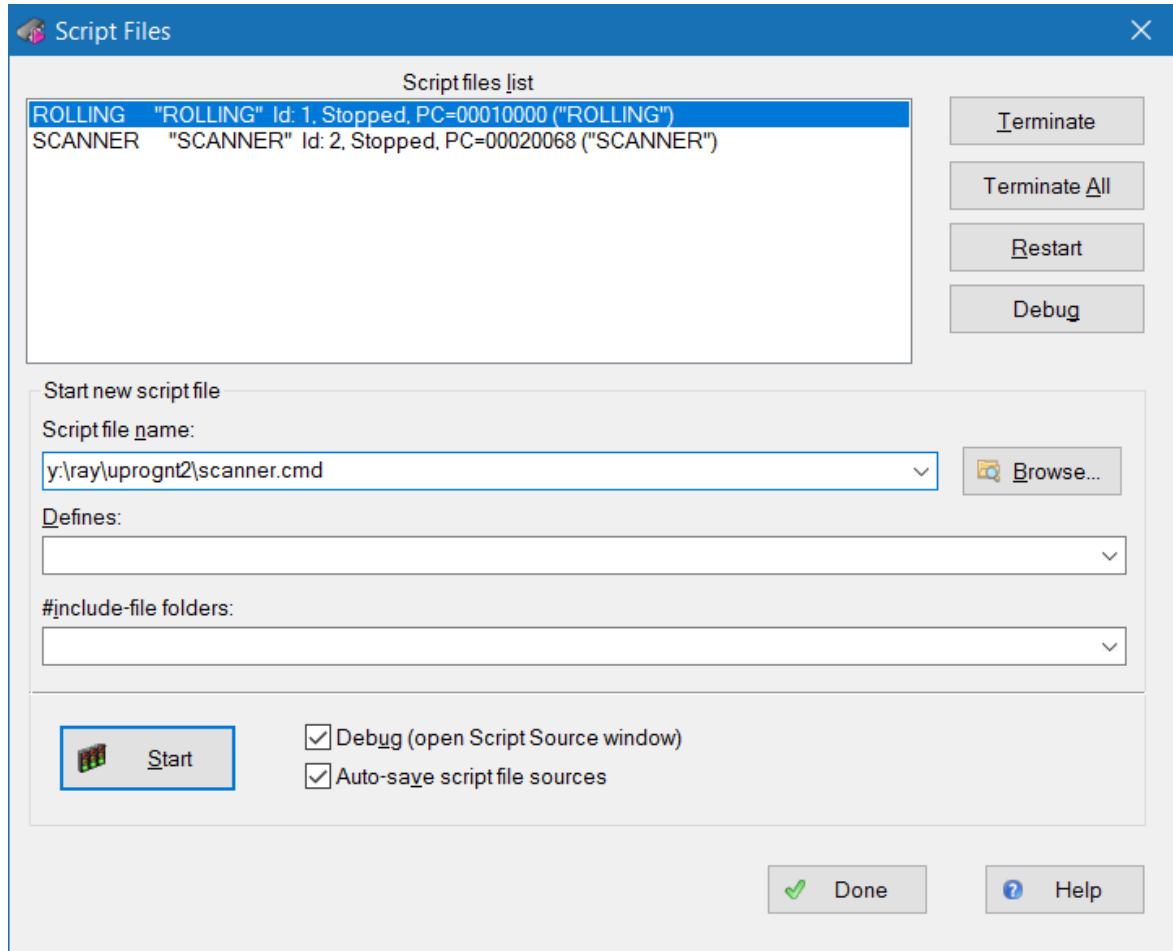
When the ChipProg-02 application starts, it automatically runs the **start.CMD** script if it exists. This is similar to execution of the `autoexec.bat` file in Windows. ChipProg-02 first looks for **start.CMD** file in the current directory; if it is not found, ChipProg-02 then looks for **start.CMD**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

7.3 Running Scripts

Scripts can be started and restarted in several ways. The easiest one uses the commands of the [Script Files](#) ^[178] dialog. A script can also be started by calling the **StartCommandFile()** function from another script.

7.3.1 The Script Files Dialog

This dialog is used to 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:

<u>State of Script</u>	<u>Description</u>
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:

<u>Button</u>	<u>Description</u>
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 ^[18] 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:

<u>Dialog Control</u>	<u>Description</u>
Script File Name	Specifies the filename of the script for loading. You may type in file name with full path, or 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 ^[18] .
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](#)^[23].

7.3.2 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.

7.3.3 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, then a line that 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.

7.4 Debugging a Script

A script can be started in **Debug mode**. This is usually necessary while you test the script to see if it works properly, and make necessary corrections. To start a script in debug mode, highlight its name in the [Script Files dialog](#)^[178] and click the **Debug** button. This brings up the [Script Window](#)^[181].

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](#)^[181] and **Watches** windows. If the **Debug** option is set in the [Script Files](#)^[178][dialog](#)^[178], the [Script Source](#)^[181] window opens automatically when starting the script.

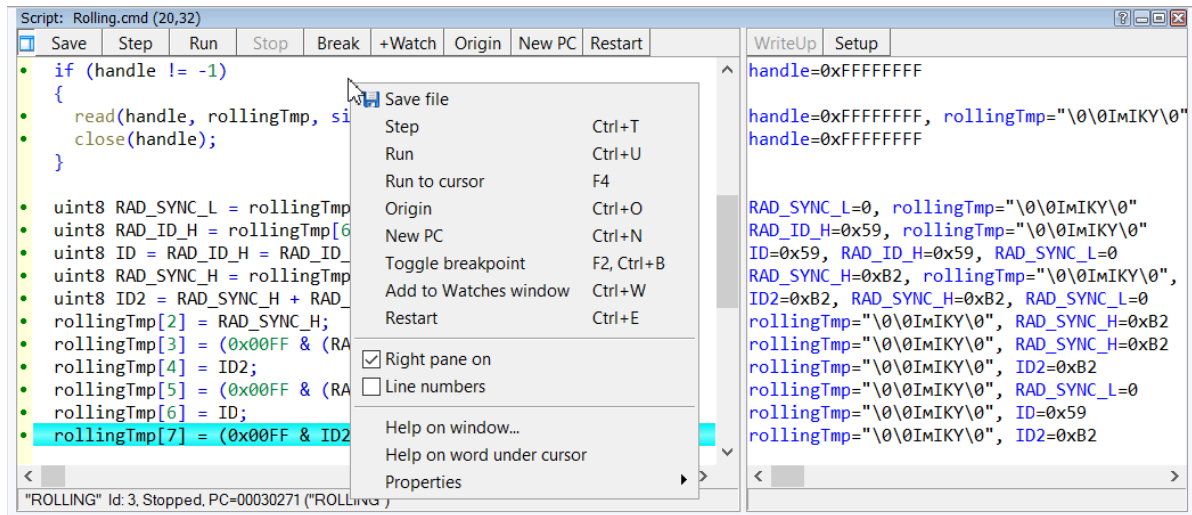
When the StartCommandFile() function in a script is called to start another script, you can specify parameter instructing it to start the new script in debug mode and open the Process window.

To view the value of a script variable in the [Watches](#)^[182][window](#)^[182], 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**.

7.4.1 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](#)^[182].

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](#)^[189]).



Note. To get help on a function or a variable, click mouse button on the function or variable name in the script source.

7.4.1.1 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 ^[182] window ^[182] (if not already open) and places the name at the cursor into it.
Restart	Restart	Restarts execution of the highlighted script.

7.4.1.2 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.

7.4.2 The Watches Window

[AutoWatches](#)^[182] 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](#)^[202]. 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 the corresponding boxes in the **Configure** menu > **Environment** > **Fonts** tab.

Context Menu

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

<u>Command</u>	<u>Description</u>
Add Watch	Adds one or more objects to the window. Opens the Add Watch ^[184] dialog to choose an object by name. Also, you can enter an expression ^[202] 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 ^[183] dialog to change the display settings for selected object and also to add/delete tabs to/from the window.

7.4.2.1 The Display Watches Options Dialog

Use this dialog to set the display options for the selected variable or [expression](#) ^[202] in the **Watches** ^[182] 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** ^[57] menu, the **Environment** dialog, the **Fonts** ^[80] tab and check the corresponding boxes in the **Grid** field.

7.4.2.2 The Add Watch Dialog

Use this dialog to add symbol names (for example, a variable name or an [expression](#)^[2021]) to the **Watches** window. The dialog contains a list of symbol names defined in, or known to, the program.

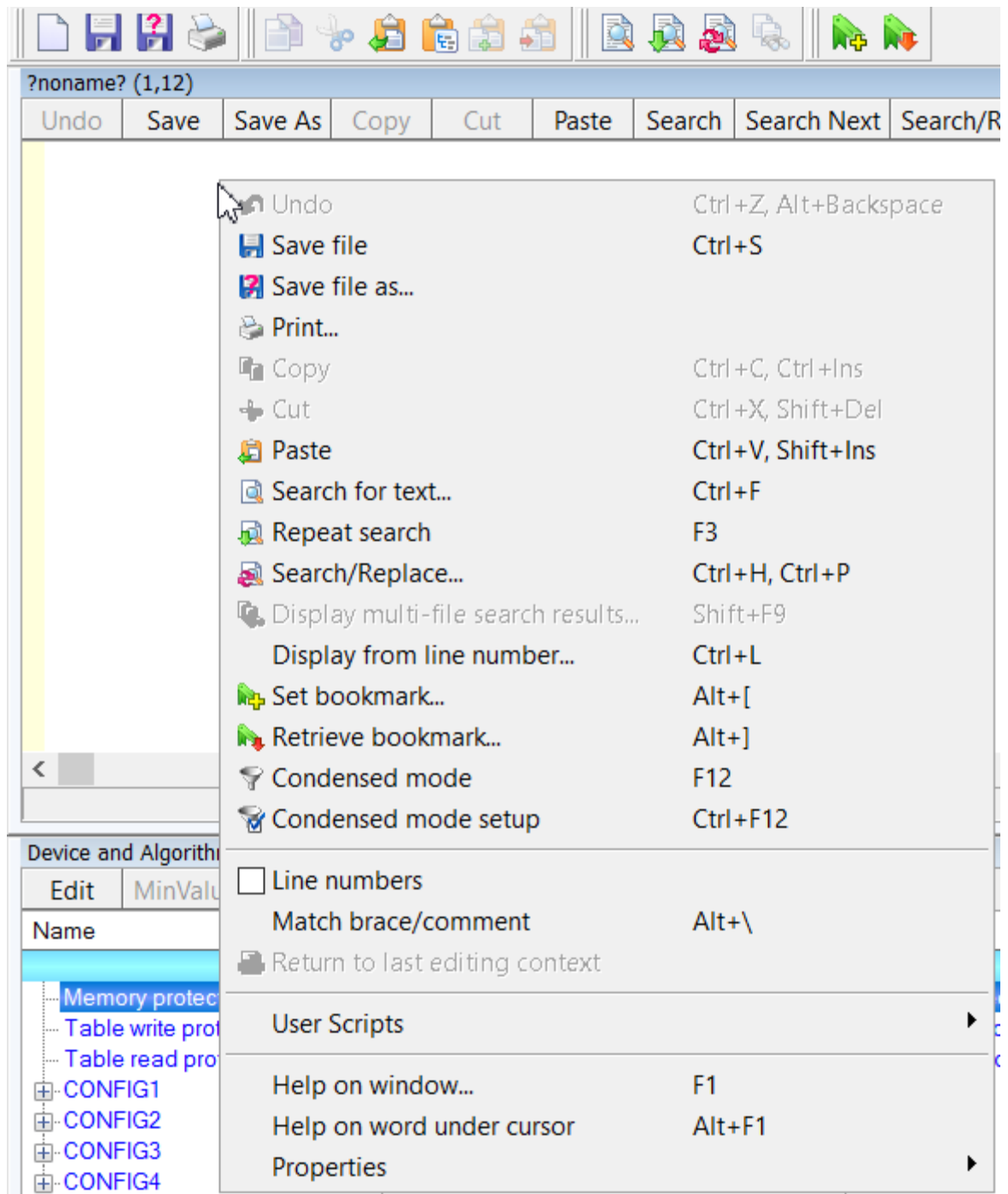
Dialog Control	Description
Name or expression to watch:	Enter the symbol name or expression to be added. You can specify several names and expressions either manually (separated with semicolons) or by selecting from the list with the Ctrl key pressed.
History	List of previous names and expressions.

7.5 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 by 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 file name and location.





7.5.1 The File Menu


Commands in this menu act on the currently active [Edit](#)¹⁸⁶ window.

Button	Command	Description
	New	Opens the Editor window ¹⁸⁶ 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.

7.5.2 The Edit Menu

Commands of this menu act on the active [Edit](#)¹⁸⁶ 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 ⁸³ 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 ^[190] dialog.
	Next Search	Repeats search with parameters used in the previous search.
	Replace	Opens the Replace Text ^[191] dialog.
	Display Multi-file Search Results	Re-opens the last multi-file search results in the Multi-File Search Results ^[192] dialog.
	Display from line number...	Opens the Display from Line Number ^[194] dialog for you to specify a line number. Source text will be displayed from this line.
	Set bookmark...	Opens the Set Bookmark ^[193] dialog to set a local bookmark.
	Retrieve bookmark	Opens the Retrieve Bookmark ^[193] dialog to retrieve a local bookmark.
	Condensed mode	Toggles Condensed display mode ^[188] on and off.
	Condensed mode setup	Opens the Condensed Mode Setup ^[194] 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.

7.5.3 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**^[83] 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):

```
char Timer0 far;
char Timer1 far;
char Osc    far;
char DMA    far;

uint8 RAD_SYNC_L = rollingTmp[7];
uint8 RAD_SYNC_H = rollingTmp[6];
```

The **Vertical Blocks** checkbox in the the **Main menu > Configure>Editor Options> General** ⁸³ 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.

7.5.4 Condensed Mode

In the Condensed mode, only lines that satisfy a specific criterion 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**^[194] 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.

7.5.5 Syntax Highlighting

When the **Source**^[186] 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**^[83] **tab>Syntax Highlighting** flag. In addition, you can change the color of each construction; to do so use **Main menu > Configure> Environment > Colors**^[80] **tab**.

7.5.6 Automatic Word Completion

It is normal for words (labels, names of variables) to be repeated within 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**^[83] **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.

7.5.7 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.

7.5.8 Dialogs

This section describes dialogs used by Script Editor.

7.5.8.1 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 ¹⁹³ .
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](#)¹⁹²⁾ dialog remains open.

7.5.8.2 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 ¹⁹³⁾ .
Prompt at Replace	Checked by default. If checked, the editor will always pop up the Confirm Replace ¹⁹²⁾ 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: <code>*.txt;*.c;c:\prog*.h</code> . 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: <code>c:\prog\text\source*.txt;c:\prog\text\source*.doc</code> In this case, make use of wildcards (<code>*.txt;*.doc</code>) and common path (<code>c:\prog\text\source</code>).

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](#)^[192] dialog remains open.

7.5.8.3 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](#)^[191] 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.

7.5.8.4 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](#)^[190] 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.

7.5.8.5 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 <i>?ell</i> 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: <i>%Counter</i> - 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: <i>Counter\$</i> - find the word "Counter" at the line end.
@	Match the next character literally; '@' lets you specify the control characters as usual letters. Example: <i>@?</i> - search for the question mark character.
\xNN	The hexadecimal value of the character. Example: <i>\xA7</i> - find the character with the hexadecimal code of A7.
+	Indefinite number of repetitions of the previous character. For example, if you specify <i>1T+2</i> , 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: <i>[A-Z]</i> means any letter from A to Z.
[~c1-c2]	Match any character whose value is outside the interval from c1 to c2. Example: <i>[~A-Z]</i> 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: <i>LPT COM CON</i> means search for "LPT" or "COM" or "CON."

7.5.8.6 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.

7.5.8.7 The Condensed Mode Setup Dialog

This dialog sets up the parameters for the [Condensed mode](#)^[188] of the [Source](#)^[186] 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](#)^[193].
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.

7.5.8.8 The Display from Line Number Dialog

Use this dialog to display source file in the active [Source](#)^[186] window starting with specified line. Enter the line number or select any previous number from the **History** list. Line numbers start with 1.

8 Reference

8.1 How to ...

This chapter describes typical operations with a CPI2-B1 device programmer running in the [Single-programming](#)^[26] **control** mode. The description refers to the operation made withing the ChipProg-02 [GUI](#)^[120], only.

8.1.1 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](#)^[105] window, or
 - c) Select the **Blank check** function line in the **Function** list of the [Program Manager](#)^[105] 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](#)^[105] window, or for the warning message if the device is not blank.

8.1.2 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](#)^[105] window or
 - c) Select the **Erase** function line in the **Function** list of the [Program Manager](#)^[105] 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](#)^[105] window or for the warning message if the device is not blank after erasing.

8.1.3 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](#)^[105] window, or
- select the **Read** function line in the **Function** list of the [Program Manager](#)^[105] 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](#)^[105] window or for the warning message if the device could not be read.

8.1.4 How to program a device

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

- [load the file](#)^[195] that you want to write to the device;
- [edit the file](#)^[196] (if necessary);
- [configure](#)^[196] the device to be programmed (if necessary);
- [write](#)^[196] the prepared information into the device and verify the programming.

8.1.4.1 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](#)^[18] and sub-level to load the file to.
3. Wait for the message **File loaded: "....."** in the [Program Manager](#)^[105] window, or for a warning message if the file cannot be loaded for some reason.

8.1.4.2 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](#)^[95] window. Please keep in mind that the **View** button must be released to enable editing.
2. Make necessary changes using [Modify](#)^[100] dialog or select the data to be modified and type new data over old data.

8.1.4.3 How to configure target device

1. Parameters displayed in the [Device and Algorithm Parameters](#)^[93] 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 want 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](#)^[105] window.

8.1.4.4 How to write information into the device

1. Click on the [Options](#)^[109] tab in [Program Manager](#)^[105] window. Check the options you need. We recommend you always check [Blank check](#)^[194] before programming and [Verify](#)^[197] after programming to ensure reliable programming.
2. Click on the [Program Manager](#)^[107] 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](#)^[107] 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](#)^[93] 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.

8.1.5 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](#)^[105] window, or
- select the **Verify** function line in the **Function** list of the [Program Manager](#)^[105] window and click the **Execute** button, or
- select the **Commands > Verify** menu command.

Wait for the message **Verifying ... OK** in the [Program Manager](#)^[105] window or for a warning message if the device verification has failed.

8.1.6 How to save data to disc

1. After you have read device content into the [Buffer](#)^[17] or specified [Buffer layer](#)^[17] 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.

8.1.7 Multi-Target Programming

Multi-target device programming

In production environments, maximum programming efficiency is an important goal. It is possible to organize several CPI2-B1 programmers into multiple virtual programmer clusters in order to achieve concurrent parallel programming that takes the least amount of time to accomplish. Consider the following example.:

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

For concurrent, parallel programming, $4 \times 3 = 12$ CPI2-B1 device programmers would be required. A typical scenario of use is as follows:

1. Split 12 programmers in 4 groups by 3x CPI2-B1 programmers in each. Each group will independently and concurrently program three different devices on one board. Consequently, 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 three programming [projects](#)^[47] - one for each target device. Save their **.upp** files that includes device types, file names and other options. It is assumed that these are well debugged projects. This can be accomplished prior to gang programming by using one CPI2-B1 programmer working in a single-programming mode for each of the devices on the boards.
4. Launch three instances of the ChipProg-02 program in the gang mode. In the command line of the [startup](#)^[38] dialog specify serial numbers of the programmers - four numbers per 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](#)^[24] or manually by executing the [Auto Programming](#)^[108] command in the [GUI](#)^[46]. Then replace target panels upon successful programming of all 12 devices.

8.2 Error Messages

8.2.1 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."

8.2.2 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."

8.2.3 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 = %08IX, 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'"

.

8.2.4 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"

8.2.5 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."

8.2.6 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'"

8.2.7 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."

8.2.8 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 Phytion."

4000 "The attached programmer with id = %u is not supported"

4102 "Device programming countdown value is zero%s"

8.2.9 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."

8.2.10 Error configuration

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')."

8.2.11 Error device

5326 "Device selection error"

4018 "Device '%s' is not supported by the %s. Please choose another device."

8.2.12 Error check box

6852 "Error in check box option specification string: '=' expected"

6853 "Cannot find check box option string '%s'"

8.2.13 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"

8.2.14 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."

8.3 Expressions

Expressions are mathematical constructs for [operations](#)^[202] on one or more [operands](#)^[204].

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`.

8.3.1 Operations

The program supports all arithmetic and logical operations valid for the C language, as well as pointer and address operations:

<u>Designation</u>	<u>Description</u>
()	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 *).

8.3.2 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; 11111111111111000011B

5) Symbol (ASCII).

Examples: 'a'; 'ab'; '\$B%8'.'. .

8.3.3 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)
```

8.4 Scripting Reference

[Description of Script Language](#) ²⁰⁵

[Script Language Built-in Functions](#) ²³¹

[Script Language Built-in Variables](#) ^[256]

[Alphabetical List of Script Language Built-in Functions and Variables](#) ^[256]

8.4.1 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](#) ^[205].

Here are the links to the sections of this scripting language manual.

[General Syntax of Script Language](#) ^[206]

[Basic Data Types](#) ^[209]

[Data byte order](#) ^[210]

[Operations and Expressions](#) ^[210]

[Operators](#) ^[220]

[Functions](#) ^[225]

[Descriptions](#) ^[227]

[Directives of the Script File Language Preprocessor](#) ^[230]

[Predefined Symbols in the Script File Compilation](#) ^[231]

8.4.1.1 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](#) ^[343], 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](#) ^[317], 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](#) ^[319], with a variable number of parameters.
- Declaration of **user** function parameters such as **void array[]** is not supported. The system functions such as [memcpy](#) ^[317], 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()
{
    ...
}
```

8.4.1.2 Scripting Language Syntax

[Format](#)^[206]
[Comments](#)^[206]
[Identifiers](#)^[207]
[Reserved words](#)^[207]
[Integer Constants](#)^[207]
[Long integer constants](#)^[208]
[Floating-point Constants](#)^[208]
[Character Constants](#)^[209]
[String Constants](#)^[209]

8.4.1.2.1 Format

Spaces, tabs, line advance and page advance symbols are used as separators. You can use any number of these separator symbols.

8.4.1.2.2 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 */
```

8.4.1.2.3 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
```

8.4.1.2.4 Reserved words

```
break    extern  return
case     float   short
char     for     sizeof
continue goto    switch
default  if      unsigned
do       int     void
else     long    while
```

8.4.1.2.5 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.

8.4.1.2.6 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)

8.4.1.2.7 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).
```

8.4.1.2.8 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)	<code>'\n'</code>
Horizontal tabulation	HT	<code>'\t'</code>
Vertical tabulation	VT	<code>'\v'</code>
Backspacing	BS	<code>'\b'</code>
Carriage return	CR	<code>'\r'</code>
Form feed	FF	<code>'\f'</code>
Backslash	<code>\</code>	<code>'\\'</code>
Apostrophe	<code>'</code>	<code>'\''</code>
Quotation marks	<code>"</code>	<code>'\"'</code>
Zero character (null)	NUL	<code>'\0'</code>

Note. The character constants are considered to be the int-type data.

8.4.1.2.9 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"
"_____ "
" "
```

8.4.1.3 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.

8.4.1.4 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

8.4.1.5 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](#) ²¹¹

[Arithmetic Operations](#) ²¹¹

[Assignment Operations](#) ²¹²

[Relation Operations](#) ²¹⁴

[Logical Operations](#) ²¹⁵

[Bit Operations](#) ²¹⁶

[Array Operations](#) ²¹⁶

[Other Operations](#) ²¹⁷

[Operation Execution Priorities and Order](#) ²¹⁸

[Operand Execution Order](#) ²¹⁹

[Arithmetic Conversions in Expressions](#)

8.4.1.5.1 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.

8.4.1.5.2 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;
```

8.4.1.5.3 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;
```

8.4.1.5.4 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.

8.4.1.5.5 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++;
```

8.4.1.5.6 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].

8.4.1.5.7 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;
```

8.4.1.5.8 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);
```

8.4.1.5.9 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
=		
*=	/=	%=
+=	-=	
<<=	>>=	&=
^=	=	

8.4.1.5.10 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);
```

8.4.1.5.11 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.

8.4.1.6 Operators

[Format and nesting](#) ^[220]

[Operator label](#) ^[220]

[Composite operator](#) ^[220]

[Operator-expression](#) ^[221]

[Operator Break](#) ^[221]

[Operator Continue](#) ^[222]

[Operator Return](#) ^[222]

[Operator Goto](#) ^[222]

[Conditional Operator If-Else](#) ^[222]

[Cycle Operator While](#) ^[224]

[Cycle Operator Do-While](#) ^[224]

[Cycle Operator For](#) ^[225]

8.4.1.6.1 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.

8.4.1.6.2 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;
```

8.4.1.6.3 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;  
}
```

8.4.1.6.4 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.

8.4.1.6.5 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;
```

8.4.1.6.6 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];  
}
```

8.4.1.6.7 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;
```

8.4.1.6.8 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;
```

8.4.1.6.9 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");
```

8.4.1.6.10 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++;
}
```

8.4.1.6.11 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);
```

8.4.1.6.12 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.

8.4.1.7 Functions

[Function Definition](#) 

[Function Call](#) 

[Function Main](#) 

8.4.1.7.1 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
```

<code>func(</code>	function name
<code> long a, char str[]</code>	list of parameters, which describes the names and
<code>types</code>	
<code>)</code>	
<code>{</code>	composite operator
<code> // ...</code>	
<code> return 0;</code>	returned value
<code>}</code>	

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 errmsg(char s[])
{
    printf{****Error: %s", s);
    // the Return operator (explicit) is not required
}
```

8.4.1.7.2 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.

8.4.1.7.3 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.

8.4.1.8 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](#)^[227]

[Arrays](#)^[227]

[Local Variable Definition](#)^[228]

[Global Variable Definition](#)^[228]

8.4.1.8.1 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](#)^[209]

8.4.1.8.2 Arrays

Only one-dimensional arrays are supported.

Example:

```
int a[50];
```

Variable a is the array consisting of 50 integer numbers.

8.4.1.8.3 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.

8.4.1.8.4 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](#) ²²⁹

[External Object Description](#) ²²⁹

8.4.1.8.4.1 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'};
```

8.4.1.8.4.2 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` when 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.

8.4.1.9 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](#) 

8.4.1.9.1 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.

8.4.1.9.2 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.

8.4.1.9.3 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](#).^[178]

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
```

8.4.1.10 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.

8.4.2 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](#) ²³²

[Device programming control functions](#) ²³⁷

[Mathematical Functions](#) ²⁴⁵

[String Operation Functions](#) ²⁴⁶

[Character Operation Functions](#) ²⁴⁷

[Functions for File and Directory Operation](#) ²⁴⁷

[Stream File Functions](#) ²⁴⁹

[Formatted Input-Output Functions](#) ²⁵⁰

[Script File Manipulation Functions](#) ²⁵⁰

[Text Editor Functions](#) ²⁵⁰

[Control Functions](#) ²⁵²

[Windows Operation Functions and Other System Functions](#) ²⁵³

[Graphical Output Functions](#) ²⁵³

[I/O Stream](#) ²⁵⁴ [Window Operation Functions](#) ²⁵⁴

[Event Wait Functions](#) ²⁵⁵

[Other Various Functions](#) ²⁵⁵

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**.

8.4.2.1 Buffer access functions

[LoadProgram](#) ²³⁴

[ReloadProgram](#) ²³⁵

[SaveData](#) ²³⁵

[SetDevice](#) ²³⁶

[MinAddr](#) ²³⁵

[MaxAddr](#) ²³⁵

[GetByte](#) ²³³

[GetWord](#) ²³⁴

[GetDword](#) ²³³

[SetByte](#) ²³⁶

[SetWord](#) ²³⁷

[SetDword](#) ²³⁶

[GetMemory](#) ²³³

[SetMemory](#) ²³⁷

[Checksum](#) ²³²

8.4.2.1.1 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("%08IX", CheckSum(0, 0x1FFF, SubLevel(1, 0)));
```

8.4.2.1.2 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));
```

8.4.2.1.3 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("%08IX", GetDword(0, 0x1F));
```

8.4.2.1.4 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));
```

8.4.2.1.5 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));
```

8.4.2.1.6 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));
```

8.4.2.1.7 MaxAddr

```
unsigned long MaxAddr(int addr_space);
```

Description

Returns the address of the address space{addr_sp} upper boundary.

8.4.2.1.8 MinAddr

```
unsigned long MinAddr(int addr_space);
```

Description

Returns the address of the address space{addr_sp} lower boundary.

8.4.2.1.9 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](#) ⁵¹ menu.

8.4.2.1.10 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);
```

8.4.2.1.11 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);
```

8.4.2.1.12 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");
```

8.4.2.1.13 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);
```


8.4.2.1.14 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);
```

8.4.2.1.15 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);
```

8.4.2.2 Device programming control functions and variables

Here is the list of the functions that control programming scripts (alphabetic order):

[AllProgOptionsDefault](#)^[238]
[ExecFunction](#)^[238]
[GetProgOptionBits](#)^[240]
[GetProgOptionFloat](#)^[240]
[GetProgOptionList](#)^[240]
[GetProgOptionLong](#)^[241]
[GetProgOptionString](#)^[241]
[mprintf](#)^[241]
[ProgOptionDefault](#)^[241]
[SetProgOption](#)^[242]

Here is the list of the variables that controls programming operations in scripts (alphabetic order):

[BlankCheck](#)^[243]
[BufferStartAddr](#)^[243]
[ChipEndAddr](#)^[243]
[ChipStartAddr](#)^[243]
[DialogOnError](#)^[244]

[InsertTest](#)^[244]
[LastErrorMessage](#)^[244]
[ReverseBytesOrder](#)^[244]
[VerifyAfterProgram](#)^[245]
[VerifyAfterRead](#)^[245]

8.4.2.2.1 Function AllProgOptionsDefault

void AllProgOptionsDefault();

Description:

Set all the programming options to their default values.

8.4.2.2.2 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](#)^[105].

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](#)^[244].

Example:

```
if (ExecFunction("Blank Check") != EF_OK)
    printf("Error in blank check: %s", LastErrorMessage);
```

See also [DialogOnError](#)^[244].

8.4.2.2.3 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()**.

8.4.2.2.4 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).

8.4.2.2.5 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.

8.4.2.2.6 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.

8.4.2.2.7 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.

8.4.2.2.8 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.

8.4.2.2.9 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.

8.4.2.2.10 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.

8.4.2.2.11 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.

8.4.2.2.12 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.

8.4.2.2.13 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.

8.4.2.2.14 Function mprintf

```
void mprintf(char format[], ... );
```

Description:

The **mprintf** function is used just like [printf](#)^[340] but the message is displayed not in the **Console** window but in the "Operation Progress" window of the [Program Manager](#)^[105] window.

8.4.2.2.15 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.

8.4.2.2.16 Function ProgOptionDefault

```
void ProgOptionDefault(char option_name[]);
```

Description:

Set the default value of the **option_name** programming option.

8.4.2.2.17 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).

8.4.2.2.18 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](#) ⁹³ 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](#) ⁹³ 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](#) ⁹³ 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.

8.4.2.2.19 Function WriteShadowArea

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).

8.4.2.2.20 Variable BlankCheck

extern int BlankCheck;

The value of the "Blank check before program" option in the [Program Manager](#)¹⁰⁵ window (tab **Options**). Assigning value to **BlankCheck** automatically changes the option in the window and vice versa.

8.4.2.2.21 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.

8.4.2.2.22 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.

8.4.2.2.23 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.

8.4.2.2.24 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.

8.4.2.2.25 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.

8.4.2.2.26 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](#)^[238]), the dialog with error description is displayed. Otherwise no dialog is displayed and ExecFunction() immediately returns with code EF_ERROR.

8.4.2.2.27 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.

8.4.2.2.28 Variable InsertTest

```
extern int InsertTest;
```

The value of the "Insert test" option in [The Program Manager Window](#)^[105] (tab Options). Assigning value to InsertTest automatically changes the option in the window and vice versa.

8.4.2.2.29 Variable LastErrorMessage[]

```
extern char LastErrorMessage[];
```

String that contains the last error message about operation on device. See also [ExecFunction](#)^[238].

8.4.2.2.30 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.

8.4.2.2.31 Variable ReverseBytesOrder

```
extern int ReverseBytesOrder;
```

The value of the "Reverse bytes order" option in [The Program Manager Window](#)^[105] (tab Options). Assigning value to ReverseBytesOrder automatically changes the option in the window and vice versa.

8.4.2.2.32 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.

8.4.2.2.33 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.

8.4.2.2.34 Variable VerifyAfterProgram

extern int VerifyAfterProgram;

The value of the "Verify after program" option in [The Program Manager Window](#)^[105] (tab Options). Assigning value to VerifyAfterProgram automatically changes the option in the window and vice versa.

8.4.2.2.35 Variable VerifyAfterRead

extern int VerifyAfterRead;

The value of the "Verify after read" option in [The Program Manager Window](#)^[105] (tab Options). Assigning value to VerifyAfterRead automatically changes the option in the window and vice versa.

8.4.2.3 Mathematical functions

[sin](#)^[340]

[asin](#)^[268]

[cos](#)^[274]

[acos](#)^[266]

[tan](#)^[349]

[tanh](#)^[349]

[atan](#)^[268]

[log](#)^[309]

[log10](#)

[sqrt](#)^[340]

[ceil](#)^[271]

[floor](#)^[287]

[exp](#)^[282]

[fabs](#)^[282]

[fmod](#)^[288]

[frexp](#)^[292]

[abs](#) ²⁶⁶

[pow](#) ³¹⁸

[pow10](#) ³¹⁸

8.4.2.4 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](#) ³¹⁰

[memcpy](#) ³¹¹

[memmove](#) ³¹²

[movmem](#) ³¹⁴

[memchr](#) ³¹⁰

[memset](#) ³¹²

[setmem](#) ³³⁶

[memcmp](#) ³¹¹

[memicmp](#) ³¹¹

[stpcpy](#) ³⁴²

[strcat](#) ³⁴²

[strchr](#) ³⁴²

[strcmp](#) ³⁴²

[stricmp](#) ³⁴³

[strcmpi](#) ³⁴³

[\[****\]](#) ³⁴³

[strcpy](#) ³⁴³

[\[****\]](#) ³⁴⁴

[strlwr](#) ³⁴⁴

[\[****\]](#) ³⁴⁹

[strncat](#) ³⁴⁴

[strncmp](#) ³⁴⁵

[strncmpi](#) ³⁴⁵

[strnicmp](#) ³⁴⁶

[strncpy](#) ³⁴⁵

[strnset](#) ³⁴⁶

[strpbrk](#) ³⁴⁶

[strspn](#) ³⁴⁷

[****](#) ³⁴⁷

[strchr](#) ³⁴⁶

[strrev](#)

[****](#) ³⁴⁷

8.4.2.5 Character operation functions

[isalnum](#) ³⁰²

[isalpha](#)

[isascii](#) ³⁰²

[iscntrl](#) ³⁰³

[isdigit](#) ³⁰³

[isgraph](#) ³⁰³

[islower](#) ³⁰³

[isprint](#) ³⁰⁴

[ispunct](#) ³⁰⁴

[isspace](#) ³⁰⁴

[isupper](#)

[isxdigit](#) ³⁰⁴

[toascii](#) ³⁵⁰


[tolower](#) ³⁵¹

[toupper](#) ³⁵¹

8.4.2.6 Functions for file and directory operation

[chdir](#) ²⁷¹

[getcurdir](#) ²⁹⁵

[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](#)  309

[open](#)  314

[read](#)  327

[write](#)  358

[rename](#)  329

[setmode](#)  337


[****](#) 

8.4.2.7 Stream file functions

[clearerr](#)

[fclose](#)  282

[fdopen](#)  283

[feof](#)  283

[ferror](#)  284

[fflush](#)  284

[fgetc](#)  284

[fgets](#)  285

[fileno](#)  285

[fopen](#)  288

[fprintf](#)  289

[fputc](#)  290

[fputs](#)  290

[fread](#)  291

[freopen](#)

[fscanf](#)  292

[fseek](#)  293


[ftell](#)  293

[fwrite](#)  294

[getc](#)  294

[getw](#)  299

[putc](#)  326

[****](#)  326

[rewind](#)

8.4.2.8 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](#) ²⁸⁹

[fscanf](#) ²⁹²

[scanf](#) ³³¹

[pscanf](#) ³²⁵

[sscanf](#) ³⁴¹

[printf](#) ³¹⁹

[_printf](#) ²⁶⁶

[sprintf](#) ³⁴⁰

[MessageBox](#)

[MessageBoxEx](#) ³¹²

8.4.2.9 Script File Manipulation Functions

[ExecScript](#) ²⁸¹

[GetScriptFileName](#) ²⁹⁸

[TerminateScript](#) ³⁵⁰

[TerminateAllScripts](#) ³⁵⁰

[exit](#) ²⁸¹

8.4.2.10 Text editor functions

The text editor functions manipulate with text in the [Source](#) ¹⁸⁶. 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](#)^[300]

[Up](#)

[Down](#)^[278]

[Left](#)

[Right](#)^[329]

[Tof](#)^[350]

[Eof](#)^[279]

[Eol](#)^[280]

[BackSpace](#)^[269]

[Cr](#)^[274]

[DelLine](#)^[277]

[DelChar](#)^[276]

[CurChar](#)^[276]

[GetLine](#)^[297]

[ForwardTill](#)^[289]

[ForwardTillNot](#)^[289]

[_GetWord](#)^[265]

[WordLeft](#)^[358]

[WordRight](#)^[358]

[FirstWord](#)^[287]

[SetMark](#)^[336]

[GetMark](#)^[297]

[Text](#)^[350]

[BlockBegin](#)^[269]

[BlockEnd](#)^[270]

[BlockOff](#)^[270]

[BlockCopy](#)^[269]

[BlockFastCopy](#)^[270]

[BlockDelete](#)^[269]

[BlockMove](#)^[270]

[BlockPaste](#)^[270]

[Search](#)^[332]

[SearchReplace](#) ³³³

[SetFileName](#) ³³⁵

[GetFileName](#) ²⁹⁶

[SaveFile](#) ³³¹

[****]

[OpenEditorWindow](#) ³¹⁵

Text editor built-in variables

[InsertMode](#) ³⁶²

[CaseSensitive](#) ³⁶¹

[WholeWords](#) ³⁶⁴

[****] ³⁶³

[BlockCol1](#) ³⁶⁰

[BlockCol2](#) ³⁶⁰

[BlockLine1](#) ³⁶⁰

[BlockLine2](#) ³⁶⁰

[BlockStatus](#) ³⁶⁰

[CurLine](#) ³⁶¹

[CurCol](#) ³⁶¹

[LastFoundString](#) ³⁶²

8.4.2.11 Debug shell control functions

These functions control CPI2-B1.

[RedrawScreen](#) ³²⁸

[LoadDesktop](#) ³⁰⁶

[LoadOptions](#) ³⁰⁷

[SaveDesktop](#) ³³⁰

[SaveOptions](#) ³³¹

[OpenWindow](#) ³¹⁶

[OpenUserWindow](#) ³¹⁶

[OpenStreamWindow](#) ³¹⁵

[CloseWindow](#) ²⁷³

[FindWindow](#) ²⁸⁷

[MoveWindow](#) ³¹⁴

[ActivateWindow](#) ²⁶⁶

[SetWindowSize](#) ³³⁹

[SetWindowSizeT](#) ³³⁹

[GetWindowWidth](#) ²⁹⁹

[GetWindowHeight](#) ²⁹⁹

[SetWindowFont](#) ^[338]
[WindowHotkey](#) ^[357]
[AddWatch](#) ^[267]
[Inspect](#) ^[301]
[ExecMenu](#) ^[280]
[ExitProgram](#) ^[282]
[LoadProject](#) ^[308]
[CloseProject](#) ^[273]
[LoadProgram](#) ^[307]
[ReloadProgram](#) ^[328]
[SaveData](#) ^[330]

8.4.2.12 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](#) ^[268]
[LoadLibrary](#) ^[307]
[FreeLibrary](#) ^[291]
[WaitSendMessage](#) ^[355]
[WaitGetMessage](#) ^[354]
[inport](#) ^[301]
[inportb](#) ^[301]
[outport](#) ^[317]
[outportb](#) ^[317]
[peek](#) ^[317]
[peekb](#) ^[317]
[poke](#) ^[317]
[pokeb](#) ^[318]
[exec](#) ^[280]
[getenv](#) ^[296]
[putenv](#) ^[326]

8.4.2.13 Graphical output functions

Graphical output functions draw various graphical objects and text in special [User window](#) ^[180]. To draw in a user window, first open it with the [OpenUserWindow](#) ^[316] 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](#) ^[180].

In all graphical output functions, the first parameter () is the window identifier.

[OpenUserWindow](#) ^[316][ClearWindow](#) ^[273][SetCaption](#) ^[335][SetToolbar](#) ^[338][SetUpdateMode](#) ^[338][UpdateWindow](#) ^[352][SelectPen](#) ^[333][SelectBrush](#) ^[333][SelectFont](#) ^[333][SetTextColor](#) ^[337][SetBkColor](#) ^[334][SetBkMode](#) ^[334][DisplayText](#) ^[277][DisplayTextF](#) ^[277][MoveTo](#) ^[314][LineTo](#) ^[306][FillRect](#) ^[286][Rectangle](#) ^[328][\[****\]](#) ^[290][InvertRect](#) ^[301][Curcuit](#) ^[276][Ellipse](#) ^[279][Polyline](#) ^[318][SetPixel](#) ^[337][AddButton](#) ^[268][RemoveButtons](#) ^[328][WaitWindowEvent](#) ^[356][\[****\]](#) ^[305][LastEventInt{1...4}](#) ^[306]

8.4.2.14 I/O Stream window operation functions

Stream window control functions allow you to display text in the special [I/O Stream window](#) ^[180].

In all Stream window control functions, the first parameter (handle) is the window identifier.

[OpenStreamWindow](#) ^[315]

[SetTextColor](#) ^[337]

[wprintf](#) ^[358]

[wgetchar](#) ^[356]

[LastChar](#) ^[305]

[wgethex](#) ^[357]

[wgetstring](#) ^[357]

[LastString](#) ^[306]

8.4.2.15 Event Wait Functions

These extremely useful functions serve to simulate external environment. Also, you can use them in simulators to develop various tests.

[Wait](#) ^[352]

[WaitMemoryAccess](#) ^[354]

[WaitExprTrue](#) ^[353]

[WaitExprChange](#)

[WaitStop](#) ^[356]

[WaitWindowEvent](#) ^[356]

8.4.2.16 Other Various Functions

[delay](#) ^[276]

[gettime](#) ^[298]

[getdate](#) ^[295]

[difftime](#) ^[277]

[atof](#) ^[268]

[atoi](#) ^[269]

[itoa](#) ^[305]

[ltoa](#) ^[310]

[ultoa](#) ^[351]

[rand](#) ^[327]

[random](#) ^[327]

[randomize](#) ^[327]

[srand](#) ^[341]

[strtol](#) ^[348]

[strtoul](#) ^[348]

8.4.3 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](#) ^[244]
[ReverseBytesOrder](#) ^[244]
[BlankCheck](#) ^[243]
[VerifyAfterProgram](#) ^[245]
[VerifyAfterRead](#) ^[245]
[ChipStartAddr](#) ^[243]
[ChipEndAddr](#) ^[243]
[BufferStartAddr](#) ^[243]
[LastErrorMessage](#) ^[244]
[DialogOnError](#) ^[244]

Text editor built-in variables:

[InsertMode](#) ^[362]
[CaseSensitive](#) ^[361]
[WholeWords](#) ^[364]
[RegularExpressions](#) ^[363]
[BlockCol1](#) ^[360]
[BlockCol2](#) ^[360]
[BlockLine1](#) ^[360]
[BlockLine2](#) ^[360]
[BlockStatus](#) ^[360]
[CurLine](#) ^[361]
[CurCol](#) ^[361]
[LastFoundString](#) ^[362]

Miscellaneous variables:

[WorkFieldWidth](#) ^[364]
[WorkFieldHeight](#) ^[364]
[AppIName\[\]](#) ^[359]
[DesktopName\[\]](#) ^[361]
[SystemDir\[\]](#) ^[364]
[errno](#) ^[361]
[_fmode](#) ^[359]
[MainWindowHandle](#) ^[363]
[NumWindows](#) ^[363]
[WindowHandles\[\]](#) ^[364]
[SelectedString\[\]](#) ^[364]
[LastMessageInt](#) ^[363]
[LastMessageLong](#) ^[363]

8.4.4 List of Built-in Functions and Variables

Below is the alphabetical list of all built-in functions and variables of scripting language.

[AllProgOptionsDefault](#) ^[238]
[API](#) ^[268]
[ActivateWindow](#) ^[266]

[AddButton](#) ²⁶⁶
[AddWatch](#) ²⁶⁷
[ApplName\[\]](#) ³⁵⁹
[BackSpace](#) ²⁶⁹
[BlankCheck](#) ²⁴³
[BlockBegin](#) ²⁶⁹
[BlockCol1](#) ³⁶⁰
[BlockCol2](#) ³⁶⁰
[BlockCopy](#) ²⁶⁹
[BlockDelete](#) ²⁶⁹
[BlockEnd](#) ²⁷⁰
[BlockFastCopy](#) ²⁷⁰
[BlockLine1](#) ³⁶⁰
[BlockLine2](#) ³⁶⁰
[BlockMove](#) ²⁷⁰
[BlockOff](#) ²⁷⁰
[BlockPaste](#) ²⁷⁰
[BlockStatus](#) ³⁶⁰
[BufferStartAddr](#) ²⁴³
[CaseSensitive](#) ³⁶¹
[Checksum](#) ²³²
[ChipEndAddr](#) ²⁴³
[ChipStartAddr](#) ²⁴³
[ClearWindow](#) ²⁷³
[CloseProject](#) ²⁷³
[CloseWindow](#) ²⁷³
[Cr](#) ²⁷⁴
[CurChar](#) ²⁷⁶
[CurCol](#) ³⁶¹
[CurLine](#) ³⁶¹
[Curcuit](#) ²⁷⁶
[DelChar](#) ²⁷⁶
[DelLine](#) ²⁷⁷
[DesktopName\[\]](#) ³⁶¹
[DialogOnError](#) ²⁴⁴
[DisplayText](#) ²⁷⁷
[DisplayTextF](#) ²⁷⁷
[Down](#) ²⁷⁸
[Ellipse](#) ²⁷⁹
[Eof](#) ²⁷⁹
[Eol](#) ²⁸⁰
[ExecFunction](#) ²³⁸
[ExecMenu](#) ²⁸⁰
[ExecScript](#) ²⁸¹
[ExitProgram](#) ²⁸²
[Expr](#) ²⁸²
[FileChanged](#) ²⁸⁵
[FillRect](#) ²⁸⁶
[FindWindow](#) ²⁸⁷

[FirstWord](#) ²⁸⁷
[FloatExpr](#) ²⁸⁷
[ForwardTill](#) ²⁸⁹
[ForwardTillNot](#) ²⁸⁹
[FrameRect](#) ²⁹⁰
[FreeLibrary](#) ²⁹¹
[GetByte](#) ²³³
[GetDword](#) ²³³
[GetFileName](#) ²⁹⁶
[GetLine](#) ²⁹⁷
[GetMark](#) ²⁹⁷
[GetMemory](#) ²⁹⁷
[GetProgOptionBits](#) ²⁴⁰
[GetProgOptionFloat](#) ²⁴⁰
[GetProgOptionList](#) ²⁴⁰
[GetProgOptionLong](#) ²⁴¹
[GetProgOptionString](#) ²⁴¹
[GetScriptFileName](#) ²⁹⁶
[GetWindowHeight](#) ²⁹⁹
[GetWindowWidth](#) ²⁹⁹
[GetWord](#) ³⁰⁰
[GotoXY](#) ³⁰⁰
[InsertMode](#) ³⁶²
[InsertTest](#) ²⁴⁴
[Inspect](#) ³⁰¹
[InvertRect](#) ³⁰¹
[LastChar](#) ³⁰⁵
[LastErrorMessage](#) ²⁴⁴
[LastEvent](#) ³⁰⁵
[LastEventInt{1...4}](#) ³⁰⁵
[LastFoundString](#) ³⁶²
[LastMessageInt](#) ³⁶³
[LastMessageLong](#) ³⁶³
[LastString](#) ³⁰⁶
[Left](#) ³⁰⁷
[LineTo](#) ³⁰⁶
[LoadDesktop](#) ³⁰⁶
[LoadLibrary](#) ³⁰⁷
[LoadOptions](#) ³⁰⁷
[LoadProgram](#) ³⁰⁷
[LoadProject](#) ³⁰⁸
[MainWindowHandle](#) ³⁶³
[MaxAddr](#) ²³⁵
[MessageBox](#) ³¹²
[MessageBoxEx](#) ³¹²
[MinAddr](#) ²³⁵
[MoveTo](#) ³¹⁴
[MoveWindow](#) ³¹⁴
[NumWindows](#) ³⁶³
[OpenEditorWindow](#) ³¹⁵

[OpenStreamWindow](#) ³¹⁵
[OpenUserWindow](#) ³¹⁶
[OpenWindow](#) ³¹⁶
[Polyline](#) ³¹⁸
[ProgOptionDefault](#) ²⁴¹
[Rectangle](#) ³²⁸
[RedrawScreen](#) ³²⁸
[RegularExpressions](#) ³⁶³
[ReloadProgram](#) ³²⁸
[RemoveButtons](#) ³²⁸
[ReverseBytesOrder](#) ²⁴⁴
[Right](#) ³²⁹
[SaveData](#) ³³⁰
[SaveDesktop](#) ³³⁰
[SaveFile](#) ³³¹
[SaveOptions](#) ³³¹
[Search](#) ³³²
[SearchReplace](#) ³³³
[SelectBrush](#) ³³³
[SelectFont](#) ³³³
[SelectPen](#) ³³³
[SelectedString\[\]](#) ³⁶⁴
[SetBkColor](#) ³³⁴
[SetBkMode](#) ³³⁴
[SetByte](#) ³³⁴
[SetCaption](#) ³³⁵
[SetDevice](#) ²³⁶
[SetDWord](#) ³³⁵
[SetFileName](#) ³³⁵
[SetMark](#) ³³⁶
[SetMemory](#) ³³⁷
[SetPixel](#) ³³⁷
[SetProgOption](#) ²⁴²
[SetTextColor](#) ³³⁷
[SetToolbar](#) ³³⁸
[SetUpdateMode](#) ³³⁸
[SetWindowFont](#) ³³⁸
[SetWindowSize](#) ³³⁹
[SetWindowSizeT](#) ³³⁹
[SetWord](#) ³³⁹
[SystemDir\[\]](#) ³⁶⁴
[TerminateAllScripts](#) ³⁵⁰
[TerminateScript](#) ³⁵⁰
[Text](#) ³⁵⁰
[Tof](#) ³⁵⁰
[Up](#) ³⁵²
[UpdateWindow](#) ³⁵²
[VerifyAfterProgram](#) ²⁴⁵
[VerifyAfterRead](#) ²⁴⁵

[WaitEprTrue](#) ³⁵³
[WaitGetMessage](#) ³⁵⁴
[WaitSendMessage](#) ³⁵⁵
[WaitWindowEvent](#) ³⁵⁶
[WholeWords](#) ³⁶⁴
[WindowHandles\[\]](#) ³⁶⁴
[WindowHotkey](#) ³⁵⁷
[WordLeft](#) ³⁵⁸
[WordRight](#) ³⁵⁸
[WorkFieldHeight](#) ³⁶⁴
[WorkFieldWidth](#) ³⁶⁴
[_GetWord](#) ²⁶⁵
[_ff_attrib](#) ²⁶⁴
[_ff_date](#) ²⁶⁴
[_ff_name](#) ²⁶⁴
[_ff_size](#) ²⁶⁵
[_ff_time](#) ²⁶⁵
[_fmode](#) ³⁵⁹
[_fullpath](#) ²⁶⁵
[_printf](#) ²⁶⁶
[abs](#) ²⁶⁶
[acos](#) ²⁶⁶
[asin](#) ²⁶⁸
[atan](#) ²⁶⁸
[atof](#) ²⁶⁸
[atoi](#) ²⁶⁹
[ceil](#) ²⁷¹
[chdir](#) ²⁷¹
[chsize](#) ²⁷¹
[clearerr](#) ²⁷²
[close](#) ²⁷³
[cos](#) ²⁷⁴
[creat](#) ²⁷⁴
[creatnew](#) ²⁷⁵
[creattemp](#) ²⁷⁵
[delay](#) ²⁷⁶
[difftime](#) ²⁷⁷
[dup](#) ²⁷⁸
[dup2](#) ²⁷⁸
[eof](#) ²⁷⁹
[errno](#) ³⁶¹
[exec](#) ²⁸⁰
[exit](#) ²⁸¹
[exp](#) ²⁸²
[fabs](#) ²⁸²
[fclose](#) ²⁸²
[fdopen](#) ²⁸³
[feof](#) ²⁸³
[ferror](#) ²⁸⁴
[fflush](#) ²⁸⁴

[fgetc](#) ²⁸⁴
[fgets](#) ²⁸⁵
[filelength](#) ²⁸⁵
[fileno](#) ²⁸⁵
[findfirst](#) ²⁸⁶
[findnext](#) ²⁸⁶
[floor](#) ²⁸⁷
[fmod](#) ²⁸⁸
[fnmerge](#) ²⁶³
[fnsplit](#) ²⁸⁸
[fopen](#) ²⁸⁸
[fprintf](#) ²⁸⁹
[fputc](#) ²⁹⁰
[fputs](#) ²⁹⁰
[fread](#) ²⁹¹
[freopen](#) ²⁹¹
[frexp](#) ²⁹²
[fscanf](#) ²⁹²
[fseek](#) ²⁹³
[ftell](#) ²⁹³
[fwrite](#) ²⁹⁴
[getc](#) ²⁹⁴
[getcurdir](#) ²⁹⁵
[getcwd](#) ²⁹⁵
[getdate](#) ²⁹⁵
[getdfree](#) ²⁹⁶
[getdisk\(\)](#) ²⁹⁶
[getenv](#) ²⁹⁶
[getftime](#) ²⁹⁶
[gettime](#) ²⁹⁸
[getw](#) ²⁹⁹
[inport](#) ³⁰¹
[inportb](#) ³⁰¹
[isalnum](#) ³⁰²
[isalpha](#) ³⁰²
[isascii](#) ³⁰²
[isatty](#) ³⁰²
[iscntrl](#) ³⁰³
[isdigit](#) ³⁰³
[isgraph](#) ³⁰³
[islower](#) ³⁰³
[isprint](#) ³⁰⁴
[ispunct](#) ³⁰⁴
[isspace](#) ³⁰⁴
[isupper](#) ³⁰⁴
[isxdigit](#) ³⁰⁴
[itoa](#) ³⁰⁵
[lock](#) ³⁵⁹
[locking](#) ³⁰⁸

[log](#)^[309]
[log10](#)^[309]
[lseek](#)^[309]
[ltoa](#)^[310]
[memccpy](#)^[310]
[memchr](#)^[310]
[memcmp](#)^[311]
[memcpy](#)^[311]
[memicmp](#)^[311]
[memmove](#)^[312]
[memset](#)^[312]
[mkdir](#)^[313]
[movmem](#)^[314]
[mprintf](#)^[241]
[open](#)^[314]
[outport](#)^[317]
[outportb](#)^[317]
[peek](#)^[317]
[peekb](#)^[317]
[poke](#)^[317]
[pokeb](#)^[318]
[pow](#)^[318]
[pow10](#)^[318]
[printf](#)^[319]
[pscanf](#)^[325]
[putc](#)^[326]
[putenv](#)^[326]
[putw](#)^[326]
[rand](#)^[327]
[random](#)^[327]
[randomize](#)^[327]
[read](#)^[327]
[rename](#)^[329]
[rewind](#)^[329]
[rmdir](#)^[329]
[scanf](#)^[331]
[searchpath](#)^[332]
[setdisk](#)^[335]
[setftime](#)^[336]
[setmem](#)^[336]
[setmode](#)^[337]
[sin](#)^[340]
[sprintf](#)^[340]
[sqrt](#)^[340]
[srand](#)^[341]
[sscanf](#)^[341]
[stpcpy](#)^[342]
[strcat](#)^[342]
[strchr](#)^[342]

[strcmp](#)³⁴²
[strcmpi](#)³⁴³
[strcpy](#)³⁴³
[strcspn](#)³⁴³
[stricmp](#)³⁴³
[strlen](#)³⁴⁴
[strlwr](#)³⁴⁴
[strncat](#)³⁴⁴
[strncmp](#)³⁴⁵
[strcmpi](#)³⁴⁵
[strncpy](#)³⁴⁵
[strnicmp](#)³⁴⁶
[strnset](#)³⁴⁶
[strpbrk](#)³⁴⁶
[strrchr](#)³⁴⁶
[strrev](#)³⁴⁷
[strset](#)³⁴⁷
[strspn](#)³⁴⁷
[strstr](#)³⁴⁷
[strtol](#)³⁴⁸
[strtoul](#)³⁴⁸
[strupr](#)³⁴⁹
[tan](#)³⁴⁹
[tanh](#)³⁴⁹
[tell](#)³⁴⁹
[toascii](#)³⁵⁰
[tolower](#)³⁵¹
[toupper](#)³⁵¹
[ultoa](#)³⁵¹
[unlink](#)³⁵¹
[unlock](#)³⁵²
[wgetchar](#)³⁵⁶
[wgethex](#)³⁵⁷
[wgetstring](#)³⁵⁷
[wprintf](#)³⁵⁸
[write](#)³⁵⁸

8.4.5 Scripting Functions

Enter topic text here.

8.4.5.1 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](#)^[288] are invertible: if you split the given path with fnsplit, then merge the resultant components with fnmerge and you end up with this path.

8.4.5.2 Function _ff_attrb

Declaration:

```
char _ff_attrb(char ffblk[]);
```

Description

Returns the attribute byte of the file found upon the function [findfirst](#)^[286] or [findnext](#)^[286] access. The **ffblk** parameter is the buffer filled with information on the file after findfirst or findnext access.

Example

See function [findfirst](#)^[286]

8.4.5.3 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](#)^[286] or [findnext](#)^[286] access. The **ffblk** parameter is the buffer filled with information on the file after the findfirst or findnext access.

Example

See function [findfirst](#)^[286]

8.4.5.4 Function _ff_name

Declaration:

```
void _ff_name(char ffblk[], char fname[]);
```

Description

Copies the name of the file found upon the function [findfirst](#)^[286] or [findnext](#)^[286] access to the **fname** 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](#)^[286]

8.4.5.5 Function `_ff_size`

Declaration:

```
long _ff_size(char ffblk[]);
```

Description

Returns the size of the file found upon the function [findfirst](#)^[286] or [findnext](#)^[286] access. The `ffblk` parameter is the buffer filled with information on the file after the `findfirst` or `findnext` access.

Example

See function [findfirst](#)^[286].

8.4.5.6 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](#)^[286] or [findnext](#)^[286] access. The `ffblk` parameter is the buffer filled with information on the file after the `findfirst` or `findnext` access.

Example

See function [findfirst](#)^[286].

8.4.5.7 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`.

8.4.5.8 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.

8.4.5.9 Function `_printfv`

Declaration:

```
void _printf(char format[], ... );
```

Description

Acts like `printf`^[319], 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.

8.4.5.10 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`.

8.4.5.11 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`.

8.4.5.12 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%>.

8.4.5.13 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](#)^[356] 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](#)^[356] function to determine, which button was clicked (there multiple buttons in the window).

Example

```
AddButton(handle, "Start", 50, 50, 70, 24);
```

8.4.5.14 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](#)^[282], [FloatExpr](#)^[287], [Operations and Expressions](#)^[210].

8.4.5.15 Function AddWatch

Declaration:

```
void AddWatch(char name[], int format=DF_HEX);
```

Description

Adds the specified name (the **name** parameter) to the [Watches](#)^[182] [window](#)^[182] 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
```

8.4.5.16 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](#)_[317], [poke](#)_[317], [peekb](#)_[317], or [pokeb](#)_[318] functions.

Example

```
int ScreenHeight = API("GetSystemMetrics", SM_CYFULLSCREEN);
```

8.4.5.17 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`.

8.4.5.18 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`.

8.4.5.19 Function atof

Declaration:

```
float atof(char s[]);
```


Description

Converts an ASCII-string (parameter **s**) into the floating-point number.

8.4.5.20 Function atoi

Declaration:

```
int atoi(char s[]);
```

Description

Converts an ASCII-string (parameter **s**) into the integer number.

8.4.5.21 Function BackSpace

Declaration:

```
void BackSpace();
```

Description

Works like the **BackSpace** key.

8.4.5.22 Function BlockBegin

Declaration:

```
void BlockBegin(int block_type);
```

Description

Begins marking of block (see [Block Operations](#)^[187]). 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

8.4.5.23 Function BlockCopy

Declaration:

```
void BlockCopy();
```

Description

Copies the block to the clipboard.

8.4.5.24 Function BlockDelete

Declaration:

```
void BlockDelete();
```

Description

Deletes the block. The block is copied to the clipboard

8.4.5.25 Function BlockEnd

Declaration:

```
void BlockEnd();
```

Description

Finishes marking of block. It is supposed that before calling BlockEnd(), the [BlockBegin](#)^[269] function is called and then the cursor is moved to the end of the block.

8.4.5.26 Function BlockFastCopy

Declaration:

```
void BlockFastCopy();
```

Description

Copies the block from the cursor position.

8.4.5.27 Function BlockMove

Declaration:

```
void BlockMove();
```

Description

Moves the block to the cursor position.

8.4.5.28 Function BlockOff

Declaration:

```
void BlockOff();
```

Description

Turns the block off..

8.4.5.29 Function BlockPaste

Declaration:

```
void BlockPaste();
```

Description

Pastes the block from the clipboard to the cursor position

8.4.5.30 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](#)^[268] call.

Example

```
unsigned long instance = LoadLibrary("EXTEND.DLL");  
long result = CallLibraryFunction(instance, "Initialize", 0, 1L);
```

8.4.5.31 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`.

8.4.5.32 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.

8.4.5.33 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, 0xFFFF, AS_DATA));
```

8.4.5.34 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](#)^[361] global variable to one of the following values:

EACCESS	Permission denied
EBADF	Bad file number
ENOSPC	No space left o

8.4.5.35 Function ClearAllBreaks

Declaration:

```
void ClearAllBreaks();
```

Description

Clears all breakpoints of all types.

8.4.5.36 Function ClearBreak

Declaration:

```
void ClearBreak(unsigned long addr);
```

Description

Clears the code breakpoint at the specified address.

8.4.5.37 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.

8.4.5.38 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.

8.4.5.39 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](#).

8.4.5.40 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](#), or [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

8.4.5.41 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.

8.4.5.42 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.

8.4.5.43 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`.

8.4.5.44 Function Cr

Declaration:

```
void Cr();
```

Description

Works like the **Enter** key.

8.4.5.45 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`^[359] 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 <code>amode</code>	Access permission
<code>S_IWRITE</code>	Permission to write
<code>S_IREAD</code>	Permission to read
<code>S_IREAD S_IWRITE</code>	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`^[361] global variable is set to one of the following:

<code>EACCES</code>	Permission denied
<code>ENOENT</code>	Path or file name not found
<code>EMFILE</code>	Too many open files

8.4.5.46 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](#)^[361] 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

8.4.5.47 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](#)^[359] 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](#)^[361] 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

8.4.5.48 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.

8.4.5.49 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](#)³³³ 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.

8.4.5.50 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
}
```

8.4.5.51 Function DelChar

Declaration:

```
void DelChar(int count=1);
```

Description

Deletes **count** characters beginning from the cursor position

8.4.5.52 Function DelLine

Declaration:

```
void DelLine(int count=1);
```

Description

Deletes the current line.

8.4.5.53 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](#)^[298] 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));
}
```

8.4.5.54 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](#)^[277] function.

8.4.5.55 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](#)^[333] function) and graphical coordinates (in pixels).

8.4.5.56 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](#)^[361] built-in variable.

8.4.5.57 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))

handle [creat](#)^[274], [open](#)^[314], [dup](#)^[278], or [dup2](#)^[278].

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](#)^[361] global variable is set to one of the following values:

EBADF	Bad file number
EMFILE	Too many open files

8.4.5.58 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](#)^[274], [open](#)^[314], [dup](#)^[278], or [dup2](#)^[278] call.

Returned value

dup2 returns 0 on successful completion, and -1 otherwise. In the event of error, the [errno](#)^[361] global variable is set to one of the following values:

EBADF Bad file number

EMFILE Too many open files

8.4.5.59 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](#)^[333] function and paints it with the brush selected by the [SelectBrush](#)^[333] 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.

8.4.5.60 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](#)^[361] global variable is set to

EBADF Bad file number

8.4.5.61 Function Eof

Declaration:

```
void Eof();
```

Description

Move the cursor to the file end.

8.4.5.62 Function Eol

Declaration:

```
void Eol();
```

Description

Move the cursor to the end of the current line.

8.4.5.63 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");
```

8.4.5.64 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\[\]](#)^[364] 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");
}

```

8.4.5.65 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:

<code>file_name[]</code>	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.
<code>include_dir[]</code>	The listing of directories, where the compiler will search for the #include-files. You can specify multiple directory names separated by semicolon.
<code>char defines[]</code>	The string with the definitions of preprocessor variables. Also, see the Script Files ^[178] dialog ^[178] .
<code>debug</code>	If not equal to 0, then the Script Source window ^[177] 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 \(#include\)](#) ^[230].

8.4.5.66 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.

8.4.5.67 Function ExitProgram

Declaration:

```
void ExitProgram();
```

Description

Exits the work session of <%CM%> in the same way as by closing its window.

8.4.5.68 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`.

8.4.5.69 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](#)^[267], [FloatExpr](#)^[267], [Expressions](#)^[210].

8.4.5.70 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`.

8.4.5.71 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.

8.4.5.72 Function fdopen

Declaration:

```
unsigned long fdopen(long handle, char type[]);
```

Description

Associates a stream with a file handle.

obtained from [creat](#)^[274], [dup](#)^[278], [dup2](#)^[278], or [open](#)^[314]. 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.
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+	

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 fseek or 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.

8.4.5.73 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.

8.4.5.74 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.

8.4.5.75 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.

8.4.5.76 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.

8.4.5.77 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, whichever 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.

8.4.5.78 Function FileChanged

Declaration:

```
int FileChanged();
```

Description

If the file is changed since the last save, it will return 1; 0 otherwise.

8.4.5.79 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](#)^[36] global variable is set to

EBADF Bad file number

8.4.5.80 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.

8.4.5.81 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](#)^[333] function; (x1, y1) are the coordinates of the upper left corner; (x2, y2) are the coordinates of the lower right corner.

8.4.5.82 Function findfirst

Declaration:

```
int findfirst(char path[], char ffbk[], 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](#)^[266] function.

The **ffbk** parameter specifies an internal data storage buffer for the function. Its size should be 48 bytes.

After findfirst access, the ffbk buffer contains information about the found file. The [_ff_attrib](#)^[264], [_ff_time](#)^[265], [_ff_date](#)^[264], [_ff_size](#)^[265] and [_ff_name](#)^[264] functions receive ffbk 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 ffbk[48];
int done = findfirst("c:\\data.*", ffbk, 0);
long total_size = 0;
while (@!done)
{
    total_size += _ff_size(ffbk);
    done = findnext(ffbk);
}
printf("Total size of the files %lu", total_size);
```

8.4.5.83 Function findnext

Declaration:

```
int findnext(char ffbk[]);
```

Description

Continues the search for files started by the [findfirst](#)^[286] 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](#)^[264], [_ff_time](#)^[265], [_ff_date](#)^[264], [_ff_size](#)^[265] and [_ff_name](#)^[264] 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](#)^[286].

8.4.5.84 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](#)^[316] 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.

8.4.5.85 Function FirstWord

Declaration:

```
void FirstWord();
```

Description

Moves the cursor to the first non-empty character in the line.

8.4.5.86 Function FloatExpr

Declaration:

```
float FloatExpr(char str[]);
```

Description

The same as [Expr](#)^[282], but the result is a floating-point number.

Also, see [AddrExpr](#)^[267], [Expr](#)^[282].

8.4.5.87 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.

8.4.5.88 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.

8.4.5.89 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`.

8.4.5.90 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 identifier in subsequent operations. The `mode` string used in calls to `fopen` is one of the following values:

Value	Description
<code>r</code>	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 `fseek` or `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.

8.4.5.91 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(" {[<");
```

8.4.5.92 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.

8.4.5.93 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](#)^[319].

Returned Value

`fprintf` returns the number of bytes that were output. In the event of error, it returns EOF.

8.4.5.94 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.

8.4.5.95 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.

8.4.5.96 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](#)^[339] 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.

8.4.5.97 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).

8.4.5.98 Function FreeLibrary

Declaration:

```
void FreeLibrary(unsigned long inst);
```

Description

De-allocates the specified DLL. HINSTANCE obtained by the [LoadLibrary](#)^[307] call is transferred as the parameter.

8.4.5.99 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](#)^[359] 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 `fseek` or `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`.

8.4.5.100 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`.

8.4.5.101 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](#)^[331].

`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](#)^[331] 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.

8.4.5.102 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 fromwhere. offset should be 0 or the value returned by [ftell](#)^[293]. fromwhere 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](#)^[309]. 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](#)^[361] global variable is set to one of the following values:

EBADF	Bad file pointer
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

8.4.5.103 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](#)^[293].

Returned Value

on success `ftell` returns the current file pointer position. It returns `-1L` on error and sets the [errno](#)^[361] global variable to a positive value. In the event of error return, the [errno](#)^[361] global variable is set to one of the following values:

<code>EBADF</code>	Bad file pointer
<code>ESPIPE</code>	Illegal seek on device

8.4.5.104 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.

8.4.5.105 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));
```

8.4.5.106 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.

8.4.5.107 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

8.4.5.108 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**.

8.4.5.109 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]);
```

8.4.5.110 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.

8.4.5.111 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.

8.4.5.112 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);
```

8.4.5.113 Function GetFileName

Declaration:

```
void GetFileName(char dest[]);
```

Description

Copies the name of the current [Edit](#) ¹⁸⁶ [window](#) ¹⁸⁶ to the `dest` array.

8.4.5.114 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
---------	-------------------------

8.4.5.115 Function GetLine

Declaration:

```
void GetLine(char dest[]);
```

Description

Copies the whole current line to the **dest** array.

8.4.5.116 Function GetMark

Declaration:

```
void GetMark(int number);
```

Description

Retrieves the bookmark with the **number** number (1...10).

8.4.5.117 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);
```

8.4.5.118 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);
```

8.4.5.119 Function gettimeofday

Declaration:

```
void gettimeofday(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 **gettimeofday** function uses the system timer, you may expect a time error of about 52 milliseconds.

Example

```
int time[4];  
while (1)  
{  
    gettimeofday(time);  
    printf("Time: %d:%d:%d.%d", time[3], time[2], time[1], time[0]);  
}
```

8.4.5.120 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](#)^[283] or [ferror](#)^[284] should be used to detect the end-of-file or error.

8.4.5.121 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](#)^[316], and [FindWindow](#)^[287] functions.

This function is useful, when it is necessary to draw in the [User](#)^[180] [window](#)^[180] regardless of its size.

Returned value

The height of the specified window user area in pixels.

8.4.5.122 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](#)^[316], and [FindWindow](#)^[287] functions.

This function is useful, when it is necessary to draw in the [User](#)^[180] [window](#)^[180] regardless of its size.

Returned value

The height of the specified window user area in pixels.

8.4.5.123 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));
```

8.4.5.124 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));
```

8.4.5.125 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](#)^[361] and [CurLine](#)^[361] built-in variables.

8.4.5.126 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](#)^[341] function.

Note. The screen is not updated automatically after the HStep call. To organize automatic updates, use the [RedrawScreen](#)^[328] function at the appropriate moment.

8.4.5.127 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);
```

8.4.5.128 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);
```

8.4.5.129 Function Inspect

Declaration:

```
unsigned int Inspect(char name[]);
```

Description

Opens the **Inspector window** for the specified name (the **name** parameter).

8.4.5.130 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.

8.4.5.131 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.

8.4.5.132 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.

8.4.5.133 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.

8.4.5.134 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.

8.4.5.135 Function `isctrl`

Declaration:

```
int isctrl(unsigned char c);
```

Description

The `isctrl` function checks, if parameter `c` is a control character (from 0x00 to 0x1F, or 0x7F).

Returned character

The `isctrl` function will return a non-zero value, if `c` is a control character or digit, otherwise it will return 0.

8.4.5.136 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.

8.4.5.137 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.

8.4.5.138 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.

8.4.5.139 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.

8.4.5.140 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.

8.4.5.141 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.

8.4.5.142 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.

8.4.5.143 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.

8.4.5.144 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.

8.4.5.145 Function LastChar

Declaration:

```
int LastChar(unsigned long handle);
```

Description

Returns the code of the button pressed at the last call of [wgetchar](#)^[356] or the hexadecimal number entered at the last call of [wgethex](#)^[357].

8.4.5.146 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](#)^[358] 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](#)^[180] [window](#)^[180]) - You clicked a mouse button, when the mouse cursor was in the window. Information on the click can be obtained by calling the [LastEventIntx](#)^[308] 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](#)^[268] 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](#)^[266]) 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.

8.4.5.147 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](#)^[305], [WaitWindowEvent](#)^[356].

8.4.5.148 Function LastString

Declaration:

```
int LastString(unsigned long handle, char s[]);
```

Description

Copies the string entered at the last call of [wgettext](#)^[357] to the string (the **s** parameter).

8.4.5.149 Function LineTo

Declaration:

```
void LineTo(unsigned long handle, int x, int y);
```

Description

Draws a line from the point set up by the [MoveTo](#)^[314] or `LineTo` function to the point with coordinates (x, y). The line is drawn with the pen selected with the [SelectPen](#)^[333] 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
```

8.4.5.150 Function LoadDesktop

Declaration:

```
void LoadDesktop(char file_name[]);
```

Description

Downloads the specified screen configuration file (see [Configuration Files](#)^[52]).

8.4.5.151 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](#)^[36] built-in variable.

8.4.5.152 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](#)^[270].

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");
```

8.4.5.153 Function LoadOptions

Declaration:

```
void LoadOptions(char file_name[]);
```

Description

Downloads the specified option file (see [Configuration Files](#)^[52]).

8.4.5.154 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.

8.4.5.157 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`.

8.4.5.158 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`.

8.4.5.159 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`^[361] 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.

8.4.5.160 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).

8.4.5.161 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](#)^[313]

8.4.5.162 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 **src** 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.

8.4.5.163 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

8.4.5.164 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`.

8.4.5.165 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 `src` 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.

8.4.5.166 Function `memcmp`

Declaration:

```
int memicmp(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` 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 `memcmp` function returns the positive, negative or zero value, depending on the result of comparing the first `n` bytes of objects `s1` and `s2`.

8.4.5.167 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 `src` 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.

8.4.5.168 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.

8.4.5.169 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:

[Formatted Input-Output Functions](#) ^[250]

[Alphabetical List of Script Language Built-in Functions and Variables](#) ^[256]

8.4.5.170 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](#)^[319]).

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:

[Formatted Input-Output Functions](#)^[250]

[Alphabetical List of Script Language Built-in Functions and Variables](#)^[256]

8.4.5.171 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);
```

8.4.5.172 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](#)^[361] global variable contains one of the following values:

EACCES Permission denied

ENOENT No such file or directory

8.4.5.173 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](#)^[306] 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);
```

8.4.5.174 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](#)^[316], and [FindWindow](#)^[287] 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.

8.4.5.175 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.

8.4.5.176 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](#)^[359] 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](#)^[361] 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

8.4.5.177 Function OpenEditorWindow

Declaration:

```
unsigned long OpenEditorWindow(char file_name[]);
```

Description

Opens the [Source](#)^[186] [window](#)^[186] 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.

8.4.5.178 Function OpenStreamWindow

Declaration:

```
unsigned long OpenStreamWindow(char title[]);
```

Description

Opens the [I/O Stream](#)^[180] [window](#)^[180] and sets up its title (the [title](#) parameter).

You can do the same with the [OpenWindow](#)^[316] 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](#)^[339] 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");
```

8.4.5.179 Function OpenUserWindow

Declaration:

```
unsigned long OpenUserWindow(char title[]);
```

Description

Opens the [User](#)^[180] [window](#)^[180] and specifies its title (parameter [title](#)).

This can also be done with the [OpenWindow](#)^[316] 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](#)^[339] 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");
```

8.4.5.180 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](#)^[52] [menu](#)^[52] 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](#)^[339], [SetWindowSizeT](#)^[339], or [MoveWindow](#)^[314] 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.

8.4.5.181 Function outport

Declaration:

```
void outport(unsigned int port_num, unsigned int value);
```

Description

Writes a value (word) to the specified parallel port.

8.4.5.182 Function outportb

Declaration:

```
void outportb(unsigned int port_num, unsigned char value);
```

Description

Write a value (byte) to the specified parallel port.

8.4.5.183 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.

8.4.5.184 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.

8.4.5.185 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**.

8.4.5.186 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**.

8.4.5.187 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);
```

8.4.5.188 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.

8.4.5.189 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.

8.4.5.190 Function printf

Declaration:

```
void printf(char format[], ... );
```

Description

The printf function displays the values of transferred parameters in the [Console](#) ^[104] 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](#) ^[358] 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](#) ^[323]

[Format Specifiers](#) ^[322]

[Flag Characters](#) ^[320]

[Width Specifiers](#) ^[325]

[Precision Specifiers](#) ^[323]

[Input-size Modifiers](#) ^[323]

[Type Characters](#) ^[319]

[Format Specifier Conventions](#) ^[320]

Returned Value

Example

```
printf("Counter = %d\n"
      "Value  = %08lX",
      Counter, Value);
```

8.4.5.190.1 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](#) ^[322].

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.

8.4.5.190.2 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 ^[322] .
Note. Plus (+) takes precedence over blank () if both are given	

8.4.5.190.3 printf Format Specifier Conventions

Certain conventions accompany some of the [printf format specifiers](#)^[322] for the following

conversions:

[- %e or %E](#) ^[321]

[- %f](#) ^[321]

[- %g or %G](#) ^[321]

[- %x or %X](#) ^[321]

Note. Infinite floating-point numbers are printed as +INF and -INF. An IEEE Not-a-Number is printed as +NAN or -NAN.

8.4.5.190.3.1 %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.

8.4.5.190.3.2 %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).

8.4.5.190.3.3 %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 G if G is the conversion character.

8.4.5.190.3.4 %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.

8.4.5.190.3.5 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 i u	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.

8.4.5.190.4 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) ^[320] Output justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes.
[width]	(Optional)	Width specifier ^[325] Minimum number of characters to print, padding with blanks or zeros.
	(Optional)	Precision specifier ^[323] 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](#)³¹⁹.

8.4.5.190.5 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.

8.4.5.190.6 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.

8.4.5.190.7 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.

8.4.5.190.8 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.

8.4.5.191 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](#)^[331] 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](#)^[331].

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](#)^[331] 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.

8.4.5.192 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](#) 289

[fputc](#) 290

[fputs](#) 290

[fwrite](#) 294

[getc](#) 294

[printf](#) 319

[putw](#) 326

8.4.5.193 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.

8.4.5.194 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](#)^[284] to detect errors with `putw`.

8.4.5.195 Function `rand`

Declaration:

```
int rand();
```

Returns a pseudorandom number in the range from 0 to 32767.

8.4.5.196 Function `random`

Declaration:

```
int random(int num);
```

Description

-1.

8.4.5.197 Function `randomize`

Declaration:

```
void randomize();
```

Description

Initializes a random number generator by a random number.

8.4.5.198 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](#)^[274], [open](#)^[314], [dup](#)^[278], or [dup2](#)^[278] 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](#)^[361] global variable to one of the following two values:

EACCES	Permission denied
EBADF	Bad file number

8.4.5.199 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](#)^[333] function and paints it using the brush selected with the [SelectBrush](#)^[333] function; (x1, y1) are the coordinates of the upper left corner; (x2, y2) are the coordinates of the lower right corner.

8.4.5.200 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();
```

8.4.5.201 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](#).^[51]

8.4.5.202 Function RemoveButtons

Declaration:

```
void RemoveButtons(unsigned long handle);
```

Description

Removes all buttons from the window that were added by the [AddButton](#)^[266] 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.

8.4.5.203 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

8.4.5.204 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](#)^[293](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.

8.4.5.205 Function Right

Declaration:

```
void Right(int count=1);
```

Description

Move the cursor positions right. The same result can be achieved by incrementing the [CurCol](#)^[361] built-in variable.

8.4.5.206 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](#)^[361] global variable contains one of the following values:

EACCES Permission denied

ENOENT Path or file function not found

8.4.5.207 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);
```

8.4.5.208 Function SaveDesktop

Declaration:

```
void SaveDesktop(char file_name[]);
```

Description

Saves the screen configuration in the specified file (see [Configuration Files](#))⁵²

8.4.5.209 Function SaveFile

Declaration:

```
int SaveFile();
```

Description

Saves the file from the current [window](#)¹⁸⁶.

8.4.5.210 Function SaveOptions

Declaration:

```
void SaveOptions(char file_name[]);
```

Description

Saves the options in the specified file (see [Configuration Files](#))⁵²

8.4.5.211 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](#)³³¹ 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](#)³³¹ 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".
```

8.4.5.212 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](#)^[361], [WholeWords](#)^[364] and [RegularExpressions](#)^[363] 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](#)^[362] variable. This is because the found string may not be the same as the search argument

8.4.5.213 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](#)^[288]).

searchpath returns TRUE on success, otherwise it returns FALSE.

8.4.5.214 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 occurrence of text is replaced. If `replace_all` is 0, then only the first occurrence will be replaced, otherwise, all occurrences.

`SearchReplace` returns the number of replaces

8.4.5.215 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.

8.4.5.216 Function SelectFont

Declaration:

```
void SelectFont(unsigned long handle, char name[], int height);
```

Description

Selects a font for text output. As opposed to the [SetWindowFont](#)^[338] function, this font can be proportional. It is used for displaying text with the [DisplayTextF](#)^[277] function anywhere in the window.

`name` is the line with the font name; `height` specifies the font height.

8.4.5.217 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

8.4.5.218 Function SetBkColor

Declaration:

```
void SetBkColor(unsigned long handle, unsigned long color);
```

Description

Sets up the window background [color](#).

8.4.5.219 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](#)^[277], [DisplayTextF](#)^[277]) 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.

8.4.5.220 Function SetBreak

Declaration:

```
void SetBreak(unsigned long addr);
```

Description

Sets up the code breakpoint at the specified address

8.4.5.221 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.

8.4.5.222 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);
```

8.4.5.223 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.

8.4.5.224 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.

8.4.5.225 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);
```

8.4.5.226 Function SetFileName

Declaration:

```
void SetFileName(char name[]);
```

Description

Sets the file name for the current [Source](#) 

8.4.5.227 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](#)³⁶⁷ global variable is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
EINVFNC	Invalid function number

8.4.5.228 Function SetMark

Declaration:

```
void SetMark(int number);
```

Description

Sets the bookmark with the numberNumber shall be within 1...10.

8.4.5.229 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.

8.4.5.230 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);
```

8.4.5.231 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](#)^[36] global variable to

`EINVAL` Invalid argument

8.4.5.232 Function **SetPixel**

Declaration:

```
void SetPixel(unsigned long handle, int x, int y, unsigned long color);
```

Draws one point of the specified [color](#)^[80] in the specified place.

8.4.5.233 Function **SetTextColor**

Declaration:

```
void SetTextColor(unsigned long handle, unsigned long color);
```

Description

Sets up [color](#)^[80] of the text printed out by the [wprintf](#)^[358] function, or displayed by the [DisplayText](#) and [DisplayTextF](#)^[277] 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");
```

8.4.5.234 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.

8.4.5.235 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](#)^[352] 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);
```

8.4.5.236 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](#)^[287] 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](#)^[180]. To select the font, use the [SelectFont](#)^[333] function.

Example

```
unsigned long handle = OpenWindow(WIN_DUMP);  
  
SetWindowFont(handle, "Courier New", 12);
```

8.4.5.237 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](#)^[316], and [FindWindow](#)^[287] 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.

8.4.5.238 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](#)^[316], and [FindWindow](#)^[287] functions. w is the number of text characters in the line; h is the number of lines in the window.

8.4.5.239 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);
```

8.4.5.240 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`.

8.4.5.241 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.

8.4.5.242 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.

8.4.5.243 Function srand

Declaration:

```
void srand(unsigned int seed);
```

Description

Initializes a random number generator by a specified number.

8.4.5.244 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 CPL2-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](#)^[331] 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](#)^[331].

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](#)^[331] for a discussion on possible causes.

Returned Value

8.4.5.245 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](#)^[328] function at the appropriate moment.

8.4.5.246 Function Stop

Declaration:

void Stop();

Description

Stops the program under execution.

8.4.5.247 Function stpcpy

Declaration:

```
int stpcpy(char dest[], char src[], int dest_index=0, int src_index=0);
```

Description

The stpcpy function copies the 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

8.4.5.248 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.

8.4.5.249 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.

8.4.5.250 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

8.4.5.251 Function strcmpi

Declaration:

```
int strcmpi(char s1[], char s2[], int s1_index=0, int s2_index=0);
```

The same as [strcmp](#)^[343]

8.4.5.252 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.

8.4.5.253 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).

8.4.5.254 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 strcmp 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

8.4.5.255 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.

8.4.5.256 Function strlwr

Declaration:

```
void strlwr(char s[], int index=0);
```

Description

The strlwr function converts line s to the lower case.

Returned value

None.

8.4.5.257 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 src 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.

8.4.5.258 Function strcmp

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

8.4.5.259 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

8.4.5.260 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 `src` 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.

8.4.5.261 Function strnicmp

Declaration:

```
int strnicmp(char dest[], char src[], int n, int dest_index=0, int src_index=0);
```

The same as [strncmpi](#)^[345].

8.4.5.262 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.

8.4.5.263 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.

8.4.5.264 Function strchr

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.

8.4.5.265 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

8.4.5.266 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.

8.4.5.267 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).

8.4.5.268 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.

8.4.5.269 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.

8.4.5.270 Function strtoul

Declaration:

```
unsigned long strtoul(char s[], int endptr[], int radix, int index=0);
```

Description

The strtoul function is the same as [strtol](#)³⁴⁸, except that it returns the unsigned long integer.

8.4.5.271 Functionstrupr

Declaration:

```
voidstrupr(char s[], int index=0);
```

Description

Thestrupr function converts line s to the upper case.

Returned value

None.

8.4.5.272 Functiontan

Declaration:

```
floattan(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](#).

8.4.5.273 Functiontanh

Declaration:

```
floattanh(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](#).

8.4.5.274 Functiontell

Declaration:

```
longtell(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](#)³⁶¹ global variable is set to

8.4.5.275 Function TerminateAllScripts

Declaration:

```
void TerminateAllScripts();
```

Description

Stops execution of all script files (except the script called by this function).

8.4.5.276 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.

8.4.5.277 Function Text

Declaration:

```
void Text(char text[]);
```

Description

text text from the cursor position, as if it were typed from the keyboard.

8.4.5.278 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

8.4.5.279 Function Tof

Declaration:

```
void Tof();
```

Description

Move the cursor to the top of the file (position (1:1)).

8.4.5.280 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.

8.4.5.281 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.

8.4.5.282 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).

8.4.5.283 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](#)^[36] global variable to one of the following values:

EACCES Permission denied
 ENOENT Path or file name not found

8.4.5.284 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](#)^[359]. 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.

8.4.5.285 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](#)^[361] built-in variable.

8.4.5.286 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](#)^[338] function call with the UM_ONREQUEST parameter

8.4.5.287 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
}
}
```

8.4.5.288 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
}
}
```

8.4.5.289 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
}

```

8.4.5.290 Function WaitGetMessage

Declaration:

```
void WaitGetMessage(int id);
```

Description

[WaitSendMessage](#)^[355]

8.4.5.291 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[362]      the memory address
LastMemAccAddrSpace[362] the type of address space
LastMemAccLen[362]      the amount of bytes
LastMemAccType[362]     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
}

```

8.4.5.292 Function WaitSendMessage

Declaration:

```
void WaitSendMessage(int id, unsigned int int_data, unsigned long long_data);
```

Description

The WaitSendMessage and [WaitGetMessage](#)^[354] functions provide a mechanism for message exchange between two copies of the CPI2-B1 program (or other Phytion 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](#)^[363] built-in variable and long_data is copied to [LastMessageLong](#)^[363].

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;
}
```

8.4.5.293 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.

8.4.5.294 Function WaitWindowEvent

Declaration:

```
void WaitWindowEvent(unsigned long handle);
```

Description

Allows to organize interaction between user and the [User window](#)^[180] and the [_____](#)^[180]. 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](#)^[305]

[LastEventInt{1...4}](#)^[306]

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

    }
}
```

8.4.5.295 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](#)^[305] 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
```

8.4.5.296 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](#)^[305] function.

The entered characters are automatically displayed in the window. The Enter key moves the window cursor to the beginning of the new line.

8.4.5.297 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](#)^[306] function.

The entered characters are automatically displayed in the window.

8.4.5.298 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
```

8.4.5.299 Function WordLeft

Declaration:

```
void WordLeft();
```

Description

Moves the cursor to the next word (on the right).

8.4.5.300 Function WordRight

Declaration:

```
void WordRight();
```

Description

Moves the cursor to the previous word (on the left).

8.4.5.301 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");
```

8.4.5.302 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 [creat](#)^[274], [open](#)^[314], [dup](#)^[278], or [dup2](#)^[278] 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](#)^[361] global variable to one of the following values:

EACCES Permission denied

EBADF Bad file number

8.4.5.303 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](#)^[361] global variable to

EACCES Locking violation

8.4.5.304 Variable _fmode

Declaration:

```
extern int _fmode;
```

This is the file operation mode (text or binary).

8.4.5.305 Variable AppName

Declaration:

```
extern char AppName[];
```

This is the program name, i.e. the string of "CM-ARM".

Available only for reading.

8.4.5.306 Variable BlockCol1

Declaration:

```
int BlockCol1;
```

This is the number of the left column of block in the current [window](#)^[186]. BlockCol1 is zero for the line blocks. If no block is marked, BlockCol1 will also be zero.

Also, see [Text Editor Functions](#)^[250].

8.4.5.307 Variable BlockCol2

Declaration:

```
int BlockCol2;
```

This is the number of the right column of block in the current [Source](#)^[186]. BlockCol2 is zero for the line blocks. If no block is marked, BlockCol2 will also be zero.

Also, see [Text Editor Functions](#)^[250].

8.4.5.308 Variable BlockLine1

Declaration:

```
int BlockLine1;
```

This is the number of the upper line of block in the current [Source](#)^[186].

If no block is marked, BlockCol2 will be zero.

Also, see [Text Editor Functions](#)^[250].

8.4.5.309 Variable BlockLine2

Declaration:

```
int BlockLine2;
```

This is the number of the lower line of block in the current [Source](#)^[186].

If no block is marked, BlockCol2 will be zero.

Also, see [Text Editor Functions](#)^[250].

8.4.5.310 Variable BlockStatus

Declaration:

```
int BlockStatus;
```

This is the type of block in the current [Source](#)^[186]. 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](#)^[250].

8.4.5.311 Variable CaseSensitive

Declaration:

int CaseSensitive;

[Source](#)^[186]

Also, see [Text Editor Functions](#).

8.4.5.312 Variable CurCol

Declaration:

int CurCol;

This is the number of the current column (the column the cursor is in) in the current [Source](#)^[186]. 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](#)^[300], [Up](#)^[352], [Down](#)^[278], [Left](#)^[307], [Right](#)^[329], [Tof](#)^[350], [Eof](#)^[279], [Eol](#)^[280].

8.4.5.313 Variable CurLine

Declaration:

int CurLine;

[Source](#)^[186]. Lines are numbered with 1.

Assigning a value to CurLine changes the cursor position. Also, see functions [GotoXY](#)^[300], [Up](#)^[352], [Down](#)^[278], [Right](#)^[329], [Tof](#)^[350], [Eof](#)^[279], [Eol](#)^[280].

8.4.5.314 Variable DesktopName

Declaration:

extern char DesktopName[];

This string is the name of the current screen configuration file (see [Configuration Files](#)^[52]).

Available only for reading.

8.4.5.315 Variable errno

Declaration:

extern int errno;

This is the error code set up by some built-in functions such as [read](#)^[327].

8.4.5.316 Variable InsertMode

Declaration:

```
int InsertMode;
```

This is the insert mode for the current [Source](#) ^[186]. Assigning a value to InsertMode toggles the insert mode for the window.

Also, see [Text Editor Functions](#) ^[250].

8.4.5.317 Variable LastFoundString

Declaration:

```
char LastFoundString[];
```

This is the string with the text that was last found in the current [Source](#) ^[186]. 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](#) ^[250].

8.4.5.318 Variable LastMemAccAddr

Declaration:

```
extern unsigned long LastMemAccAddr;
```

This is the microcontroller memory address accessed at the last return from the [WaitMemoryAccess](#) ^[354] function.

8.4.5.319 Variable LastMemAccAddrSpace

Declaration:

```
extern unsigned int LastMemAccAddrSpace;
```

This is the type of microcontroller address space accessed at the last return from the [WaitMemoryAccess](#) ^[354] function

8.4.5.320 Variable LastMemAccLen

Declaration:

```
extern int LastMemAccLen;
```

[WaitMemoryAccess](#) ^[354] function.

8.4.5.321 Variable LastMemAccType

Declaration:

extern int LastMemAccType;

This is the microcontroller memory access type that caused a return from the [WaitMemoryAccess](#)^[354] function. For example, MA_READ, MA_WRITE or a combination of them.

8.4.5.322 Variable LastMessageInt

Declaration:

unsigned int LastMessageInt;

LastMessageInt keeps the 16-bit parameter received by the [WaitGetMessage](#) function.

8.4.5.323 Variable LastMessageLong

Declaration:

unsigned long LastMessageLong;

LastMessageLong keeps the 32-bit parameter received by the [WaitGetMessage](#)^[354] function.

8.4.5.324 Variable MainWindowHandle

Declaration:

extern unsigned long MainWindowHandle;

This is HWND of the main window of CPI2-B1. It is only for experienced programmers.

8.4.5.325 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.

8.4.5.326 Variable RegularExpressions

Declaration:

int RegularExpressions;

Sets up the use of regular expressions for the operation of search in the current [Source](#)^[186].

Also, see [Text Editor Functions](#)^[250].

8.4.5.327 Variable SelectedString

Declaration:

```
extern char SelectedString[];
```

This is the string selected from the menu at the last call of the built-in [ExecMenu](#)^[280] function.

8.4.5.328 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.

8.4.5.329 Variable WholeWords

Declaration:

```
int WholeWords;
```

Sets up the whole words option for the operation of search in the current [Source](#)^[186] [window](#)^[186].

Also, see [Text Editor Functions](#)^[250].

8.4.5.330 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.

8.4.5.331 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.

8.4.5.332 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.

8.5 ACI Fuctions and Structures

This section contains detailed descriptions of ACI functions and structures.

8.5.1 ACI Fuctions

This sections contains **alphabetical list** of all ACI functions.

8.5.1.1 ACI_AllProgOptionsDefault

[ACI_FUNC ACI_AllProgOptionsDefault\(\);](#)

Description

This function sets default device-specific options and parameters specified in the [Device and Algorithm Parameters Editor](#)^[93] window. These default parameter sets vary. They are defined by the device manufacturers in the device data sheets.

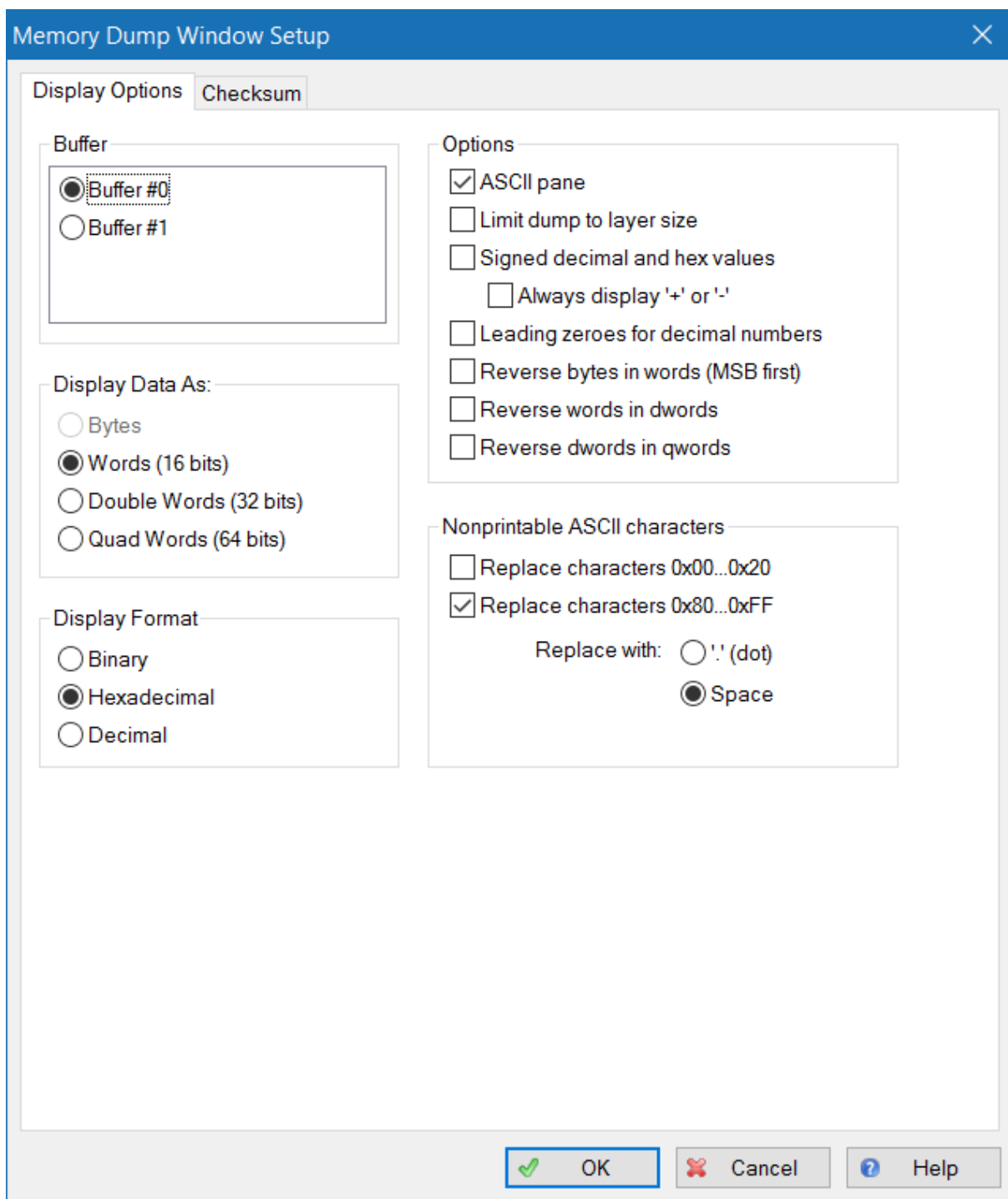
Note! This function **does not physically restore the default settings into the device being programmed**. It actually writes to some virtual memory locations in the host PC's RAM, associated with physical locations in the target device's memory and registers. In order to complete programming the device 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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes.

8.5.1.2 ACI_BuffersDialog

[ACI_BuffersDialog\(\);](#)

Description

This macro opens the [Memory Dump Window Setup](#)^[98] 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](#)^[98] dialog will appear on the computer screen to allow the buffer setup. See the dialog example below.



8.5.1.3 ACI_ConnectionStatus

`ACI_FUNC ACI_ConnectionStatus();`

Description

Get a current connection status. If the connection is active, i.e. the programmer or multiple programmers set by the [ACI_SetConnection](#)^[376] function, is operable and responds requests sent via ACI, the return code is ACI_ERR_SUCCESS. If, for any reason, the connection was broken the return code is ACI_ERR_NOT_CONNECTED.

See also: [ACI_SetConnection](#)^[376], [ACI_GetConnection](#)^[369].

8.5.1.4 ACI_CreateBuffer

[ACI_FUNC ACI_CreateBuffer](#)([ACI_Buffer_Params](#)^[378] * params);

Description

This function creates a buffer with the parameters specified by the [ACI_Buffer_Params](#)^[378] 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](#)^[378] structure description.

8.5.1.5 ACI_ErrorString

[ACI_FUNC ACI_ErrorString](#)([ACI_ErrorString_Params](#)^[381] * 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.

8.5.1.6 ACI_ExecFunction

[ACI_FUNC ACI_ExecFunction](#)([ACI_Function_Params](#)^[383] * params);

Description

This function launches one of the programming operation (**Read**, **Erase**, **Verify**, etc.) specified by the [ACI_Function_Params](#)^[383]. 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](#)^[367] from the [ACI_StartFunction](#)^[378] that returns control immediately after it was called.

8.5.1.7 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](#)^[371] function. In this case you should manually close the ChipProg-02 program via the Windows Task Manager.

8.5.1.8 ACI_FileLoad

```
ACI_FUNC ACI_FileLoad(ACI\_File\_Params[381] * 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.

8.5.1.9 ACI_FileSave

```
ACI_FUNC ACI_FileSave(ACI\_File\_Params[381] * params);
```

Description

This function saves a specified file from a specified buffer's layer. The ChipProg-02 software enables [saving files](#)^[104] in all popular formats: HEX, Binary, etc..

8.5.1.10 ACI_FillLayer

```
ACI_FUNC ACI_FillLayer(ACI\_Memory\_Params[388] * 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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes.

8.5.1.11 ACI_GangStart

```
ACI_FUNC ACI_GangStart(ACI_GangStart_Params[384] * params);
```

Description

This function is used to control [multiple device programmers](#)^[197] 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](#)^[371] function. For controlling a single CPI2-B1 device programmer use [ACI_StartFunction](#)^[378] or [ACI_ExecFunction](#)^[367].

The **ACI_GangStart** function launches [Auto Programming](#)^[108] on multiple CPI2-B1 device programmers for the programming socket specified in the **SiteNumber** parameter of the [ACI_PStatus_Params](#)^[395] structure. The function returns control immediately. In order to detect the ending time of the **ACI_GangStart** execution, use the [ACI_GetStatus](#)^[371] function.

8.5.1.12 ACI_GangTerminateFunction

```
ACI_FUNC ACI_GangTerminateFunction(ACI_GangTerminate_Params[385] * params);
```

Description

This function, similar to the [ACI_TerminateFunction](#)^[378] 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](#)^[197] (see the **/gang** parameter among other [Command line options](#)^[120] for the [ACI_Launch](#)^[371] function). In order to terminate an operation for a running single-site CPI2-B1 programmer use the [ACI_TerminateFunction](#)^[378].

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.

8.5.1.13 ACI_GetConnection

```
ACI_FUNC ACI_GetConnection(ACI_Connection_Params[381] * 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](#)^[371] function, then executing the [ACI_GetConnection](#)^[369] function returns a current [ConnectionId](#)^[381] parameter as a part of the [ACI_Launch_Params](#)^[385] structure.

See also [ACI_SetConnection](#)^[376].

8.5.1.14 ACI_GetDevice

```
ACI_FUNC ACI_GetDevice(ACI_Device_Params[381] * 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: M25P32VME, Micron; MC9S08DN60AMLC, NXP, etc.).

8.5.1.15 ACI_GetLayer

[ACI_FUNC ACI_GetLayer](#)([ACI_Layer_Params](#)^[386] * params);

Description

This function gets the parameters of a specified memory buffer and buffer's layer.

See also the [ACI_Layer_Params](#)^[386] structure description.

8.5.1.16 ACI_GetProgOption

[ACI_FUNC ACI_GetProgOption](#)([ACI_ProgOption_Params](#)^[389] * 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.

Device and Algorithm Parameters Editor

Edit	MinValue	MaxValue	DefaultValue	AllDefault	
Name		Value		Description	
Device Parameters					
...	Fuse Bits	...			Fuses
...	Lock bits	...			Lock bits
...	Calibration Byte	00h			Calibration value for the internal RC Oscillator
Algorithm Parameters					
...	Algorithm	"In-System Programming"			Programming algorithm
...	Oscillator Frequency	2500 kHz			Oscillator frequency
...	Delay after Vcc is On	120 ms			Delay after Vcc is On
...	Programming Mode	...			Programming Mode
...	Vcc	5.00 V			Power supply voltage

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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes. Then you can read the execute the [ACI_GetProgOption](#) function.

See also the [ACI ProgOption_Params](#)^[389] structure description.

8.5.1.17 ACI_GetProgrammingParams

[ACI_FUNC ACI_GetProgrammingParams](#)([ACI_Programming_Params](#)^[393] * params);

Description

This function gets current programming parameters specified in the tab [Option](#)^[109] of the [Program Manager](#)^[109] window (memory buffer configurations, programming options, test of the device insertion, etc.).

See the [ACI_Programming_Params](#)^[393] structure description.

8.5.1.18 ACI_GetStatus

[ACI_FUNC ACI_GetStatus](#)([ACI_PStatus_Params](#)^[395] * params);

Description

This function gets the programmer status that includes:

- 1) The status of the programming operation initiated by the [ACI_StartFunction](#)^[378] 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](#)^[109] of the [Program Manager](#)^[109] window).

8.5.1.19 ACI_Launch

[ACI_FUNC ACI_Launch](#)([ACI_Launch_Params](#)^[385] * params);

Description

This function launches the ChipProg-02 software. Optionally this ACI function can launch the programmer with a specified [Command line options](#)^[120] and load the file that will [configure](#)^[52] the CPI2-B1 hardware.

Note! This ACI function must always be called before any other ACI function !

8.5.1.20 ACI_LoadConfigFile

[ACI_FUNC ACI_LoadConfigFile](#)([ACI_Config_Params](#)^[380] * 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](#)^[108] commands. Then it automatically restores these parameters when the user changes the device type.

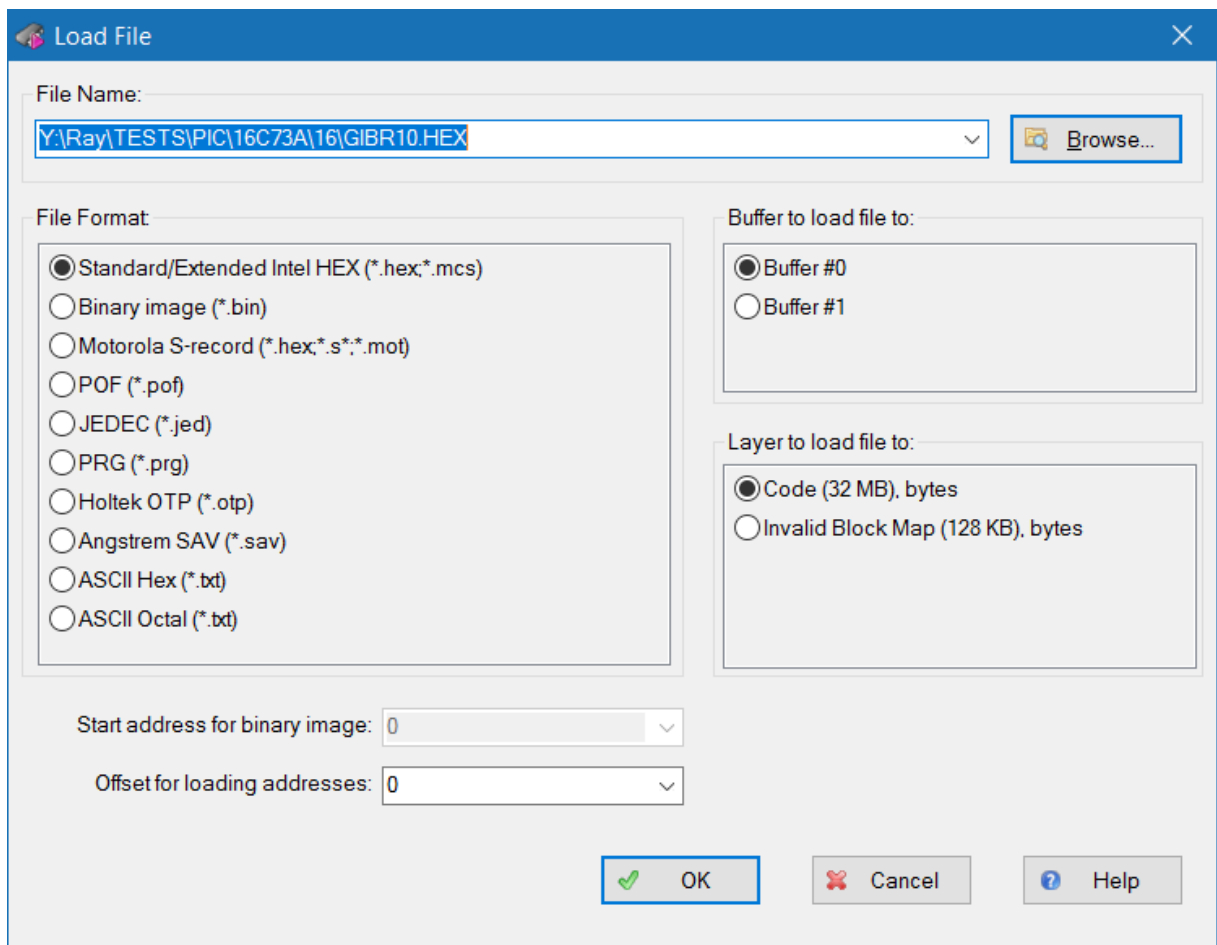
See also: [ACI_SetProgrammingParams](#)^[377], [ACI_SetProgOption](#)^[376], [ACI_GetProgrammingParams](#)^[371], [ACI_GetProgOption](#)^[370], [ACI_SaveConfigFile](#)^[374]

8.5.1.21 ACI_LoadFileDialog

[ACI_LoadFileDialog\(\)](#);

Description

This macro opens the [Load File](#)^[102] dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the [Load File](#)^[102] dialog will appear on the computer screen. See the dialog example below.



8.5.1.22 ACI_LoadProject

`ACI_FUNC ACI_LoadProject(ACI_Project_Params395 * 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.

8.5.1.23 ACI_ReadLayer

```
ACI_FUNC ACI_ReadLayer(ACI\_Memory\_Params[388] * 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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes.

8.5.1.24 ACI_ReallocBuffer

```
ACI_FUNC ACI_ReallocBuffer(ACI\_Buffer\_Params[378] * params);
```

Description

This function changes the size of the layer #0 in the memory buffer specified in the [ACI_Buffer_Params](#)^[378] structure.

See also the [ACI_Buffer_Params](#)^[378] structure description.

8.5.1.25 ACI_SaveConfigFile

```
ACI_FUNC ACI_SaveConfigFile(ACI\_Config\_Params[380] * params);
```

Description

This function saves the CPI2-B1 options specified in the tab [Option](#)^[109] of the [Program Manager](#)^[103] 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](#)^[108] commands and then automatically restores these parameters when the user changes the device type.

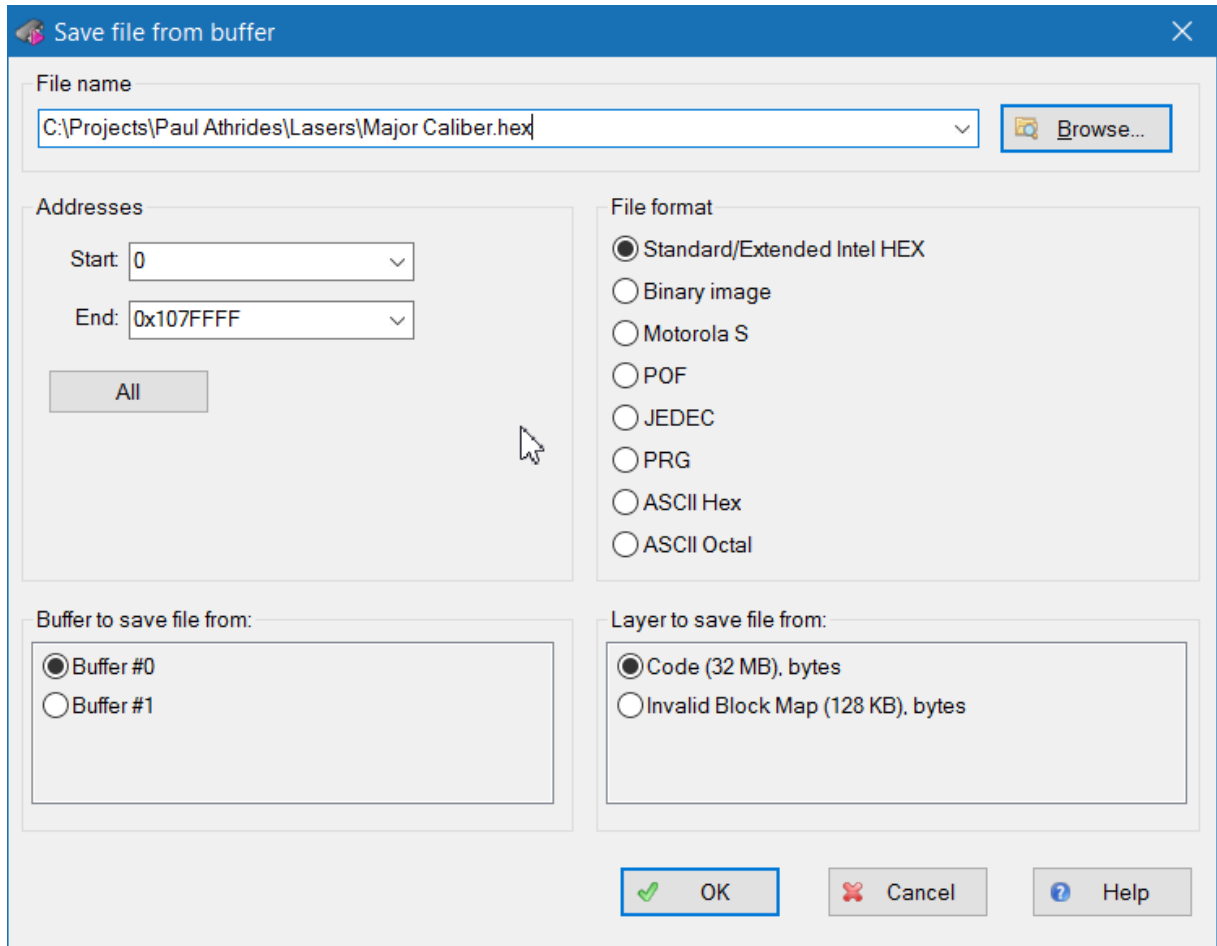
. : [ACI_SetProgrammingParams](#)^[377], [ACI_SetProgOption](#)^[376],
[ACI_GetProgrammingParams](#)^[371], [ACI_GetProgOption](#)^[370], [ACI_LoadConfigFile](#)^[371]

8.5.1.26 ACI_SaveFileDialog

```
ACI_SaveFileDialog();
```

Description

This macro sends a command that opens the [Save File](#)^[104] dialog. The dialog will be visible irrespective of the ChipProg-02 main window status; the main window can remain closed but the [Save File](#)^[104] dialog will appear on the computer screen. See the dialog example below.



8.5.1.27 ACI_SelectDeviceDialog

[ACI_SelectDeviceDialog\(\)](#);

Description

This macro sends a command that opens the [Select Device](#)^[58] 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](#)^[58] dialog will appear on the computer screen.

8.5.1.28 ACI_SerializationDialog

[ACI_SerializationDialog\(\)](#);

Description

This macro sends a command that opens the [Serialization, Checksum, and Log Dialog](#)^[63].

8.5.1.29 ACI_SetConnection

[ACI_FUNC ACI_SetConnection](#)([ACI_Connection_Params](#)^[381] * 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](#)^[371] function. Each connection gets its own unique identifier. Executing of the [ACI Launch](#)^[371] function returns the [ConnectionId](#)^[381] as part of the [ACI Launch_Params](#)^[385] structure.

After establishing the connection, all the ACI functions following the ACI_SetConnection function will work exclusively with the established connection. 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**, but not six connections.

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](#)^[381] that is the next one in order.

The ConnectionId can be always checked by executing the function [ACI_GetConnection](#)^[369].

8.5.1.30 ACI_SetDevice

[ACI_FUNC ACI_SetDevice](#)([ACI_Device_Params](#)^[381] * 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](#)^[108] command list if the selected device type has ever been selected earlier, but the parameters listed above were changed during the programming session.

8.5.1.31 ACI_SetProgOption

[ACI_FUNC ACI_SetProgOption](#)([ACI_ProgOption_Params](#)^[389] * params);

Description

This function sets device-specific options and parameters, which are specified in the [Device and Algorithm Parameters Editor](#)^[93] window. As an example see this window for one of the microcontrollers below.

Device and Algorithm Parameters Editor		
Edit	MinValue	MaxValue
DefaultValue	AllDefault	
Name	Value	Description
Device Parameters		
Fuse Bits	...	Fuses
Lock bits	...	Lock bits
Calibration Byte	00h	Calibration value for the internal RC Oscillator
Algorithm Parameters		
Algorithm	"In-System Programming"	Programming algorithm
Oscillator Frequency	2500 kHz	Oscillator frequency
Delay after Vcc is On	120 ms	Delay after Vcc is On
Programming Mode	...	Programming Mode
Vcc	5.00 V	Power supply voltage

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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes.

See also the [ACI_ProgOption_Params](#)^[389] structure description.

8.5.1.32 ACI_SetProgrammingParams

```
ACI_FUNC ACI_SetProgrammingParams(ACI\_Programming\_Params[393] * params);
```

Description

This function sets programming parameters specified in the tab [Option](#)^[109] of the [Program Manager](#)^[105] window (memory buffer configurations, programming options, test of the device insertion, etc.).

See also the [ACI_Programming_Params](#)^[393] structure description.

8.5.1.33 ACI_SettingsDialog

```
ACI_SettingsDialog();
```

Description

This macro opens the [Configure > Preferences](#)^[78] 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](#)^[78] setting dialog will appear on the computer screen, thus allowing manipulations in the dialog.

8.5.1.34 ACI_StartFunction

```
ACI_FUNC ACI_StartFunction(ACI_Function_Params[383] * params);
```

Description

This function launches one of the programming operation (**Read**, **Erase**, **Verify**, etc.) specified by the [ACI_Function_Params](#)^[383] 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](#)^[378] is different from the [ACI_ExecFunction](#)^[367]. It is possible to check if the operation has completed by the [ACI_GetStatus](#)^[371] function call. This allows monitoring the execution of programming operations if they last for a long time.

8.5.1.35 ACI_TerminateFunction

```
ACI_FUNC ACI_TerminateFunction();
```

Description

This function terminates a current programming operation initiated by the [ACI_StartFunction](#)^[378] call.

8.5.1.36 ACI_WriteLayer

```
ACI_FUNC ACI_WriteLayer(ACI_Memory_Params[388] * 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](#)^[367] or [ACI_StartFunction](#)^[378] with appropriate attributes.

8.5.2 ACI Structures

This sections contains **alphabetical list** of all ACI structures.

8.5.2.1 ACI_Buffer_Params

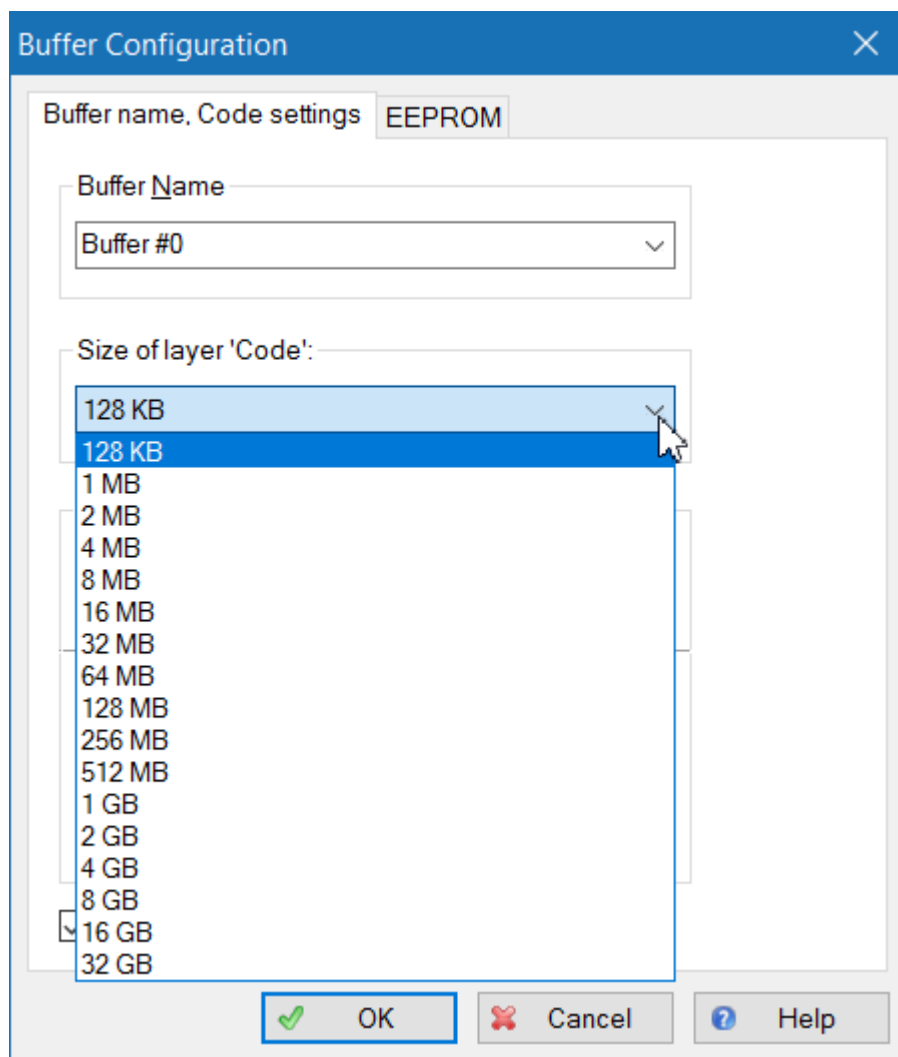
```
typedef struct tagACI_Buffer_Params
{
    UINT    Size;                // (in)  Size of structure, in bytes
    DWORD   Layer0SizeLow;      // (in/out) Low 32 bits of Layer 0 size, in bytes
```

```

DWORD Layer0SizeHigh; // (in/out) High 32 bits of Layer 0 size, in bytes
                        // Layer size is rounded up to a nearest value supported by p
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 ^[367] function this structure member represents the name of the buffer that will be created. If used with the ACI_ReallocBuffer ^[374] function will be ignored.
BufferNumber	After calling the ACI_CreateBuffer ^[367] function this structure member returns the created buffer's number. After calling the ACI_ReallocBuffer ^[374] 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](#)^[367], [ACI_ReallocBuffer](#)^[374]

8.5.2.2 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](#)^[371], [ACI_SaveConfigFile](#)^[374]

8.5.2.3 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, internally used, ACI parameter.
---------------------	---

See also: [ACI_SetConnection](#)^[376], [ACI_GetConnection](#)^[369].

8.5.2.4 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](#)^[376], [ACI_GetDevice](#)^[369]

8.5.2.5 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
                                   // call to ACI function
} ACI_ErrorString_Params;
```

ErrorString	String describing the error returned by the last ACI function call.
--------------------	---

See also: [ACI_ErrorString](#)^[367]

8.5.2.6 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 <code>aciprog.h</code> header file (see below).
StartAddressLow, StartAddressHigh	1) If used with the ACI_FileSave ³⁶⁸ 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 ³⁶⁸ 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 ³⁶⁸ 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 // Angstrom SAV
#define ACI_PLF_ASCII_HEX     8 // ASCII Hex
#define ACI_PLF_ASCII_OCTAL   9 // ASCII Octal

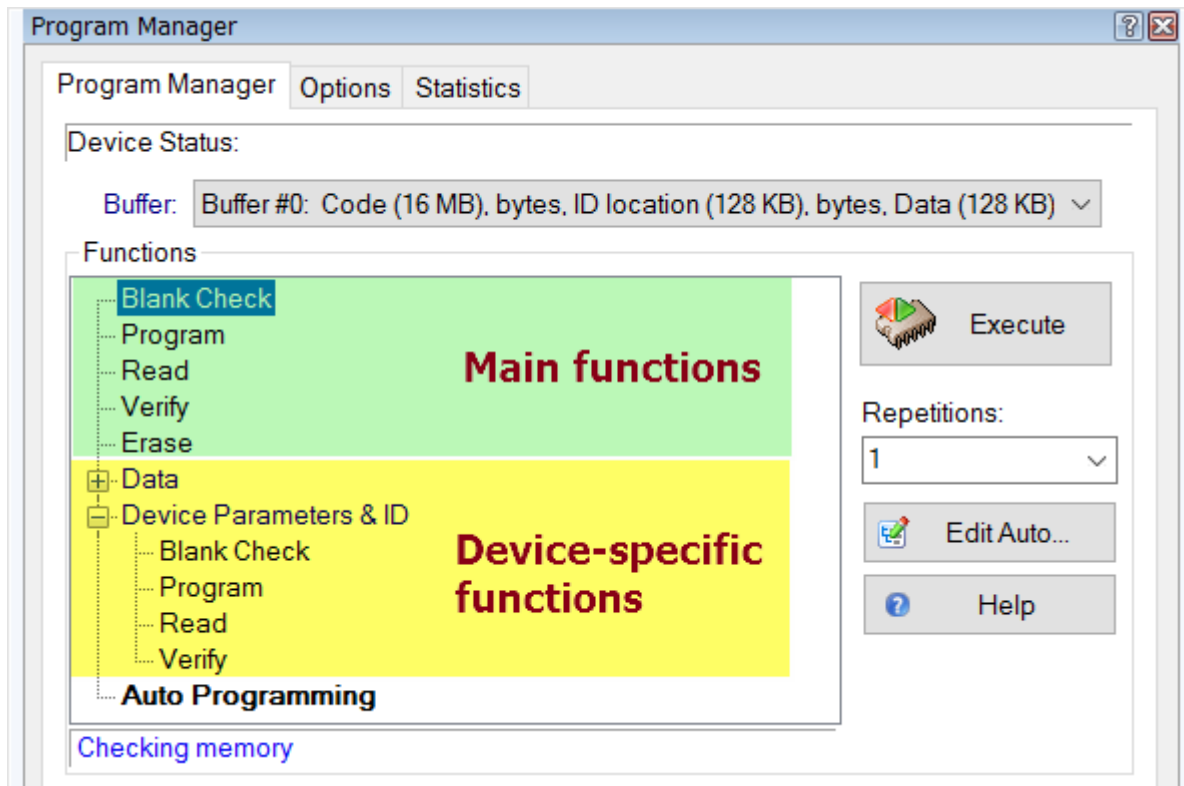
```

See also: [ACI FileLoad](#)^[368], [ACI FileSave](#)^[368].

8.5.2.7 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
                                //        To execute Auto Programming, set FunctionName to NULL,
    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	<p>The name of the CPI2-B1 function is one of those listed in the window Functions of the ChipProg-02 Program Manager tab^[107]. 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).</p> <p>To launch the AutoProgramming batch set the FunctionName either to NULL, a blank string, or the "Auto Programming".</p> <p>There is no restrictions in use of uppercase and lowercase characters in the function names.</p>
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](#)^[367], [ACI_StartFunction](#)^[378], [ACI_GetStatus](#)^[371]

8.5.2.8 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_GetStatus
} 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 ^[369] function is launched. The site (socket) numbers begin from #0.
BufferNumber	The ordinal number of the memory buffer ^[95] , content of which is required by the ACI_GangStart ^[369] 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 ^[371] function to receive the error message.

See also: [ACI_GangStart](#)^[369], [ACI_GetStatus](#)^[371]

8.5.2.9 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 op
    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 ^[369] 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 ^[369] succeeded. In case of successful termination an operation <i>before</i> expiring the Timeout the SiteStopped parameter sets TRUE. Otherwise, it will be set FALSE.

See also: [ACI_GangTerminateFunction](#)^[369], [ACI_TerminateFunction](#)^[378].

8.5.2.10 ACI_Launch_Params

```
typedef struct tagACI_Launch_Params
{
    UINT    Size;                // (in)  Size of structure, in bytes
    LPCSTR  ProgrammerExe;       // (in)  Programmer executable file name
    LPCSTR  CommandLi ne;       // (in)  Optional programmer command-line parameters
    UINT    DebugMode;          // (in)  Debug mode. See DM_xxx constants
    UINT    NumSi tes;           // (out) For Gang mode: Number of sites
    LPVOID  Connecti onId;       // (out) Connection identifier
    CHAR    ProgrammerName[64]; // (out) Programmer name
} ACI_Launch_Params;
```

ProgrammerExe	<p>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.</p> <p>The target folder name, where the the UprogNT2.EXE file resides, is defined by the parameter "Folder" of the ""HKLM\SOFTWARE\Phyton\Phyton ChipProg-02 Programmer\x.yy.zz" key. It is supposed that multiple ChipProg-02 versions can be installed on the host computer.</p>
CommandLine	<p>This structure member specifies the Command line options^[120]. One of the option is NULL (no keys). If the host computer drives a cluster^[197] of multiple programmers then the only way to launch a certain programmer is to specify the /N<serial number> for the CommandLine structure member.</p>
DebugMode	<p>The following options can be set for this parameter in accordance to the declaration in the ACIProg.h file:</p> <p>DM_GUI_HIDDEN - makes the CPI2-B1 GUI^[48] invisible. The software stops issue the messages that are normally visible in the GUI except the messages about critical system errors (for example, if a PC loses connection with the CPI2-B1 hardware).</p> <p>DM_GUI_VISIBLE - makes the CPI2-B1 GUI^[48] visible. This constant enables to manipulate with the CPI2-B1 programmer in between of the ACI function calls. These manipulations includes, but not limited to, opening and resizing windows, watching and modifying data in buffers, executing commands, etc.</p> <p>DM_GUI_NONE - completely disables any operation with the GUI that becomes invisible. No messages, even those which warn about very critical system errors, will be issued to the computer screen. The user's application, that controls the programmer via the ChipProg-ISP2 ACI, should completely handle all the errors issued by the ACI functions.</p>
NumSites	<p>If multiple CPI2-B1 programmers were launched in the gang^[197] mode, then after return the NumSites indicates the number of device programmers in the gang cluster.</p>
ConnectionId	<p>After return with the ACI_ERR_SUCCESS code this field contains the connection identifier - see the ACI_SetConnection^[376] and ACI_GetConnection^[369] functions..</p>
ProgrammerName	<p>After return with the ACI_ERR_SUCCESS code this field contains a string with the device programmer name, here CPI2-B1.</p>

See also: [ACI_Launch](#)^[371]

8.5.2.11 ACI_Layer_Params

```
typedef struct tagACI_Layer_Params
{
```

```
    UI NT    Si ze;                // (in)  Size of structure, in bytes
    UI NT    BufferNumber;         // (in)  Number of buffer of interest, the first buffer number
    UI NT    LayerNumber;         // (in)  Number of layer of interest, the first layer number
    DWORD    LayerSi zeLow;       // (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
DWORD DeviceBufStartAddrHigh;  // (out) High 32 bits of device memory start address in buffer
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 ^[95] , content of which is required by the ACI_GetLayer ^[370] function. Numbers of CPI2-B1 memory buffers begin from #0.
LayerNumber	The ordinal number of the layer in the memory buffer ^[95] , the content of which is required by the ACI_GetLayer ^[370] 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 ^[95] - 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 ^[374] 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.

BufferNumber	The ordinal number of the memory buffer ^[95] , content of which is required by the ACI_GetLayer ^[370] function. Numbers of CPI2-B1 memory buffers begin from #0.
BufferName	The name of the memory buffer as it was defined in the CPI2-B1 interface or by the ACI_CreateBuffer ^[367] 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](#)^[370]

8.5.2.12 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)
    DWORD   AddressHigh;    // (in)    High 32 bits of address, in layer units (natural to device)
    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
    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 ^[368] function.
Data	Since these are used with different ACI functions this structure member has different meanings. In case of use with the ACI_ReadLayer ^[374] 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 ^[376] - 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 ^[368] 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 ^[368] 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](#)^[374], [ACI_WriteLayer](#)^[378], [ACI_FillLayer](#)^[368]

8.5.2.13 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 "ListName^OptionName"
    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
    {
        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_BITSTREAM
                                //   in: Size of buffer pointed by String if OptionType == ACI_PO_STRING
                                //   out: Size of buffer needed for storing Bitstream data if OptionType == ACI_PO_BITSTREAM
                                //       Set Value.Bitstream to NULL to get buffer size without setting
                                //   out: Size of buffer needed for storing String if OptionType == ACI_PO_STRING
                                //       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 ^[370] 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 ^[370] function this structure member returns the option description.														
ListString	After executing the ACI_GetProgOption ^[370] 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 ^[370] function this structure member returns the option's presentation format - for example: integer, floating point, list, bitstream, etc.. See the ACI_PO_*** constant description in the aciprog.h header file below.														
ReadOnly	Setting ReadOnly =TRUE disables modification of the option specified by the ACI_GetProgOption ^[370] function.														
Value	<p>Use of this union depends on the ACI_PO_LONG* option type as it is shown in the matrix below:</p> <table border="1"> <thead> <tr> <th>Option type</th><th>Use of the Value union</th></tr> </thead> <tbody> <tr> <td>ACI_PO_LONG</td><td>The option is in the Value.LongValue</td></tr> <tr> <td>ACI_PO_FLOAT</td><td>The option is in the Value.FloatValue</td></tr> <tr> <td>ACI_PO_STRING</td><td>The option is represented as a string, the pointer on which is in the Value.String. See the note below^[391].</td></tr> <tr> <td>ACI_PO_CHECKBOXES</td><td>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^[391].</td></tr> <tr> <td>ACI_PO_LIST</td><td>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^[393].</td></tr> <tr> <td>ACI_PO_BITSTREAM</td><td>Stream of bits. This option type is not in use yet but can be used for future applications.</td></tr> </tbody> </table>	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 ^[391] .	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 ^[391] .	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 ^[393] .	ACI_PO_BITSTREAM	Stream of bits. This option type is not in use yet but can be used for future applications.
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 ^[391] .														
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 ^[391] .														
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 ^[393] .														
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 ^[391] .														
Mode	<p>Mode of using of the structure member Value (See the description of the ACI_PP_*** constants in the aciprog.h<) header file:</p> <table border="1"> <thead> <tr> <th>The Mode setting (value)</th><th>Use of the parameter Value</th></tr> </thead> <tbody> <tr> <td>ACI_PP_MODE_VALUE</td><td>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.</td></tr> <tr> <td>ACI_PP_MODE_DEFAULT_VALUE</td><td>1) If used with the ACI_GetProgOption^[370] function it issues a command to put the default Option value into the Value. 2) If used with the ACI_SetProgOption^[370] function, the Value will be ignored; the Option will be set to the default level defined in the CPI2-B1 hardware.</td></tr> <tr> <td>ACI_PP_MODE_MIN_VALUE</td><td>1) If used with the ACI_GetProgOption^[370] function it commands to put the minimal Option value into the Value. 2) If used with the ACI_SetProgOption^[370] function the</td></tr> </tbody> </table>	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 ^[370] function it issues a command to put the default Option value into the Value . 2) If used with the ACI_SetProgOption ^[370] 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_VALUE	1) If used with the ACI_GetProgOption ^[370] function it commands to put the minimal Option value into the Value . 2) If used with the ACI_SetProgOption ^[370] function the						
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 ^[370] function it issues a command to put the default Option value into the Value . 2) If used with the ACI_SetProgOption ^[370] 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_VALUE	1) If used with the ACI_GetProgOption ^[370] function it commands to put the minimal Option value into the Value . 2) If used with the ACI_SetProgOption ^[370] 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_VALUE	1) If used with the ACI_GetProgOption ^[370] function it commands to put the maximal Option value into the Value . 2) If it is used with the ACI_SetProgOption ^[370] 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

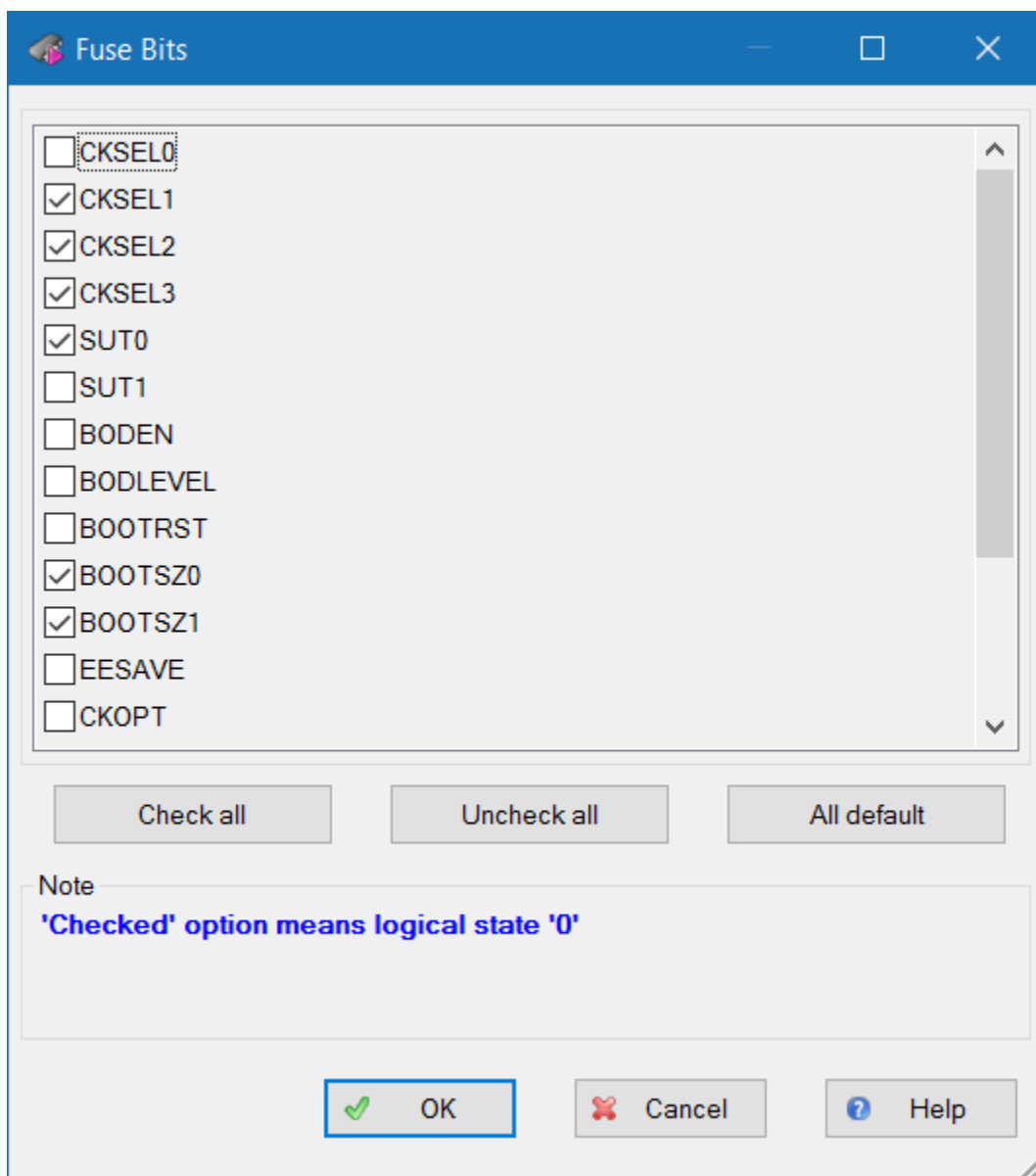
```
#define ACI_PO_LONG           0 // Signed integer option
#define ACI_PO_FLOAT         1 // Floating point option
#define ACI_PO_STRING        2 // String option
#define ACI_PO_CHECKBOXES    3 // 32-bit array of bits
#define ACI_PO_LIST          4 // List (radiobuttons)
#define ACI_PO_BITSTREAM     5 // Bit stream - variable size bit array
```

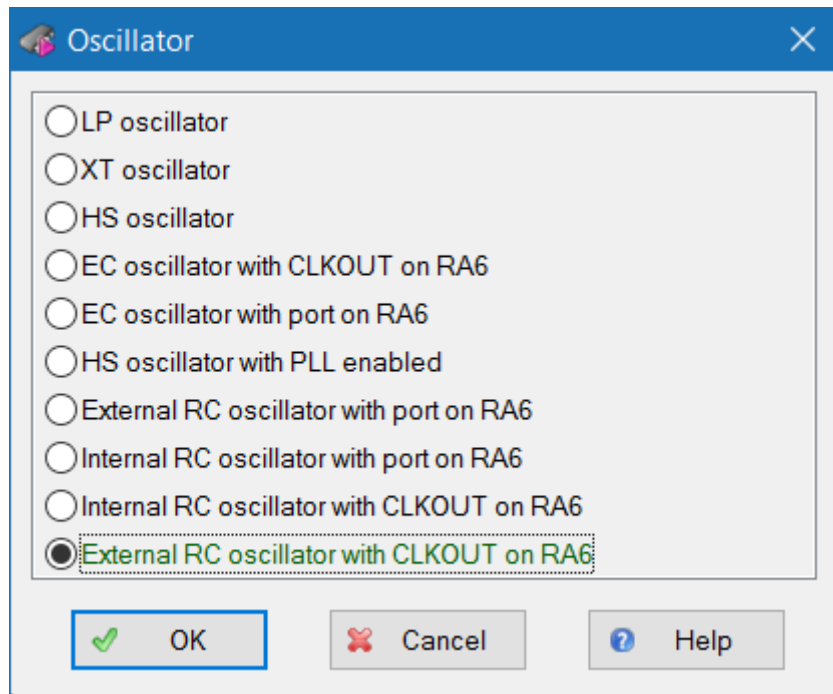
****// ACI Programming Option Mode constants for ACI_GetProgOption()/ACI_SetProgOption()**

```
#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.





See also: [ACI_GetProgOption](#)^[370], [ACI_SetProgOption](#)^[376]

8.5.2.14 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
    BOOL    DialogBoxOnError;    // (in || out) On error, display dialog box
    UINT    AutoDetectAction;    // (in || out) Action to perform on device autodetect or 'S
    DWORD   DeviceStartAddrLow;  // (in || out) Low 32 bits of device start address for prog
    DWORD   DeviceStartAddrHigh; // (in || out) High 32 bits of device start address for prog
    DWORD   DeviceEndAddrLow;    // (in || out) Low 32 bits of device end address for program
    DWORD   DeviceEndAddrHigh;   // (in || out) High 32 bits of device end address for program
    DWORD   DeviceBufStartAddrLow; // (in || out) Low 32 bits of device memory start address i
    DWORD   DeviceBufStartAddrHigh; // (in || out) High 32 bits of device memory start address i
} 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 ^[194] before executing the Program ^[195] command.										
VerifyAfterProgram	This is the command to verify ^[197] the data written into the device every time after executing the Program ^[195] command.										
VerifyAfterRead	This is the command to verify ^[197] the data written into the device every time after executing the Read ^[195] command.										
SplitData	This is the command to split ^[110] data in accordance with the value of the constants ACI_SP_*** 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 ^[176] or a batch of single operations programmed in the Auto Programming ^[108] 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)	<p>If the DeviceAutoDetect is TRUE then values of the ACI_AD_*** 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)</p> <table border="1"> <thead> <tr> <th>AutoDetectAction value</th><th>What to do (action)</th></tr> </thead> <tbody> <tr> <td>ACI_AD_EXEC_FUNCTION</td><td>Launch the programming operation (function) currently highlighted in the Program Manager tab^[107].</td></tr> <tr> <td>ACI_AD_EXEC_AUTO</td><td>Launch a batch of single operations programmed in the Auto Programming^[108] dialog.</td></tr> <tr> <td>ACI_AD_EXEC_SCRIPTIPT</td><td>Perform the script specified in the Script File^[176] 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^[367] or ACI_StartFunction^[378].</td></tr> </tbody> </table>	AutoDetectAction value	What to do (action)	ACI_AD_EXEC_FUNCTION	Launch the programming operation (function) currently highlighted in the Program Manager tab ^[107] .	ACI_AD_EXEC_AUTO	Launch a batch of single operations programmed in the Auto Programming ^[108] dialog.	ACI_AD_EXEC_SCRIPTIPT	Perform the script specified in the Script File ^[176] dialog.	ACI_AD_DO_NOTHING	Do not act (ignore). Then it is possible to resume operations only by executing either the ACI_ExecFunction ^[367] or ACI_StartFunction ^[378] .
AutoDetectAction value	What to do (action)										
ACI_AD_EXEC_FUNCTION	Launch the programming operation (function) currently highlighted in the Program Manager tab ^[107] .										
ACI_AD_EXEC_AUTO	Launch a batch of single operations programmed in the Auto Programming ^[108] dialog.										
ACI_AD_EXEC_SCRIPTIPT	Perform the script specified in the Script File ^[176] dialog.										
ACI_AD_DO_NOTHING	Do not act (ignore). Then it is possible to resume operations only by executing either the ACI_ExecFunction ^[367] or ACI_StartFunction ^[378] .										
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.										

<code>DeviceBufStartAddrLow,</code> <code>DeviceBufStartAddrHigh</code>	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 `aciprogram.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](#)^[377], [ACI_GetProgrammingParams](#)^[371]

8.5.2.15 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](#)^[373].

8.5.2.16 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 0
    BOOL    Executing;           // (out) The function started by ACI_StartFunction() is executing
    UINT    PercentComplete;     // (out) Percentage of the function completion, valid if 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.
    BOOL    FunctionFailed;      // (out) TRUE if last function failed
```

```

CHAR  FunctionName[128];    // (out) Name of a function being executed if Executing != FALSE
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 ^[37] 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 ^[37] , 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)	<p>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:</p> <table border="1"> <thead> <tr> <th>Status</th><th>Description</th></tr> </thead> <tbody> <tr> <td>ACI_DS_OK</td><td>The device is in the socket and the device's leads are reliably gripped by the programmer's ZIF socket's sprung contacts.</td></tr> <tr> <td>ACI_DS_OUT_OF_SOCKET</td><td>There is no device in the programmer's ZIF socket.</td></tr> <tr> <td>ACI_DS_SHIFTED</td><td>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^[58] does not correspond to the type of chip in the programmer's socket.</td></tr> <tr> <td>ACI_DS_BAD_CONTACT</td><td>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.</td></tr> <tr> <td>ACI_DS_UNKNOWN</td><td>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.</td></tr> </tbody> </table>	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 ^[58] 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.
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 ^[58] 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 ^[37] 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 FunctionFailed == TRUE.
ErrorMessage	The destination of the error message if the function fails, i.e. the FunctionFailed == 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](#)^[367], [ACI_StartFunction](#)^[378], [ACI_GetStatus](#)^[371]

Index

- _ -

_ff_attrib 264
 _ff_date 264
 _ff_name 264
 _ff_size 265
 _ff_time 265
 _fmode 359
 _fullpath 265
 _GetWord 265
 _printf 266

- A -

About
 software version 90
 abs 266
 ACI 168, 172
 DLL 158
 External application 158
 External control 158
 ACI examples 165
 ACI functions
 ACI structures 160
 ACI structures
 ACI functions 164
 acos 266
 ActivateWindow 266
 Add Watch
 dialog 184
 AddButton 266
 AddrExpr 267
 AddWatch 267
 Algorithm Parameters 93
 AllProgOptionsDefault 238
 Alphabetical List of Script Language Built-in Functions
 and Variables 256
 Alternate Forms for printf Conversion 322
 Angstrom SAV 103
 API 268
 Application Control Interface
 ACI 158
 ACI functions 158

 ACI header 158
 ACI structures 158
 DLL 158
 External application 158
 External control 158
 Programming automation 158
 Application Control Interface examples 165
 ApplName[] 359
 Arrays 227
 ASCII Hex 103
 asin 268
 atan 268
 ATE control 24
 atof 268
 atoi 269
 Auto Programming 108
 Automatic Word Completion 189
 AutoWatches
 pane 182
 AutoWatches pane 182

- B -

BackSpace 269
 Backspace unindents 83
 Basic Data Types 209
 Basic Types 227
 Binary image 103
 Blank 243
 Blank Check 243
 Block Operations 187
 BlockBegin 269
 BlockCol1 360
 BlockCol2 360
 BlockCopy 269
 BlockDelete 269
 BlockEnd 270
 BlockFastCopy 270
 BlockLine1 360
 BlockLine2 360
 BlockMove 270
 BlockOff 270
 BlockPaste 270
 Blocks 83, 187
 copying / moving 187
 line blocks 187
 non-persistent blocks 187
 persistent 83

Blocks 83, 187
 persistent blocks 187
 standard blocks 187
 vertical 83
 vertical blocks 187
 BlockStatus 360
 Buffer 18, 243
 Buffer access functions 232
 Buffer Configuration
 dialog 61
 Buffer Dump
 window 95
 Buffer layer 18
 Buffers 61
 dialog 61
 memory allocation 61

- C -

Calculator
 dialog 88
 CallLibraryFunction 270
 CaseSensitive 361
 ceil 271
 Character constants 209
 Character operation functions 247
 chdir 271
 Check 243
 Check Blank 194
 Check Sum 243
 CheckSum 69, 232, 243, 271
 ChipProg
 main menu 50
 ChipProg-ISP
 software characteristics 21
 ChipProg-ISP2 19
 chsize 271
 ClearAllBreaks 272
 ClearBreak 272
 ClearBreaksRange 272
 clearerr 272
 ClearWindow 273
 CLI 18
 close 273
 CloseWindow 273
 Colors 80
 tab 80
 Command line 18, 168, 169
 Command Line Interface 18
 Command Line Keys 120
 Command Line Mode 18
 Command line options 120, 125
 Command Line Parameters 120
 Commands
 menu 86
 Commands Menu 86
 Comments 206
 Composite operator 220
 Condensed Mode 188
 Condensed Mode Setup
 dialog 194
 Conditional Compilation 231
 Conditional Operator If-Else 222
 Configuring Editor
 dialog 83
 Configuration 57
 buffer 61
 editor Options 57
 environment 57
 Configuration Files 52
 Configuration Menu 57
 Configure the device to be programmed 196
 Configuring a Buffer
 dialog 97
 Confirm Replace
 dialog 192
 Console
 window 104
 Window Console 104
 cos 274
 CPI2-B1 19
 hardware characteristics 20
 CPI2-B1 major features
 brief characteristics 20
 Cr 274
 creat 274
 creatnew 275
 creattemp 275
 CurChar 276
 CurCol 361
 Curcuit 276
 CurLine 361
 Custom signature 245
 Cycle Operator Do-While 224
 Cycle Operator For 225
 Cycle Operator While 224

- D -

- Data byte order 210
- data caching 134
- Debug shell control functions 252
- Declaration: 285
- Define Font 80
- Define key 81
- Definitions
 - adapter 17
 - buffer 17
 - memory buffer 17
 - sub-level 17
- delay 276
- DelChar 276
- DellLine 277
- Description 285
- Description of Script Language 205
- Descriptions 227
- DesktopName[] 361
- Device and Algorithm Parameters
 - window 93
- Device Information
 - window 92
- Device Parameters 93
- Device programming control functions 237
- Device serialization 68
- Difference Between the Script Language and the C Language 205
- difftime 277
- Directives of the Script Language Preprocessor 230
- Discard device 68
- Discard serial numbers 68
- Display from address
 - dialog 100
- Display from Line Number
 - dialog 194
- Display Watches Options
 - dialog 183
- DisplayText 277
- DisplayTextF 277
- DLL 168, 172
- Down 278
- dup 278
- dup2 278
- DUT 22

- DUT connection 22

- E -

- Edit Information to be programmed 196
- Edit Key Command
 - dialog 86
- Editor Key Mapping
 - tab 85
- Editor window 186
- Ellipse 279
- Environment
 - dialog 79
- eof 279
- Eol 280
- Erase 195
- errno 361
- Ethernet 22, 74
- Ethernet settings B1 122
- Even byte 110
- Event Wait Functions 255
- Examples of ACI use 165
- Examples of Expressions 204
- exec 280
- ExecFunction 238
- ExecMenu 280
- ExecScript 281
- exit 281
- ExitProgram 282
- exp 282
- Expr 282
- Expressions 202
- External Object Description 229

- F -

- fabs 282
- fclose 282
- fdopen 283
- feof 283
- ferror 284
- fflush 284
- fgetc 284
- fgets 285
- File format 103
- File Menu
 - overview 51

FileChanged 285
 filelength 285
 filelength returns the length (in bytes) of the file associated with handle. 285
 fileno 285
 FillRect 286
 findfirst 286
 findnext 286
 FindWindow 287
 FirstWord 287
 FloatExpr 287
 Floating-point constants 208
 floor 287
 fmod 288
 fnmerge 263
 fnsplit 288
 Fonts 80
 tab 80
 fopen 288
 Format 206
 Format and nesting 220
 Formatted input-output functions 250
 ForwardTill 289
 ForwardTillNot 289
 fprintf 289
 fputc 290
 fputs 290
 FrameRect 290
 fread 291
 FreeLibrary 291
 freopen 291
 frexp 292
 fscanf 292
 fseek 293
 ftell 293
 Functions for file and directory operation 247
 fwrite 294

- G -

GangExecute 239
 GangGetError 239
 GangStatus 239
 GangWaitComplete 239
 General Editor
 settings 83
 General syntax of the script file language 206
 GetBadDeviceCount 240

GetByte 233, 294
 getc 294
 getcurdir 295
 getcwd 295
 getdate 295
 getdfree 296
 getdisk() 296
 GetDword 233, 299
 getenv 296
 GetFileName 296
 getftime 296
 GetGoodDeviceCount 240
 GetLine 297
 GetMark 297
 GetMemory 233, 297
 GetProgOptionBits 240
 GetProgOptionFloat 240
 GetProgOptionList 240
 GetProgOptionLong 241
 GetProgOptionString 241
 Gets file size in bytes. 285
 GetScriptFileName 298
 gettime 298
 getw 299
 GetWindowHeight 299
 GetWindowWidth 299
 GetWord 234
 Global Variable Definition 228
 GotoXY 300
 Graphical output functions 253
 GUI 48

- H -

Help
 menu 90
 On-line 28
 Highlight
 multi-line Comments 83
 Highlight Active Tabs 82
 Highlighting
 Syntax 83, 189
 History file 52
 Holtek OTR 103
 Hot Keys 81
 How to Get On-line Help 28
 How to start a script file 177
 How to write a script file 184

HStep 300

- I -

I/O Stream

 window 180

I/O Stream window operation functions 254

ICP 17

Identifier Change (#define) 230

Identifiers 207

Inclusion of Files (#include) 230

inport 301

inportb 301

InsertMode 362

Inspect 301

Install ChipProg 32

Install the ChipProg Software 32

Integer constants 207

Introduction 17

InvertRect 301

isalnum 302

isalpha 302

isascii 302

isatty 302

isctrl 303

isdigit 303

isgraph 303

islower 303

ISP

 ISP HV Mode 17

 ISP Mode 17

isprint 304

ispunct 304

isspace 304

isupper 304

isxdigit 304

itoa 305

- J -

JEDEC 103

Job 133

- L -

LabVIEW 168, 169, 172, 173

LabVIEW Integration 173

LAN 22, 74

LastChar 305

LastEvent 305

LastEventInt{1...4} 306

LastFoundString 362

LastMemAccAddr 362

LastMemAccAddrSpace 362

LastMemAccLen 362

LastMemAccType 362

LastMessageInt 363

LastMessageLong 363

LastString 306

layer 18, 61

Left 307

LineTo 306

Load file

 dialog 102

Load session 52

Load the file into the buffer 195

LoadDesktop 306

LoadLibrary 307

LoadOptions 307

LoadProgram 234, 307

LoadProject 308

Local Variable Definition 228

lock 359

locking 308

log 309

Log file 73

log10 309

long filelength(long handle); 285

Long integer constants 208

lseek 309

ltoa 310

- M -

Main menu

 commands 50

Main menu bar 50

MainWindowHandle 363

Mapping

 hot keys 81

Mathematical functions 245

MaxAddr 235, 310

memccpy 310

memchr 310

memcmp 311

- memcpy 311
- memicmp 311
- memmove 312
- Memory Dump Window Setup
 - dialog 98
- Memory Blocks
 - operations 100
- Memory layer 18
- memset 312
- Menu
 - Project 52
 - View 52
- Menu File 51
 - load file 51
 - save file 51
- Menu Help 90
- Menu Script 88
- Message box
 - always display 82
- MessageBox 312
- MessageBoxEx 312
- Messages
 - tab 82
- MinAddr 235, 313
- Miscellaneous Settings 82
- mkdir 313
- Modify Address
 - dialog 100
- Modify Memory
 - dialog 100
- Motorola S-record 103
- MoveTo 314
- MoveWindow 314
- movmem 314
- mprintf 241
- Multi-File Search Results
 - dialog 192
- Multi-programming mode 197

- N -

- NumWindows 363

- O -

- Odd byte 110
- On success

- EBADF Bad file number 285
- On-line Help 28
- On-the-Fly
 - On-the-Fly Command Line Options 127
 - On-the-Fly Options 127
- On-the-Fly Control
 - Example 132
 - On-the-Fly Control utility 126
- On-the-Fly utility return codes
 - return codes 131
- open 314
- Open Project 54
 - dialog 54
- OpenEditorWindow 315
- OpenProject 241
- OpenStreamWindow 315
- OpenUserWindow 316
- OpenWindow 316
- Operands 204
- Operations and Expressions
 - About 210
 - Arithmetic Conversions in Expressions 219
 - Arithmetic Operations 211
 - Array Operations 216
 - Assignment Operations 212
 - Bit Operations 216
 - Logical Operations 215
 - Operand Execution Order 219
 - Operand Metadesignation 211
 - Operation Execution Priorities and Order 218
 - Other Operations 217
- Operations with Expressions 202
- Operations with Memory Blocks 100
- Operator Break 221
- Operator Continue 222
- Operator Goto 222
- Operator label 220
- Operator Return 222
- Operator-expression 221
- Operators 220
- Options
 - dialog 78
- Options&split
 - dialog 109
- Other Various Functions 255
- outport 317
- outportb 317
- Overview

Overview
 User Interface 48

- P -

Packages/Adapters 58
 peek 317
 peekb 317
 POF 103
 poke 317
 pokeb 318
 Polyline 318
 pow 318
 pow10 318
 Predefined Symbols in the Script File Compilation 231
 Preferences 78
 PRG 103
 printf 319
 printf Conversion Type Characters 319
 ProgOptionDefault 241
 Program a Device 195
 Program Manager 107
 Auto Programming 107
 dialog 107
 Operation Progress 107
 window 105
 Programmer 17
 work with 194
 Programming
 check blank 194
 configure the device 196
 edit Information 196
 erase 195
 load the file 195
 program a Device 195
 read a device 195
 save the data 197
 verify 197
 write Information into the Device 196
 Programming automation 158
 Project 47
 Project Menu 52
 Project Options 47, 53
 dialog 53
 Project Repository
 dialog 56
 Projects 47

pscanf 325
 putc 326
 putenv 326
 putw 326

- Q -

Quick Start 28
 Quick Watch
 enabled 82
 Quick Watch Function 190

- R -

rand 327
 random 327
 randomize 327
 read 327
 Read a Device 195
 ReadShadowArea 241
 Rectangle 328
 RedrawScreen 328
 Regular Expressions
 search for 193
 RegularExpressions 363
 Relation Operations 214
 ReloadProgram 235, 328
 Remote control 120
 RemoveButtons 328
 rename 329
 Replace Text
 dialog 191
 Repository 56
 Response files 125
 Returned Value 285
 rewind 329
 Right 329
 rmdir 329
 Run ChipProg 32

- S -

Save file from buffer
 dialog 104
 Save session 52
 Save the data read out from a device 197
 SaveData 235, 330

- SaveDesktop 330
- SaveFile 331
- SaveOptions 331
- scanf 331
- Script 176, 178, 204
 - menu 88
- Script file manipulation functions 250
- Script Files 176, 204
 - dialog 178
- Script Language Built-in Functions 231
- Script Language Built-in Variables 256
- Script source window
 - open 178
- SD card 134
- Search 332
- Search for Regular Expressions 193
- Search for Text
 - dialog 190
- Search mask 58
- searchpath 332
- SearchReplace 333
- Select color 80
- Select device 58
 - dialog 58
- SelectBrush 333
- SelectedString[] 364
- SelectFont 333
- SelectPen 333
- Serial number 68
- Serialization 69
- Serialization, Checksum, Log file
 - dialog 63
- Set/Retrieve Bookmark
 - dialog 193
- SetBkColor 334
- SetBkMode 334
- SetBreak 334
- SetBreaksRange 334
- SetByte 236, 334
- SetCaption 335
- SetDevice 236
- setdisk 335
- SetDword 236, 335
- SetFileName 335
- setftime 336
- SetMark 336
- setmem 336
- SetMemory 237, 337
- setmode 337
- SetPixel 337
- SetProgOption 242
- SetTextColor 337
- SetToolbar 338
- SetUpdateMode 338
- SetWindowFont 338
- SetWindowSize 339
- SetWindowSizeT 339
- SetWord 237, 339
- Signature 245
- Signature String 70
- Simple example of a script file 176
- sin 340
- Sounds 78
- Split data 110
- sprintf 340
- sqrt 340
- srand 341
- sscanf 341
- Standalone 132, 133
- Stand-Alone 132
- Standalone Mode 132
- Standalone Operation 132
- Standard/Extended Intel HEX 103
- Start Address 243
- Startup 38
- Static IP address 74
- Statistics
 - dialog 111
- Step 341
- Stop 341
- stpcpy 342
- strcat 342
- strchr 342
- strcmp 342
- strcmpi 343
- strcpy 343
- strcspn 343
- Stream file functions 249
- stricmp 343
- String operation functions 246
- strlen 344
- strlwr 344
- strncat 344
- strncmp 345
- strncmpi 345
- strncpy 345

strnicmp 346
 strnset 346
 strpbrk 346
 strrchr 346
 strrev 347
 strset 347
 strspn 347
 strstr 347
 strtol 348
 strtoul 348
 strupr 349
 Sub-Layer 61
 additional 61
 main 61
 Sub-Layer 'Code' 61
 Sub-layer 'ID location' 61
 Syntax Highlighting 189
 System Requirements 32
 SystemDir[] 364

- T -

Tab Size 83
 tan 349
 tanh 349
 target device 22
 tell 349
 TerminateAllScripts 350
 TerminateScript 350
 Terminology 17
 Terminology and Definitions 17
 Text 350
 Text Edit 186
 Text editor functions 250
 toascii 350
 Tof 350
 tolower 351
 Toolbar
 tab 82
 toupper 351

- U -

ultoa 351
 Undo Count 83
 unlink 351
 unlock 352

Up 352
 UpdateWindow 352
 USB 22
 User
 window 180
 User Interface 48
 overview 48

- V -

Verify programming 197
 View 52
 View Menu 52

- W -

Wait 352
 WaitExprChange 353
 WaitExprTrue 353
 WaitGetMessage 354
 WaitMemoryAccess 354
 WaitSendMessage 355
 WaitStop 356
 WaitWindowEvent 356
 Watches
 window 182
 Watches Window
 add Watch 184
 display Watches Options 183
 WE_* constants 305
 wgetchar 356
 wgethex 357
 wgetstring 357
 WholeWords 364
 Window
 menu 89
 Menu Window 89
 Window Device Information 92
 Window Dump Setup
 dialog 98
 Window Editor 186
 Window I/O Stream 180
 Window Program Manager 105
 Window User 180
 Window Watches 182
 WindowHandles[] 364
 WindowHotkey 357

Windows 92
Windows operation functions and other system
functions 253
Wizard 38
Word Completion 189
WordLeft 358
WordRight 358
Work with Programmer 194
WorkFieldHeight 364
WorkFieldWidth 364
wprintf 358
write 358
Write Information into the Device 196
WriteShadowArea 242

Back Cover